HAYDEN *Macintosh Library* BOOKS

Advanced Macintosh™ Pascal

Powering Up with Files, QuickDraw, the Toolbox, and InLines

Paul Goodman

Advanced Macintosh™ Pascal

Advanced Macintosh™ Pascal

Paul Goodman

HAYDEN BOOKS
A Division of Howard W. Sams & Company
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

FIRST EDITION
FIRST PRINTING—1986

International Standard Book Number: 0-672-6570-1
Library of Congress Catalog Card Number: 86-63300

Acquisitions Editor: *Bill Grout*
Editor: *Ronnie Groff*
Cover Design: *Jim Bernard*
Cover Photo: *Lou Odor*
Composition and Production: *Publishers Network*

Printed in the United States of America

Contents

Introduction ix

Chapter 1 Macintosh System and Memory Overview 1

Macintosh Overview 2
Macintosh Pascal 4
Memory 5

Chapter 2 Advanced Pascal Structures 13

Records 14
The With Statement 16
Variant Records 18
Sets 22
Set Operations 23
  Set Union 24
  Set Intersection 24
  Set Difference 24
  Set Comparisons 24
Pointers 25

Chapter 3 Files and File Programming 33

File Concepts 33
  The File Data Type 35
Using Sequential Files 36
  Opening a Sequential File 37
  Closing a File 38

Reading Data 39
Erasing a Sequential File 40
File Processing with Sequential Files 40
Random Files 41
Opening a Random File 41
Writing and Reading Data 42
Working with Random Files 42
Finding the End of a File 43
The Haiku Writer 47
Indexed Sequential Access Method 52

Chapter 4 Events 59

Why Use Events? 59
Event Types 60
Event Records 62
Keyboard Events 64
Using Bits Masks 65
Using Events 68
Programming with GetNextEvent 70
EventAvail 72
Flush Events 74

Chapter 5 QuickDraw Programming Techniques 77

The GridEdit Program 78
Drawing Other Shapes 82
Ovals 82
Round-Cornered Rectangles 83
Simple Animation Techniques 84
The PaulPaint Program 87
PaulPaint Revisited 91
Text-Drawing Routines 94
The Three-Card Monte Program 95
Simulating Pushbuttons 97
Radio Buttons 105

Chapter 6 The InLine Routines—Accessing the Toolbox 111

Menus 113
Windows 122
More on FindWindow 128
Programming Example—Windows and Menus 133

Programmer Defined Windows 137
Miscellaneous Window Routines 140
The Controls Manager 140
Other Control Routines 152
Text Editing 153
Text Editing Summary 154
  SHIFT Click 164
  Building the Strings 165
  Input Verification 165
  Other Text Edit Routines 167
The Haiku Writer Revisited 168
Some Final Notes on Using InLines 188

Chapter 7 Advanced QuickDraw 191

GrafPorts 191
More on Drawing Text 193
  Determining Text Widths 195
Pen Characteristics 196
  Pen Location 196
  Pen Pattern 197
  Pen Mode 197
Line Drawing Routines 201
Complex Drawing Shapes 201
  Drawing a Region 204
  Disposing a Region 205
Calculations with Regions 205
Using Regions 210
Polygons 212
  Manipulating Polygons 216
Defining Custom Patterns 217
Cursors 224
Pictures 226
Calculations with Points 230
Drawing in Color 231
Additional Routines 232

Chapter 8 A Complete Application: The Logger 235

Chapter 9 **The Standard Apple Numerical Environment (SANE)** 259

Data Types 260
Arithmetic Operations 263
  Selecting a Data Type 264
Mathematical Functions 265
  Logarithm Functions 266
  Exponential Functions 266
  Financial Functions 267
  Random Number Generation 267

Appendix A **Macintosh Pascal Version 2.0** 269

The Application Shell 269
Changes to Files and Devices 270
Procedural Parameters 271
Resources 272
Memory Management 272
Other Changes 273
  Program Size 273
  Error Handling 273
  Relaxed Order of Declarations 273
  User Interface Changes 273

Appendix B **Toolbox Quick Reference** 275

Menu Manager 275
Window Manager 276
Controls 276
Text Editing 277

Appendix C **Decimal - Binary - Hexadecimal Chart** 279

Index 281

Introduction

The Macintosh with its sophisticated concepts and its rich repertoire of routines built into ROM presents an exciting programming environment to use and explore. Unfortunately, the would-be explorer has had trouble finding a suitable programming language to work with. Certainly, most programmers have outgrown BASIC, and the C language implementations for the Macintosh have been complex, tedious to use, and expensive. Many programmers have felt that they had a home with Macintosh Pascal, albeit a home without the amenities one might hope for. *Advanced Macintosh Pascal* was written for those who wish to break down the walls of that house and explore the full meaning and depth of the Macintosh. The reader of this book will already have some mastery of the Pascal language and will want to learn how to use its more advanced features or may just want to "get into the ROM." The approach of this book is broad enough for both.

The book's goals can be summarized as follows:

- To explore the Macintosh memory and memory management techniques
- To increase the reader's knowledge of Pascal structures and skills as a programmer
- To teach data file programming
- To introduce the reader to the Macintosh User Interface Toolbox and QuickDraw and explain their procedures and functions
- To show the reader how to use Macintosh Pascal to develop "real" Macintosh applications that implement windows, menus, controls and text editing.

Macintosh Pascal represents a significant advance in programming language systems. As a programming language, Macintosh Pascal is a full implementation of the ANSI Pascal standards. Add to this full IEEE numeric standards for accuracy and you have a Pascal system suitable for either scientific or business applications.

As a programming environment, Macintosh Pascal combines the power of Pascal programming with the ease of using BASIC. Implemented via an interpreter rather than a compiler, Macintosh Pascal significantly cuts program development time by eliminating long waits for compilation. Sophisticated debugging tools also reduce development time. However, because of the speed of the Macintosh, Macintosh Pascal does not suffer from the slow execution times of other interpreted languages.

As a way to explore the Macintosh, Macintosh Pascal provides a safe, easy environment to work with. Full access to the Macintosh's QuickDraw graphics package is provided via built-in routines with full Pascal-type checking. This means no fatal systems crashes will occur if you make a programming mistake using QuickDraw. Partial access is provided to the Macintosh's Toolbox, which controls the operation of the mouse and the other unique Macintosh features. Access to the remaining Toolbox routines is provided via the unprotected Inline procedures.

The only problem that programmers have faced with Macintosh Pascal has been finding information on both advanced Pascal concepts and the Macintosh itself. *Advanced Macintosh Pascal* will fill this need by introducing the reader to all these areas of Macintosh Pascal. Some knowledge of Pascal is needed but no previous exposure to the Macintosh is assumed. Chapter by chapter the reader will learn advanced Pascal concepts such as records, files, and pointers along with how to take advantage of the unique features of the Macintosh such as windowing and pulldown menus.

Chapter 1 is an overview of the Macintosh, Macintosh Pascal, and the memory structures of the Macintosh. The important memory concepts, such as the heap, the stack, and dynamic memory access through handle pointers, are covered.

Chapter 2 introduces Pascal's "power" structures: sets, pointers, and records, which are not covered in detail by most Pascal books. These structures are explained in relation to how they can be used in developing programs. Simple Toolbox routines are introduced as reinforcement and motivation.

Chapter 3 covers files and file processing. Files are the key to writing useful programs, yet information on their use and operation is scarce. This chapter teaches the techniques needed to store and retrieve data from both sequential and random files on the Macintosh. Several example programs are presented including a program that composes poetry. Lazy I/O, totally neglected in most books, is discussed in order to give the reader the full understanding of how file input/ouptut is performed. Besides data files, the important text files are covered with special emphasis placed on the use of devices such as the printer and modem.

Chapter 4 covers event programming, which is the key to developing Macintosh-like programs. Events allow a program to respond directly to user action such as keyboard and mouse input. The Toolbox routines that manage the event queue and report on events to programs are presented along with helpful programming examples.

Chapter 5 reviews the basics of QuickDraw, the Macintosh's graphics package. The Macintosh graphics coordinate system, drawing, and manipulating of geometric shapes and animation techniques are all discussed. Events and QuickDraw programming techniques are combined to simulate Macintosh controls such as pushbuttons and radio buttons. These simulations are easily transferable to other programs and are a simple way to implement Macintosh user interfaces in a program.

Chapter 6, the major focus of the book, covers the Macintosh Pascal InLine procedures and functions. Using InLine procedures, programmers can access the Toolbox routines needed to create "real" Macintosh applications that conform to the Macintosh user interface standards. Learning how to access the Toolbox is of little use without detailed information on how the Toolbox routines operate. This chapter covers four of the major features of the Toolbox: the Window Manager, the Menu Manager, the Control Manager, and the TextEdit package. This information will allow the reader to create "real" Macintosh programs utilizing multiple windows, pulldown menus, controls such as pushbuttons and check boxes, and full mouse-based text editing. The programming techniques needed to combine these concepts with event handling are carefully developed and explained.

Chapter 7 extends the reader's knowledge of QuickDraw by discussing some of the more sophisticated concepts. The information presented includes GrafPorts, control of fonts, regions, polygons, pictures, and drawing in color.

Chapter 8 presents a complete application combining QuickDraw, Events, and file handling. The program called The Logger tracks computer usage and produces a log suitable for income tax purposes.

Chapter 9 discusses the Standard Apple Numeric Environment, which includes the extended data types needed for scientific and business programming. SANE provides Macintosh Pascal with the accuracy (if not the speed) of the most powerful mainframe computers.

Pedagogically speaking, *Advanced Macintosh Pascal* teaches with the aid of small programming examples that demonstrate the concepts without confusion, and then later combines many features into larger examples. Normally, routines are introduced, explained, and then used in a program. Much care has been taken to ensure that the little questions do not go unanswered. Since it is expected that readers will combine the knowledge and insight gained from reading the book with their programs, reference sections have been added to the end of most chapters for easy review.

After reading *Advanced Macintosh Pascal*, the reader will find new doors opened when working with the Macintosh. Welcome!

Advanced Macintosh™ Pascal

CHAPTER

1

Macintosh System and Memory Overview

Macintosh Pascal is a unique programming environment. It combines the programming elegance of Pascal, the ease of execution of BASIC, unparalleled debugging facilities, and the Macintosh's familiar user interface and text editing. This combination provides a Pascal system that is easy to use but also has surprising power and flexibility.

Macintosh Pascal is implemented via an interpreter rather than the more commonly used compiler. This interpreter works by first checking the syntax of a program and then executing the program one line at a time by examining the statement and performing its action with the use of tables maintained inside the memory of the computer. There are many advantages to the use of an interpreter. Probably the greatest is the ability to perform editing, execution, and debugging in one context—that is, without having to switch programs. This is impossible when using a compiled language, normally the way Pascal is implemented. When using a compiled language, the programmer works in several different contexts in a short period of time. First, the program is written with the help of a text editor or word processor. Next, the program is saved as an ASCII file, which is then passed as input to a Pascal compiler that reads the program, checks it for syntax, and, if no mistakes exist, translates it into the machine language of the computer being used. After compilation the machine language program is not yet ready for execution and is saved in a second file passed as input to a program known as either a loader, linker, or link/loader, depending on the terminology of the particular system. The loader takes the machine language program and prepares it for execution by linking the program together with subroutines and runtime libraries. The output of the loader is placed either directly in memory or into a file for later execution. This process produces a program that executes faster than one that is interpreted but may or may not take up less memory space.

Along with this interpreter-versus-compiler debate, the developers of Macintosh Pascal had to take into account the basic difficulties inherent in running a program on the Macintosh itself and whether the programmer should have to be troubled with them. The answers to all these questions pointed to the use of an interpreter.

Macintosh Overview

The designers of the Macintosh wished to provide programmers with more than just a box containing basic computer components. They wished to include in the computer all that programmers would need to write programs based on the same building blocks. The goal was to eliminate the need for the code to accomplish these tasks and to provide the computer users with a consistent look and feel to all their programs. For instance, saving a file should work the same way in all programs. This speeds up the time it takes to learn a program and makes the program easier to remember when a user is working with several different programs.

This consistency was accomplished on the Macintosh by including in Read Only Memory a large set of already written subroutines that can be freely used by the programmer. These subroutines, most of which were originally written for the Macintosh's revolutionary predecessor, the Lisa, can be broken down into two parts, the operating system and the User Interface Toolbox.

The operating system lies on the lowest level of this software. It performs basic tasks such as the handling of files and memory. Unlike other computers, the user of the Macintosh has very little interaction with the operating system, dealing with it through a program called the Finder, which provides an easy-to-use graphical interface to perform actions such as copying a file or changing the name of a file. A program written with Macintosh Pascal will also have very little direct contact with the operating system, relying on Pascal to indirectly call the operating system to do things such as open and close files.

The next level up is the User Interface Toolbox, the set of routines that provide a way for constructing programs that conform to the standards established by Apple for the look and feel of a program. These standards are undoubtedly familiar to you already and include the use of pulldown menus, windows, text editing, controls, and dialog boxes. The Toolbox is divided into a set of managers, each one performing one of the standard functions. There is a Window Manager, a Menu Manager, and so on (Figure 1.1).

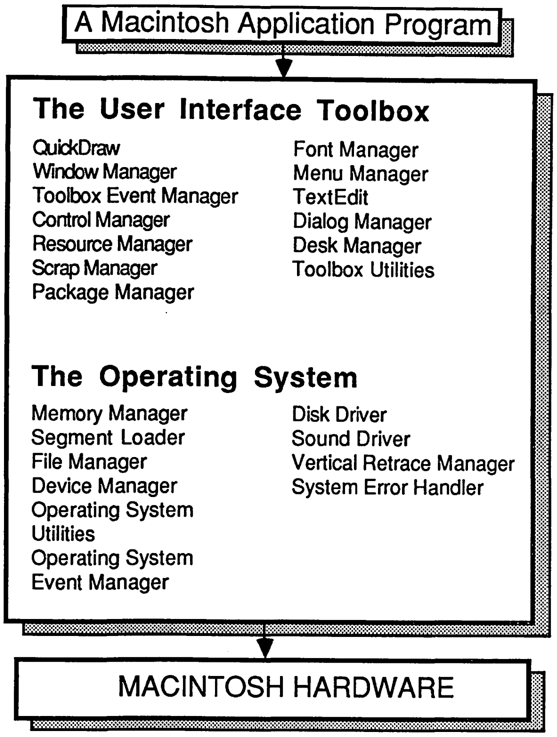


Figure 1.1 The Macintosh programming environment

The complete documentation for the Toolbox is found in *Inside Macintosh,* a publication written by the Macintosh team at Apple Computer. This book has existed in a number of formats since the introduction of the computer, including a three-binder edition, a phonebook-like edition, and a version from a major publisher. Although invaluable to the Macintosh programmer, *Inside Macintosh* is not the last word in clarity or simplicity. It has been accurately described as 25 chapters, each one assuming that you have already read the other 24. For more information on obtaining a copy of *Inside Macintosh,* contact Apple Computer.

An important part of the Toolbox is the QuickDraw graphics package. QuickDraw is responsible for drawing all graphics on the screen, including text. Built into QuickDraw is the ability to draw a variety of graphical shapes and objects, to manipulate these objects, and to draw text in a variety of shapes, sizes, and typefaces. QuickDraw itself is called by many of the Toobox routines to do graphical operations. Direct calls to QuickDraw can also be performed. QuickDraw is covered extensively in two chapters of this book.

Macintosh Pascal

The Toolbox was designed to sit between an application program and the Macintosh. Macintosh Pascal adds another level to this pile since the application program is the language interpreter executing a high-level language program. This solves some problems and causes several others. The major advantage of this setup is that some of the programmer's work in writing programs for the Macintosh is already done. This is significant because even the seemingly simple task of creating a new window and displaying it on the screen is complex enough to befuddle even a sophisticated programmer. It requires extensive knowledge of the Window Manager, QuickDraw, and the Event Manager. When programming in Macintosh Pascal, this knowledge is not required since the interpreter maintains windows for text and graphics ouptut. (However, if you really desire to do this, it is covered in Chapter 6).

The major question presented by this system is how does this system deal with the Toolbox, and how can a programmer access the Toolbox routines. This is handled by Macintosh Pascal in a variety of direct and indirect ways. Essentially, all the Toolbox and operating system routines can be used by a Macintosh Pascal program in one way or another. Direct access is provided to all of QuickDraw and a portion of the rest of the Toolbox. Direct access means that calls to these routines can be used in a program as though they were part of standard Pascal. This extends not only to procedure and functions, but also to special data types that the routines operate on. The directly supported data types can be used in a program exactly as if they were an Integer or Boolean. Pascal-type checking is also performed on all calls to routines that are directly supported.

Direct access to the ROM routines is not always desirable for an interpreted system. Some actions interfere with the interactions of the interpreter itself and disrupt it. This is why only indirect access is provided to the remainder of the Toolbox. Indirect access allows calls to Toolbox routines to pass straight through Macintosh Pascal to be acted on by the computer. Toolbox features such as windows and menus can be harnessed by your Macintosh Pascal programs to create full-featured Macintosh applications. Chapter 6 fully documents and details the use of menus, windows, controls and text editing. Caution should be used, however, since no Pascal-type checking is done, and it is very easy to crash both your program and Macintosh Pascal.

The major disadvantages of an interpreted system is a slower execution speed than a compiled program and limited ways to present a program to users without having to train them to use the interpreter. The speed problem is not as significant on the Macintosh as on other systems due to the fast clock speed of the M68000 microprocessor and the speed that QuickDraw draws graphics. The second problem—not being able to produce a "stand-alone" program that separates the user from the interpreter—has been solved in Version 2 of Macintosh Pascal. This version includes an application shell designed to execute programs without entering the interpreter environment. The additional features of Macintosh Pascal Version 2 are discussed in Appendix A.

In summary, Macintosh Pascal provides a powerful programming system to develop and debug programs as well as significant access to the many advanced features of the Macintosh. The remainder of this book will prepare you to tackle advanced programming problems in Macintosh Pascal by covering sophisticated Pascal structures and techniques, such as file handling, and the use of QuickDraw and Toolbox routines.

Memory

One place to start our exploration of Macintosh Pascal's advanced features is the Macintosh's memory. Even people who have never used a Macintosh probably know from advertising that the Macintosh comes in 128K, 512K, and 1 megabyte versions (plus home-brewed super-Macs with up to 8 megabytes of memory). These numbers refer to the amount of Random Access Memory (RAM) held in the computer and exclude the 64K ROM (128K ROM in the Macintosh Plus).

No matter what size the memory, it is not all available for use by a program. This is because a computer's memory is used for several different purposes simultaneously. Among these are:

- The Operating System—The operating system uses the first 2K of memory to hold the information it needs.
- The Toolbox—The Toolbox keeps its system globals in memory reserved for its own use.
- The Screen—The largest demand for memory comes from the bit-mapped screen which requires 20K to store the image of what is displayed.
- The Sound Buffer—The Mac's sophisticated four-part sound capability requires memory space to store patterns to be translated into sound.

The allocation of memory for purposes such as these is part of a computer's system architecture and is one of the first parts of a systems design. The following memory maps (Figure 1.2) detail the memory layout in both the 128K and 512K Macintosh.

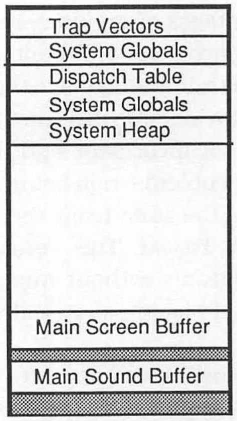


Figure 1.2 Macintosh memory map

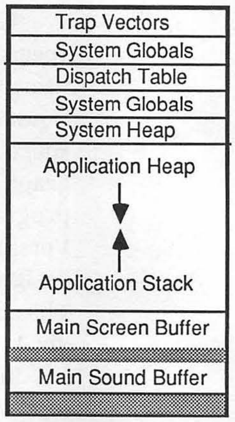


Figure 1.3 Memory, stack, and heap

The big gap in the center of the maps is the space left for application programs. This area is divided into three sections called the Application Global Space, the Stack, and the Heap.

The Application Global Space is allocated when a program starts and holds the global variables of a program; that is, those that are statically declared at the start of a program. The machine Stack is a dynamic data structure that grows and shrinks during the execution of a program whenever temporary storage is needed. For example, the Stack would hold value parameters being passed to a procedure. The Stack grows from the larger memory addresses to the smaller (from high memory to low). The Heap (Figure 1.3) is also a dynamic structure that grows toward the Stack from low memory to high. It is used to hold both the actual program code that is executing and the larger dynamic data structures created by the Toolbox and QuickDraw.

It is important to remember that this is the layout of memory prior to the loading of Macintosh Pascal, which itself is an application program. Macintosh Pascal exists in two separate code sections to preserve Heap space. One section is for the interpreter and one is for the user interface, with either of them residing in the Heap at any time. The area left in memory is what is left for your Macintosh Pascal program that will be interpreted. When Macintosh Pascal is running, it creates a second stack that starts at the end of the Heap and grows toward the Stack. This second stack, called the General Stack, (Figure 1.4) is used by Macintosh Pascal to hold temporary locations needed by a program while it is being interpreted. The Machine Stack holds information for the interpreter itself. The Heap is shared by both Macintosh Pascal and the Macintosh Pascal program.

Unlike the Stack, space on the Heap must be explicitly allocated by a program event in a high-level language. This is a manifestation of the operating system and the fact that, unlike the Stack, the Heap is not supported by hardware.

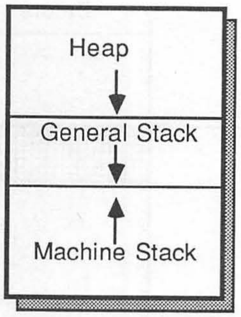


Figure 1.4 The stacks

Information in the Heap is held in blocks of a varying number of bytes that are placed in the Heap on a first-fit basis. This means that the first portion of the Heap with the required number of bytes is allocated even though a more perfect fit could also have been made. The movement of information in and out of the Heap can sometimes be rapid, occurring whenever dynamic structures such as windows or desk accessories are used. Because of this revolving door, the Heap can very quickly become fragmented, broken into scattered blocks too small to be used, leaving requests for large blocks stifled. To keep large blocks of free memory free, the operating system often compacts the contents of the Heap, moving all the unused blocks together into one large area (Figure 1.5).

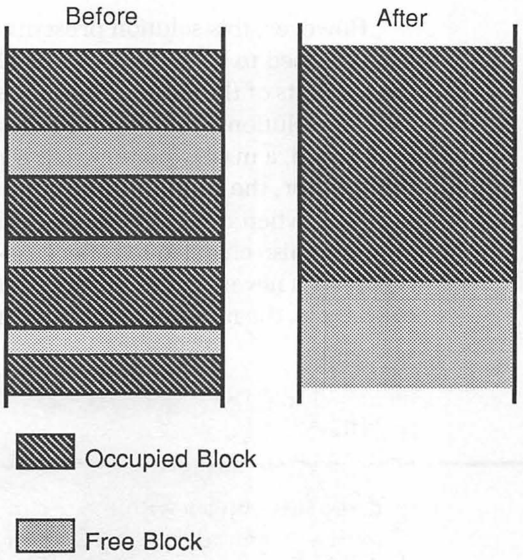


Figure 1.5 Before and after compact

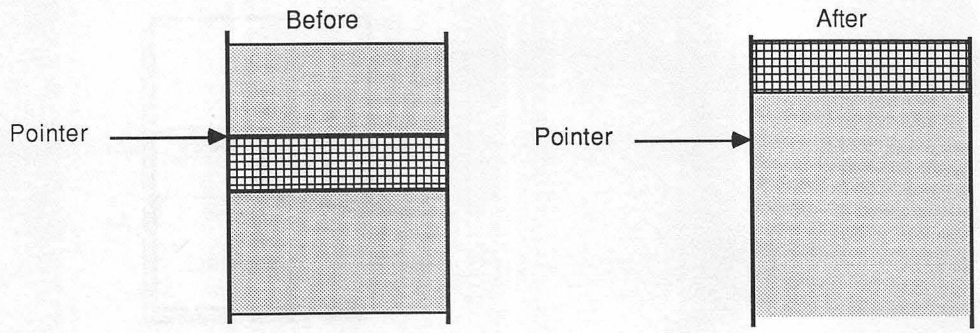


Figure 1.6 Lost pointer

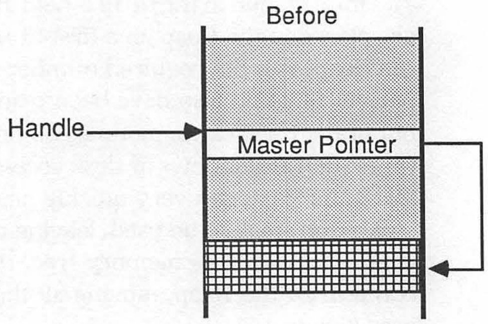


Figure 1.7 Handle

However, this solution presents a problem all it own. If a pointer (Figure 1.6) was used to keep track of a block in the Heap, it will be inaccurate when the contents of the Heap are relocated.

The solution is the use of handles (pointers to pointers). Whenever a block is allocated, a master pointer to it is placed in the Heap, and then a pointer to that pointer, the handle, is also created (Figure 1.7).

Now when a block is moved during Heap compaction, the value of the pointer is also changed to reflect the move. For all this to work, the pointer in the Heap is never moved during compaction. Now, since all access is done via the handle, the movement inside of memory is transparent to the program.

Note

If you are familar with the use of pointers, you will find that Chapter 2 reviews some advanced Pascal structures.

The clever programmers should now be asking themselves how this feature can be useful. The answer is that the amount of space left over for both a Macintosh Pascal program code and data is small and must be maximized. Allocated Heap space on a per need basis rather than as global variables in the Var section of a program will help manage the precious memory space.

Macintosh Pascal contains a series of Toolbox routines for dynamic allocation of Heap space. The first of these is the NewHandle function.

```
function NewHandle(Size: Integer) : Handle;
```

Note

Whenever a Toolbox routine is introduced, it's complete procedure or function heading will be shown to indicate all the parameters and their data types. This, by the way, is the way routines are documented in *Inside Macintosh*.

The NewHandle function allocates a block Size bytes large in the Heap and returns a handle to it. The operating system automatically takes care of creating the master pointer and linking the block, pointer, and handle together.

Since Handle is a data type not already declared in Macintosh Pascal, it must be declared in your program's type section. For example, if you want to create a handle to an Integer:

```
type
IntPtr = ^Integer;
IntHandle = ^IntPtr
```

First declared is a pointer to an Integer, and then the handle is declared as a pointer to the pointer. A variable of the handle type must also be declared.

```
var
Int : IntHandle;
```

Int is then the variable that the newly created handle is assigned to.

```
Int := NewHandle(2);
```

No Pascal-type checking is done for assignment of the result of NewHandle so the programmer must be very careful. Note that the size of an Integer is 2 bytes. It is unnecessary to know this beforehand since the SizeOf function can be employed to find the number of bytes occupied by either a data type or variable.

```
function SizeOf (ID : AnyType) : Integer;
```

The SizeOf function returns the number of bytes needed to store a data type or a variable. For instance :

```
SizeOf(Integer);
```

will return the number of bytes needed to store an Integer (which we know to be 2) and

```
SizeOf(Int)
```

will return the number of bytes occupied by this particular variable, which would be the same for any variable of that data type.

So, it is safer to always use SizeOf when calling NewHandle.

```
Int := NewHandle(SizeOf(Integer));
```

Once declared, the integer that has Int as its handle can be referenced as

```
Int^^
```

The two arrows indicate two levels of indirection. Notice that the Integer created has no variable name, which is true of all dynamically created variables (variables not declared in the Var section of a program).

While the dynamic allocation of a single Integer makes little sense, if a program required an array of 1,000 Integers to perform an operation done once and then won't need it again, it would make little sense to have all that memory space reserved.

The following set of declarations can be used to create a dynamic array.

```
type
  arrayType = array[1..1000] of Integer;
  arrayPtr = ^arrayType;
  arrayHdl = ^ arrayPtr;
var
AR : ArrayHdl
AR := NewHandle(SizeOf(AR));
```

First, an array type was declared and then a handle to it was developed. Finally, a variable of the handle type was declared in the Var section of the program. An element of the array which had AR as a handle to it (can we say AR handles to the array?) can be accessed as:

```
AR^^[1] := 99;
```

This statement would assign 99 to the first element of the array.

When the dynamic array is no longer needed, it can be disposed of and the memory it occupied reclaimed with the DisposeHandle procedure.

```
procedure DisposeHandle (Hdl : Handle);
```

The DisposeHandle procedure destroys the data structure indirectly pointed to by Hdl and reclaims its Heap space. The handle must have been created with NewHandle.

The following are three other memory management routines of lesser importance than NewHandle and DisposeHandle.

```
function GetHandleSize(Hdl : Handle) : Integer;
```

The GetHandleSize function returns the size in bytes of the data structure indirectly pointed to by the handle Hdl. To use our example dynamic array AR,

```
GetHandleSize(AR)
```

would return 2,000 since that is the size in bytes of the 1,000-element array of integers.

```
procedure SetHandleSize (Hdl : Handle; NewSize : Integer);
```

The SetHandleSize procedure changes the size of the block of memory indirectly pointed to by Hdl by NewSize number of bytes. This can be used to alter the size of a dynamic data structure. Since enough memory might be available, a call to GetHandleSize must be made to confirm the effect of SetHandleSize.

```
procedure BlockMove(FromPtr, ToPtr : Pointer; NumBytes : Integer);
```

The BlockMove procedure copies NumBytes number of bytes from the data structure pointed to by FromPtr to the data structure pointed to by ToPtr. Note that the parameters are pointers and not handles. To copy the memory pointed to by a handle, one level of indirection must be added to the parameter. For instance, if we had two dynamic arrays with AR1 and AR2 as handles, a copy of AR1 could be made with:

```
BlockMove (AR1^, AR2^, GetHandleSize(AR1));
```

Since it takes execution time to follow a handle all the way to its underlying data structure, some programmers employ a short cut by using a copy of the pointer that the pointer handle points to.

```
var
  ARPtr : ArrayPtr;
  AR : ArrayHdl;
```

By assigning AR^ to ARPtr, one level of indirection can be removed. The array can then be accessed with:

```
ARPtr^[1]
```

The speed increase with this technique is approximately 10 percent. However, danger lurks behind this technique. If memory is compacted at any time, the handle and the master pointer will be updated, but the copy of the master pointer held in ARPtr will not be. This leaves the copy pointing to an undefined and undesigned memory location. This problem is known as a *dangling pointer*.

This situation is normally prevented by locking the block of memory into its position in the Heap so it is not moved during compaction. Once it is no longer needed, it is then unlocked. Unfortunately, Macintosh Pascal does not contain the necessary Toolbox procedures to do this. They have been added in Macintosh Pascal Version 2.0 and are documented in Appendix A.

CHAPTER

2

Advanced Pascal Structures

This chapter covers three advanced Pascal structures used extensively by the Macintosh User Interface and QuickDraw—namely, records, sets, and pointers. While you may be familiar with these structures from an introductory Macintosh Pascal book, there is a good chance that they were not covered in sufficient detail to allow you full flexibility with them.

Records are a structured Pascal data type that allows the grouping of information of different data types together in the same variable. They are used with files and as a way of exchanging information between the operating system and a program. Pointers are variables that hold the address of another variable rather than a value. They are used to facilitate dynamic memory allocation both alone and in twos to form handles. Sets are the Pascal implementation of the mathematical set concept. They prove quite useful in programming tasks such as input verification.

It is important that you completely understand these programming constructs before you start exploring the inner workings of the Macintosh. Your full concentration can then be placed on the material presented and you will not be distracted by the programming techniques used to demonstrate them. If you are already familiar with these subjects or think that you might be, skim through the chapter so that you can refer back to it when necessary.

Records

The Pascal record data type provides a structure for storing information of different data types together. In a record, several different variables of any data type are stored with the same variable name. This is often compared to arrays but the similarities are slim. In an array, many different values of the same data type are stored under the array name and accessed with the use of a subscript. The number of values in an array can range into the thousands and still be easily managed with the help of a For loop. In a record, several values of different data types are stored under a record name, but no subscript is used. Access to each element in the record is done by using its complete name. This limits the number of components in a record to a few dozen at most. This is not to say, however, that records are of limited value. Their ability to coordinate data of different types is one of the most important facilities of the Pascal language.

Records are a structured type and are usually created by first declaring a record data type and then declaring a variable of that type.

```
type
 DateRec = record
  Month : 1..12;
  Day : 1..30
  Year : Integer;
  DayOfWeek : string
 end;
```

The record type DateRec contains four components or fields: Month, Day, Year, and DayOfWeek. Three of the fields contain three subranges of integers and one string.

The general form of a record-type definition is:

```
RecordName = record
 Field1 : DataType;
      .
      .
 Field2 : DataType;
end;
```

Once a record type is defined, variables of that type can be declared in the Var section of the program.

```
var
DateInfo : DateRec;
```

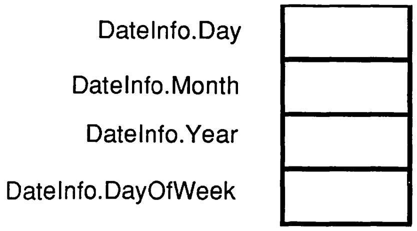


Figure 2.1 The record DateInfo

The variable section declares a record of type DateRec, called DateInfo. The record DateInfo (Figure 2.1) contains four fields based on the definition of the type. Each field in the record is referred to by its record name followed by its field name and they are separated by a period.

```
RecordName.FieldName
```

A field can be used just like any other variable. For instance, in an assignment or Writeln statement.

```
DateInfo.Year := 1986;
Writeln(DateInfo.Day);
```

More than one record of the same data type may be declared.

```
var
Day1, Day2 : DateRec;
```

If this is the case, the same fields exist in both records but the field names are different because the record names are different (Figure 2.2).

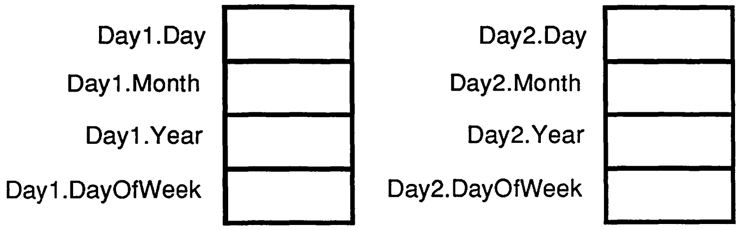


Figure 2.2 The records Day1 and Day2

When two records are of the same record type, the entire contents of one record can be assigned to the other in just one statement.

```
Day1: =Day2
```

This statement is equivalent to:

```
Day1.Day : = Day2.Day;
Day1.Month : = Day2.Month;
Day1.Year : = Day2.Year;
Day1.DayOfWeek : = Day2.DayOfWeek;
```

The With Statement

Since the full name of a field can be tedious to write, the With statement was included in Pascal to provide shorthand forms of field names. The With statement automatically adds the record name in front of the field name.

```
with Day1 do
 Year : = 1986;
```

In the preceding statement, the record name Day1 is automatically used with the assignment statement. This With statement is the equivalent of:

```
Day1.Year : = 1986;
```

The general form of the With statement is:

```
with RecordName do
 Statement;
```

Like any Pascal structure, the With statement can work with either a single statement or a compound statement delimited by a Begin and End. The With statement operates by checking the variables contained inside it to see if any are the names of fields of the specified record. If they are, the field name is preceded by record name. Assuming the following record declaration:

```
var
BirthDay : DateRec;
I : Integer;
```

The With statement:

```
with BirthDay do
 Mcnth : = I;
```

will assume that Month refers to the field in the record BirthDay and will affix the record name to it. Thus, the following statement is equivalent:

```
BirthDay.Month : = I;
```

If a field name is the same as an independent variable's name, some programming confusion can arise.

```
var
 PayDay : DateRec ;
 Month : Integer;
```

We now have a field in the record Month and a stand-alone variable both with the name Month. This is normally no problem, but a problem is created by the following With statement.

```
with PayDay do
 Month := Month;
```

In the statement above, it is not clear if the Month referred to is the field in the record or the stand-alone variable, and if it does refer to one, which one? Pascal handles this situation by following the simple rule that any variable used in a With statement that happens to be a field in the record specified is assumed to be that field. Applying the rule to our example makes the following statement equivalent.

```
PayDay.Month := PayDay.Month;
```

It is best to stay away from situations like these by avoiding naming a variable the same as a field.

Records are most commonly used with files and Chapter 3 carefully examines this. Another common use of records, especially on the Macintosh, is as a way to exchange information with the operating system, Toolbox, and QuickDraw. In order to implement this exchange, several record types have been predefined in Macintosh Pascal. To use a record of this type, no type declaration is needed. An example of this is the record type used to represent the time and date from the Macintosh's built-in clock. DateTimeRec is defined as:

```
type
DateTimeRec = record
 Year,
 Month,
 Day,
 Minute,
 Second,
 DayOfWeek : Integer
end;
```

Where each field contains:

- Year — the number of years since 0 A.D.
- Month — the number of the month with January as 1 and December as 12
- Day — the day of the month (1, 2, etc.)
- Hour — the number of hours since midnight
- Minute — hopefully, this is self-explanatory
- DayOfWeek — a number from 1 to 7 representing Sunday to Saturday.

If a record of type DateTimeRec is declared in a program,

```
var
Today : DateTimeRec;
```

information can be passed to it from the clock with the help of the GetTime procedure.

```
procedure GetTime (var Date : DateTimeRec);
```

The GetTime procedure passed to the parameter the current date and time from the clock.

The following program uses a DateTimeRec with the GetTime procedure to retrieve and display the clock information.

```
program WhatTime;
 var
  Today : DateTimeRec;
 begin
  GetTime(Today);
    with Today do
    begin
      Writeln(Hour);
      Writeln(Minute);
      Writeln(Second)
 end {with}
end
```

Variant Records

Pascal provides a mechanism for alternative storage schemes in the same record structure. Called *variant records,* they allow the number and data type of fields to change dynamically depending on what is stored in the record.

A variant record has two parts. The first is the fixed part, which is the same as a nonvariant record. The second part, known as the variant, changes depending on the value stored in the field designated as the tag field. A variant record allows us to store information in one record type that is similar but may otherwise require two different record structures. Let's take as an example an employment record that holds data for both management and regular employees. Both classes of employees require some information in common. This information forms the fixed part of the record.

```
type
 EmployeeType = (Management, Worker);
 EmployeeRec = record
  Name : string[30];
  Address : string[50];
  PhoneNo : LongInt;
```

However, for the two different types of employees, some different information is required. For management, their direct supervisor, yearly salary, and office number is needed but for workers, their hourly wage and the name of their foreman is called for. Both of these requirements can be met with the record's variant part. The variant part starts with the tag field, which lets the record know which type of data will be stored, and then a Case statement to select which fields are to be included.

```
case Position : EmployeeType of
 Management :
    (Supervisor : string[30];
    Salary : Real;
    OfficeNo : Integer);
 Employee :
    (Foreman : string[30];
    Wage : Integer)
end; {whole record}
```

The fourth field in the record, called position, is the tag field and is used in the Case statement. Syntactically, the variant record part uses parentheses instead of Begin and End to indicate a compound statement as part of the Case. Only one End is used to signify both the end of the record and the Case statement. Depending on the value of Position, the remaining fields in the record will change. The remarkable thing about this is that two consecutive records in a file can contain two different types of data. If the value of Position is Management, the record will contain the following fields.

```
Name : string[30];
Address : string[50];
PhoneNo : LongInt;
Supervisor : string[30];
Salary : Real;
OfficeNo : Integer;
```

If the value of Position is Worker, the record will contain the following fields.

```
Name : string[30];
Address : string[50];
PhoneNo : LongInt;
Foreman : string[30];
Wage : Integer
```

Notice that not only are the field names in the variant part different but so are the number of fields, the data types, and the storage requirements. A record can contain only one variant part but there may be nested variants. If the value of a tag field is changed from one value to another, the variant fields become undefined. Concerning storage space for a variant record, all the variant parts share the same memory event though one variant part is active at a time. The amount of memory or file space needed is the space needed to hold the largest combinations of the fixed fields and the variant fields; if the variant part is changed, there is always room to store the new fields.

Variant records are of interest to us since many Toolbox and QuickDraw data types are defined with variants. This gives the programmer a lot of flexibility when working with these data types. The tag field in a variant record is actually optional and is not used in most of the Macintosh data types. When no tag field is used, all the variant fields are always accessible. This is the way that the QuickDraw data type Point is defined.

```
type
 VHSelect = (V, H);
 Point = record case integer of
  0 :
   (V : Integer;
   H : Integer);
  1 :
   (VH : array [VHSelect] of Integer)
 end;
```

The definition of Point uses no tag field and only a data type as the case selector. This structure allows a programmer to access the fields of a record of type Point in either of two ways. If a record called ThePoint is declared, the fields can be called either:

```
ThePoint.V
ThePoint.H
```

or

```
ThePoint.VH[V]
ThePoint.VH[H]
```

whichever is more convenient at the time.

Many programmers use variant records without a tag field to circumvent Pascal's strong type-checking. They use the fact that variants occupy the same space in memory to redefine a value of one type into another type. Consider the following declaration:

```
var
 Switch : record case boolean of
  False :
    (Int : Integer);
  True :
    (Ch : Char)
 end;
I : Integer;
```

The above record structure setup will allow us to convert a character into its ASCII value without the help of the ORD function.

```
Switch.Ch := 'A';
I := Switch.Int;
```

Since Switch.Ch and Switch.Int occupy the same memory location, the ASCII code stored as the value of the character can be assigned to the integer I. Note that the statement

```
I := Switch.Ch;
```

could not be done since the data types are not the same. This technique is very powerful but is not recommended unless the programmer is very familiar with the way Pascal and the Macintosh represent data. Another use of this technique might be a situation in which different data is stored in the record, but the programmer can tell from the context which value is needed. An example may be a file in which all the even records contain an integer and all the odd records contain a character.

Sets

Sets are a data structure unique to Pascal. The set data type is Pascal's implementation of the mathematical set concept. A Set can have up to 255 different members, which are all of the same ordinal data type. An ordinal data type is one in which all the values can be enumerated. This includes all of the scalar data types except Real.

A set type is declared in the program's Type section.

```
type
LetterSet = set of Char;
```

The declaration defines a set data type whose membership consists of characters. The general form of a set type is:

```
type
 SetType = set of ordinal type;
```

All of the following declarations are legal.

```
NumSet = set of Integer;
CharSet = set of 'A' .. 'Z';

ColorType = (black, red, blue);
ColorSet = set of ColorType;
```

The set-type declaration does not create any set; it just establishes a data type. A set is created with a variable declaration.

```
var
InputSet : NumSet;
```

Here a set of Integers called InputSet is created. At this point, InputSet is empty since it contains no values members. The members of a set must be of the underlying data type and are placed into a set with an assignment statement. Remember, a set may contain up to 255 members, each one different.

```
InputSet := [1, 2, 3, 4, 5];
```

Members of a set are delineated by a left and right bracket and separated by commas. It can now be said that InputSet has five members, the integers 1 through 5. Since five members are consecutive values of an ordinal type, a subrange could have been used in the assignment:

```
InputSet := [1.. 5];
```

Set Operations

Pascal contains a complete repertoire of operators that act on sets. The first of these is the Set Membership function. Invoked with the keyword In, the Membership function tests to see if a value is a member of a given set, returning true if it is and false otherwise.

```
if Num in InputSet then
 Writeln('Set Member')
else
 Writeln('Not a Set Member');
```

The In function is very handy for doing many tasks, including input verification: that is, the testing of a value entered to see if it falls within a certain range. For instance, suppose we want to test an entered value to see if it falls within 1 to 10. First, declare a set having all the acceptable values as members. Then, test the value entered to see if it is a member of that set. This is most often done in a Repeat loop so that the input sequence can be repeated should the value fall out of range.

```
InputSet := [1 .. 10];
repeat
 {Display choices}
 Readln(Num);
 if not (Num in InputSet) then
  {Error message}
until Num in InputSet;
```

In this situation an If statement could also be easily used, but if the possible values are not consecutive, sets are much easier and more efficient way to go.

Set Union

The SetUnion operator is used to combine two sets. Remember, a set can have only one occurrence of any member. The plus sign (+) is used as the set union operator.

```
NewSet := [1, 2] + [2, 3, 4];
```

The members of NewSet are [1, 2, 3, 4]

Set Intersection

The set intersection operation is used to find the common members of two sets. The multiplication sign (*) is the set intersection operator.

```
NewSet := [1, 2] * [2, 3, 4];
```

The members of NewSet are [2].

Set Difference

The set difference operator is used to find all the members in one set that are not in another. The minus sign (−) is the set difference operator.

```
NewSet := [1, 2, 3] - [2, 3];
```

The members of NewSet are [1].

Set Comparisons

The standard relational operators work with sets although their meanings change some. The following table details the result of comparing two sets.

Table 2.1

EXPRESSION	RETURNS TRUE IF
Set 1 = Set 2	Set 1 and Set 2 are identical
Set 1 < > Set 2	Set 1 and Set 2 are not identical
Set 1 < = Set 2	Set 1 is a subset of Set 2
Set 1 < Set 2	Set 1 is a strict subset of Set 2
Set 1 > = Set 2	Set 2 is a subset of Set 1
Set 2 < Set 1	Set 2 is a strict subset of Set 1

Another demonstration of the use of sets is a procedure that converts the characters in a string from lower case to upper case. This is done by declaring a set consisting of the lower-case characters, testing the characters in the string to see if they are members, and converting them if they are. Lower-case characters can be converted to upper-case characters by changing their ASCII value. The upper-case characters start with 65 as their ASCII code and the lower case with 97, a difference of 32.

```
procedure LowerToUpper(S : string);
  var
   UpperSet : set of Char;
   K : Integer;
   begin
    UpperSet := ['A' .. 'Z'];
    for K := 1 to Len(S) do
     if S[K] in UpperSet then
       S[K] := Char(Ord(S[K] - 32))
   end;
```

Pointers

Pointers may be the most sophisticated feature of Pascal. In Chapter 1, pointers and handles were introduced. They are covered in more detail here. A pointer is a variable that, instead of holding a value, holds a reference to another variable (the address of another variable). While this might not seem very important, it is—because the variable that a pointer refers to can be created dynamically during program execution. This is unlike the usual scheme in which variables are declared in the variable section of a program and created by the program prior to execution.

Pointers are declared with the help of a pointer type.

```
type
 NumPointer = ^Integer;
```

The preceding type statement declares NumPointer as a pointer to a location containing an integer. The circumflex '^' (found above the 6 key on the Macintosh keyboard) indicates that NumPointer points to an integer rather than being an integer. Once a pointer type is defined, a pointer variable is declared like any other.

```
var
 P, Q : NumPointer;
```

Here we have declared two variables (pointers for short) which point to integers. The particular variable a pointer references is not originally defined, and a value can't be assigned to the pointer since it does not hold an integer. A new variable is dynamically created and linked to a pointer with the New procedure. The New procedure takes a pointer and then creates a new variable pointed to it.

```
New(P);
```

This call to New creates a new integer variable pointed to by P (Figure 2.3).

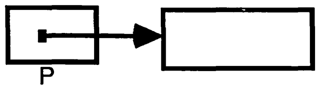


Figure 2.3 The pointer P

The variable pointed to by P has no variable name since it is dynamically created. The only way to reference it is through its pointer P. The location referenced by a pointer is accessed with the address operator symbolized by the up arrow (^). So if P points to an integer, P^ is the integer that it points to. Figure 2.4 shows that a value can be placed in that variable with: P^ := 17;

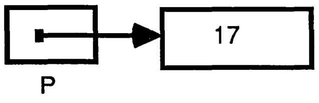


Figure 2.4 The variable P^

We are now dealing with two different types of variables: the pointer variable and a regular integer it points to that doesn't have a regular name. Because they are of different types, the following assignment statement is illegal.

```
P:= P^;
```

This statement attempts to assign an integer to a pointer producing a conflict in data types. Other pointers, however, can be assigned the value of a pointer.

```
Q:= P;
```

This statement makes the pointer Q point to the same variable as P (Figure 2.5).

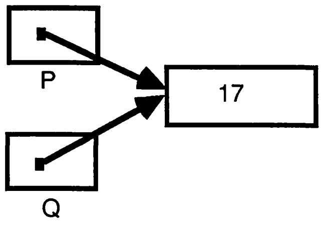


Figure 2.5 The pointers P and O

The variable can now be referenced as either P^ or Q^ (Figure 2.6) so that the following two assignment statements have the same effect.

```
P^:= 99;
Q^:= 99;
```

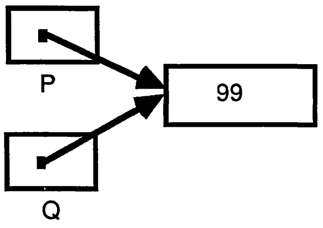


Figure 2.6 The pointers P and Q

If the value of pointer P is changed, Q still points to the same location. The value of a pointer can be changed by assigning it to a new location, passing it as a parameter to the New statement, or assigning it the value Nil. Nil is a Pascal reserved word used to represent a pointer whose value is undefined. The advantage of assigning Nil to a pointer rather than just leaving it undefined is that Nil can be checked for in an If statement.

```
if P = nil then
  Write('Undefined pointer');
```

Pointers in general cannot be printed and can only be compared to the value Nil or the value of another pointer.

The real power of pointers is harnessed when using records containing a pointer. Consider the following type and variable declarations.

```
type
 LinkRec = record
  Data : Integer;
  Link : ^LinkRec
 end;
var
 First, P, Q : ^LinkRec;
```

A record type called LinkRec is declared. One of the fields of LinkRec, Link, is a pointer to the type LinkRec. That is, that field may hold a pointer to another record of type LinkRec. Even though this definition seems recursive, it is perfectly legal. The variables declared are three pointers to the type LinkRec which initially point to nothing. With this structure it is possible to create an unlimited (except by available memory) chain of records all linked together. A structure like this is called a *link list* and is very popular since only the number of records needed can be generated.

The first step is to produce the first record (Figure 2.7).

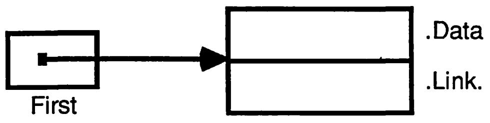


Figure 2.7 The record pointed to by First is created

```
New(First);
```

The variable pointed by First is known as First^. Because First^ is a record, there are two fields: First^.Data and First^.Link. A value can be assigned to the Data field of the record with:

```
First^.Data := 1;
```

A second record can be generated and then linked to the existing record by creating a new record pointed to by Q (Figure 2.8).

```
New(Q);
```

To facilitate the linking of many records, P should also be made to point to the first record.

```
P := First;
```

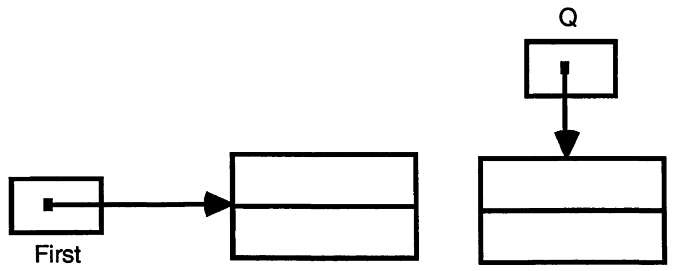


Figure 2.8 The new record pointed to by Q

Because this is an assignment of a pointer, no up arrow is used. Since two pointers point to the the first record, the Link field can be identified as either First^.Link or P^.Link. The newly created record pointed to by Q can be linked to the first by making the Link field of the first record point to the same thing as Q (Figure 2.9).

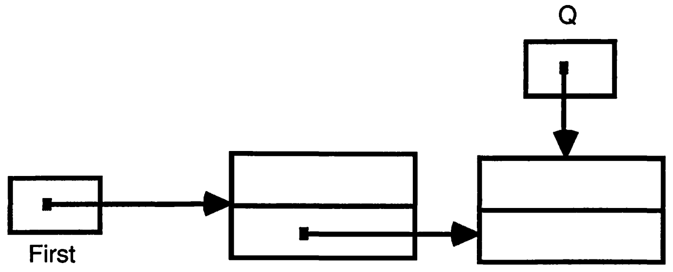


Figure 2.9 Linking the first two records

```
P^.Link := Q^; {Make the first record point to the second}
```

If P is now made to point to the last record (where Q points), the entire generation and linking of new records can be placed in a loop.

```
P := Q;
for I := 1 to 3 do
begin
  New(Q);
  P^.Link := Q;
  P := Q
end;
```

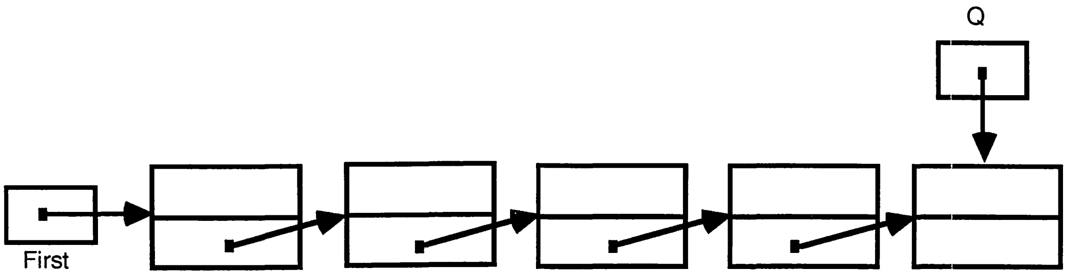


Figure 2.10 The linked list

After the last record is placed on the list (Figure 2.10), its Link field should be set to Nil in order to be able to detect it when later parsing the list.

```
Q^.Link := nil;
```

Here is the entire list-generation code together.

```
program LinkUp;
type
 LinkRec = record
 Data : Integer;
 Link : ^LinkRec
end;
var
 First, P, Q : ^LinkRec;
 New(First);
 P := First;
 P^.Link := Q^;
 P := Q;
 for I := 1 to 3 do
 begin
  New(Q);
  P^.Link := Q;
  P := Q
 end;
 Q^.Link := nil;
```

Once the list is created it can be traversed easily. For instance, to sum the Data fields in all of the records, assign one of the pointers to the first record, get the Data value, and then use the Link field to move the pointer up to the next record. The action stops when the Nil Link field is encountered, signaling the end of the list.

```
Sum := 0;
P := First;
repeat
 Sum := Sum + P^.Data;
 P := P^.Link
until P = nil;
```

Pointers are used extensively throughout Toolbox and QuickDraw since many of the data structures utilized by them are dynamic in nature and are often moved around in memory by the operating system.

CHAPTER

3

Files and File Programming

If programs lacked the ability to store data for later retrieval, all the microcomputer applications we are familiar with today such as database managers and word processors, would be impossible. Data is stored long term in files held on external media such as a floppy disk. That data exists independently of the program that created it. This data can be retrieved at a later time by the program that stored it or even by a different program. This chapter discusses the entire spectrum of files available with Macintosh Pascal, including sequential files, the more powerful random files, and text files. Since files would be of little use without the programming techniques needed to exploit them the information is presented from the point of view of file-handling techniques.

File Concepts

Files are traditionally defined as a collection of records. As with most glossary-type definitions, this does little to indicate how files operate and what purpose they serve. To understand files more precisely it would be useful to first review arrays. An array is a data structure that holds a predefined number of values of the same data type in memory. An array is declared in the **var** section of a program.

```
List : array[1..10] of Integer;
```

The preceding array declaration declares an array named List which holds ten integers (Figure 3.1).

Since arrays are stored in memory, array elements can be manipulated just like a variable; for instance, in an assignment statement such as:

```
K := List[2];
```

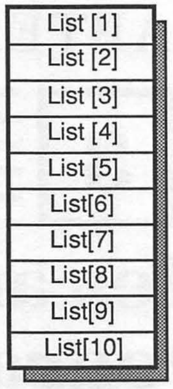


Figure 3.1 The array list

Files have similar characteristics to arrays but also some major differences. A file like an array, holds a series of values of the same data type, but these elements are not held in memory like an array's elements. Instead, they are held on a secondary storage device which, in the case of the Macintosh is usually a 3.5-inch floppy disk. The individual components of a file are known as *records*. This introduces a second usage of the term "record" (the record data type is the other). The data type of the components of a file is very often declared as a record data type but not necessarily so. Unlike an array, the number of elements in a file is not fixed but can vary. The records in files are numbered with the first record numbered zero.

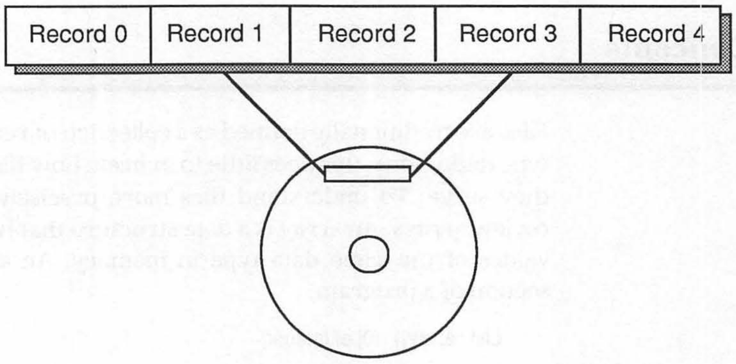


Figure 3.2 Records in a file

Since the data in a file is physically held outside the computer, it is a complicated and relatively slow process to store or retrieve it. This is due to the mechanical nature of the disk drive itself. Accessing data in a file requires the assistance of the Macintosh's operating system to act as a go-between for memory and the disk drive. The operating system contains a series of routines to do the necessary file housekeeping such as starting the disk drive spinning, checking to see that it is spinning at the proper speed, locating the proper part of the disk surface, moving the disk drive head, and so on. Due to their complexity, computer manufacturers include this series of operations in a computer's operating system to alleviate the need for programmers to write them. On the Macintosh, these routines are stored in the computer's ROM along with the User Interface Toolbox.

Macintosh Pascal supports two types of files, sequential and random. The difference lies in how the records in the file are accessed. In a sequential file, the records must be accessed in sequential order, starting with the first record in the file. For example, in order to add a record at the end of the file, the other records in the file must first be passed over. Sequential files are of limited use for this reason and are a leftover from the days before disk drives when the only secondary storage devices available were sequential access devices such as magnetic tapes. The second file type is random. These files allow sequential access but also provide a mechanism to access any record in the file directly, thus providing greater flexibility and speed.

Several operations are necessary to access files. Before a file can be used it must first be opened. Opening a file performs several actions, but most importantly, it establishes the channel of communications between the program and the storage device. Different procedures are used to open sequential and random files. When a file is no longer needed, it must then be closed. Closing a file terminates all communication between a program and the storage device with reference to the particular file closed. The action of placing data into a file from a program is known as *writing*. The opposite of writing data is reading it.

The File Data Type

Pascal implements files with the use of the File data type. A file is first declared in the **var** section of a program.

```
var
Numbers : file of Integer;
```

The preceding statement declares a variable of type File called Numbers. Following the keywords **file of** is the data type of the components of the file. This is known as the file's component type. Of course, as you can see the component type of file Numbers is Integer. Notice that there is no mention as to the size of the file. This is because the number of records in a file is free to vary dynamically. This file was declared in the **var** section of a program but a file can also be declared as a type.

```
type
 RealFile = file of Real;
 Boolean file = file of Boolean;
 IntegerFile = file of Integer;
var
 Grades : IntegerFile;
 Answers : Boolean file;
 Temperature : RealFile;
```

The component type of a file is not limited to the basic Pascal scalar data types but can also be a structured type such as a record—and very often is.

```
type
 GradeRec = record
    Name : string[20];
    ExamNumber : Integer;
    Grade : Integer
   end;
 GradeFile = file of GradeRec;
var
 Class1, Class2 : GradeFile;
```

The preceding sequence declares two files, Class1 and Class2, both having the component type GradeRec.

Declaring a file in the **var** section of a program creates an internal data structure but does not invoke any of the operating system routines that open a file on the disk drive and link it to the program. This is accomplished with one of the built-in procedures that open a file.

Using Sequential Files

The first type of files we will explore are sequential files. Since sequential files are not as useful as random files, the main focus of this chapter is random files, but the basic file operations are the same with both file types.

Opening a Sequential File

There are two routines used to open a sequential file, depending on whether write-only or read-only access to the file is desired. Once a sequential file is opened for write-only access, data can be added to the file but not read from it. When a sequential file is opened for read-only access, data can be read from the file but not added to it. Data cannot be read from *and* written to a sequential file without closing the file and then reopening it. This is a major disadvantage of sequential files. The Rewrite procedure is used to open an existing sequential file for write-only access or to create a new sequential file and then open it for write-only access. The Reset procedure is used to open an existing sequential file for read-only access.

Both Rewrite and Reset take the same two parameters: the file variable declared in the program and a string containing the name that file is identified by on the secondary storage device.

```
Rewrite(file variable, 'external name');
Reset(file variable, 'external name');
```

For example, a file of integers was declared:

```
var
 IntFile : file of Integer;
```

A file named Data can be created on the disk drive with:

```
Rewrite(IntFile, 'Data');
```

The file Data can be seen on the desktop and appears as a generic Macintosh file icon (Figure 3.3).

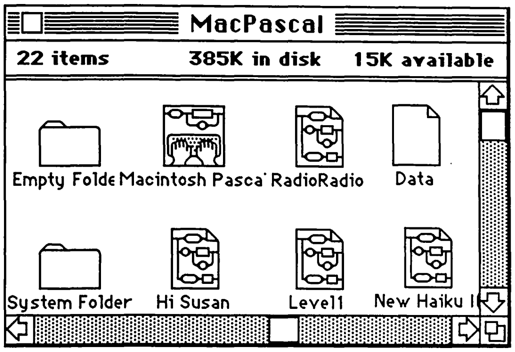


Figure 3.3 The file Data on the Macintosh desktop

Once a file is opened, communication is established between the file variable and the file on the external device. Data can be passed back and forth between the two with the Put and Get procedures.

The Put procedure is used to place information into a sequential file opened for write access. Put operates in conjunction with the file variable. When the file variable is declared, a pointer to that variable is automatically created along with it.

To place information into a file, data is assigned to the file variable pointer, and then the Put procedure is used. For example, assuming our file IntFile has just been created for the first time with Rewrite, then a record can be written to the file by first assigning a value to the file pointer:

```
IntFile^ := 2846;
```

and then calling the Put procedure with the file variable.

```
Put(IntFile)
```

At this point, the value held in the file pointer is placed in the external file associated with the file variable from the Rewrite statement. Specifically, in our example the integer 2846 is placed in the first position of file Data (record zero). The file is then ready to accept a second record.

```
IntFile^ := 777;
Put(IntFile);
```

Figure 3.4 depicts the file as it now stands.

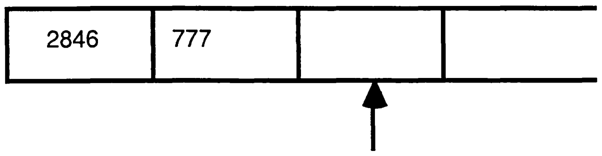


Figure 3.4 The file IntFile

Closing a File

Once a file is no longer needed, it should be closed with the Close procedure. Close terminates the association between the file variable and the external file. All subsequent references to the file variable (except to reopen it) are invalid and will trigger a run-time error. The form of the Close procedure is simple.

```
Close(file variable)
```

Reading Data

Data is read from a sequential file with the Get procedure. Get can be thought of as the inverse of Put.

```
Get(File variable);
```

When a Get is executed the data stored at the current file position is placed into the file pointer. Before data can be read from a file, it must be opened for read access with the Reset procedure. When a Reset is performed it automatically reads into the file pointer the first record in the file, and then it advances the file position to the next record in the file. This means that a Get does not have to be executed to read the value of the first record.

Note

Reset automatically performs a Get on the first record in a file.

We can read and display the integers stored on the disk drive by closing the file and then reopening it with Reset.

```
Close(IntFile);
Reset(IntFile,'Data'); {Performs the first Get}
Writeln(IntFile^);
Get(IntFile); {Get the second record}
Writeln(IntFile^);
```

Notice that only one Get is performed since the Reset reads the data stored in the first record of the file.

In the previous example it was unnecessary to close the file before it was Reset. At any point during operations with a sequential file, the file can be Reset for read operations. When a file that is already open is Reset, only one parameter, the file variable, is needed since the file is already associated with a file on an external device (Figure 3.5). In our example, the Reset is performed with:

```
Reset(IntFile)
```

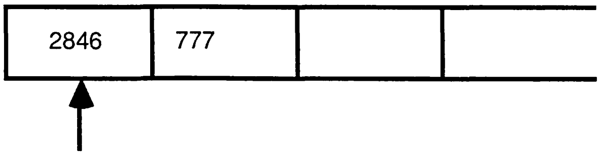


Figure 3.5 IntFile after Reset

Erasing a Sequential File

A sequential file can be removed from an external device with the Rewrite procedure. We have already seen Rewrite used to create a nonexisting sequential file, but when Rewrite is used to associate a file variable with an external file that already exists, the contents of that external file are effectively deleted. The file position is then set to the beginning of the file, and the file is ready to have new records written to it. If Rewrite is called when the file is already open

```
Rewrite(file variable)
```

the file is then rewound with the contents deleted, and the file position is set to the beginning of the file.

File Processing with Sequential Files

The inability to change a sequential file from read access to write access or vice versa while the file is open is the major weakness of sequential files. This fact of sequential file life leads to several consequences:

- Records cannot be directly appended to the end of a file, a typical file operation.
- A record in a file cannot be replaced or changed.
- The contents of a file may have to be read entirely into memory to be processed, the file erased from the disk, and then the new file written to the disk. This is not feasible if the size of the file exceeds the amount of available memory space in the computer. If this is the case, certain types of file processing are not possible.

If there are so many disadvantages to sequential files, why does Macintosh Pascal even bother with them? The answer probably has to do with maintaining compatibility between older implementations of Pascal and newer ones, such as Macintosh Pascal, so that old Pascal programs can be moved or ported on to the Macintosh.

The following program is an example of programming with sequential files. The program opens a new file, places the first 10 integers in it, and then reads them back.

```
program SeqExample;
 var
  IntFile : file of Integer;
  Num : Integer;
 begin
  Rewrite(IntFile, 'Integer.Data');
  for Num := 1 to 10 do
```

```
   begin
    IntFile^:= Num;
    Put(IntFile)
   end;
   Reset(IntFile); {Does a Get too}
   Writeln(IntFile^)
    for Num := 1 to 9 do
     begin
      Get(IntFile); {Get the next 9 records}
     end;
  end.
```

Random Files

Random files, the second Macintosh Pascal file type, have two major advantages over sequential files. Once a random file is opened, both read or write access can be performed without having to close the file first. This allows a program to replace records in a file and append data to the end of a file. The other advantage of a random file is the ability to zoom in and select a specific record in the file without having to access all the other records that sit before it in the file. This ability to directly (or randomly) access any record in the file can greatly speed up file access. For these two reasons, all the programming you do with files will probably be done with random files. In Macintosh Pascal, fortunately, any file that was created with the sequential file procedures can also be accessed as a random file.

Opening a Random File

A file is opened or, if necessary, created for random access with the Open procedure. Open works in a similar fashion to Rewrite; that is, it takes two parameters, the name of a file variable and the name of an external file. If the external file already exists, it is opened; if the external file does not exist, it is created and then opened.

The form of the Open procedure is

```
Open(File, External);
```

Where:

File is a variable declared in the program as type File.

External is a string expression containing then name of the file to be opened or created on the external storage device.

The statement:

```
Open(InventoryFile, 'May.data');
```

would open a file named May.data on the disk drive and associate it with the file variable InventoryFile. If May.Data doesn't exist, it will first be created. Like Reset, Open automatically performs a Get on the first record (record number zero) in the file and then moves the file pointer forward one record.

Writing and Reading Data

Once a random file has been opened with Open, data is written to the file with the Get procedure which operates in the same fashion as it does with a sequential file. Get "gets" the data held in the file variable pointer and places it at the current file position of the external file. Data can be read from a random file in either of two ways. The first method is to use the Put procedure just as you would when working with a sequential file. The second way of reading data from a random file is to use the Seek procedure. Seek moves the file position directly to a specific record number in the file, and then it copies the contents of that record into the file pointer. This is similar to a Get, but Seek does not also advance the file position. This leaves the file position in the correct spot to replace the record, if needed. The form of the Seek procedure is:

```
Seek(File, N)
```

Where:

File is the name of a file variable declared in the program and opened.

N is an integer expression that specifies the record number to be accessed. (The first record in a file is zero.)

Working with Random Files

As a simple example of using random files, plus the Seek procedure we can rewrite the short program that demonstrated sequential file access.

```
program Random_Example;
 var
  Phile : file of integer;
  I : Integer;
begin
 Open(phile, 'test2.data');
 for I := 0 to 4 do
  begin
   Seek(phile, I);
   Phile^ := I + 15;
   Put(Phile);
  end;
  for I := 0 to 4 do
  begin
   Seek(Phile, I);
   Writeln(Phile^)
  end;
 end.
```

In this program, data was placed in the file by "seeking" the record position with Seek and then doing a Put. Data was read from the file with Seek, which both pinpoints a specific record and then places the data in the file pointer.

Finding the End of a File

Typically, a program needs to know if any data remains in a file to be read. Pascal provides a facility to determine whether any components of a file are available to be read as input. The EOF (End of File) function is a built-in function that returns the Boolean value True if the current file position is beyond the last component of a file and False otherwise. The form of EOF is

```
EOF(Filename)
```

where Filename is a file variable of an already open file.

EOF can be used in any situation in which you want to identify the last record in a file; for example, to drive a loop that reads data from each record of a file, or to position the file pointer at the end of a file so that new records can be appended. However, EOF has several subtle characteristics in certain situations:

- When a Put is performed, EOF is True if the new file position is beyond the end of the file. Incidentally, the value of the file pointer is undefined at that point and should not be referenced.
- When a Get is performed, if no file component exists, EOF becomes True, and the value of the file pointer becomes undefined and should not be referenced.
- When a Seek is performed, if the file component requested is greater than the number of components in the file, EOF becomes True.
- When a Reset is performed, EOF is True if the file is empty and False otherwise. This is a way to tell if Reset has opened an existing file or if it has created a new empty file.
- When a Rewrite is performed, EOF is *always* True.
- When an Open is performed, EOF is True if the file is empty and False otherwise. This is a way to tell if Reset has opened an existing file or if it has created a new empty file.

The following programs point out some important peculiarities about using the EOF function. In EOF_Example1, a new file of integers is created for random access with the Open procedure. Three integer values are placed in the file and then a While loop is used to read the values back from the file.

```
program EOF_Example1;
 var
 Phile : file of Integer;
 I : Integer;
begin
 Open(Phile, 'Test.Data'); {Open file}
 {Write data}
 Phile^ : = 10;
 Put(Phile);
 Phile^ : = 20;
 Put(Phile);
 Phile^ : = 30;
 Put(Phile);
 I : = 0 ;
 {Read data}
 while not(EOF(Phile)) do
   begin
    Seek(Phile,I);
    I : = I + 1;
    Writeln(Phile^)
   end
end.
```

This program does not work as intended because of the value of the EOF function after the data is placed into the file. When the new file is opened, the value of EOF becomes True. Since all the records placed into the new file are actually being appended to the end of the file, the value of EOF after each Put is also True. Therefore, when the EOF function is checked as the condition of the While loop, it produces a value of False (*not* True) and the loop is terminated without being executed even once.

```
program EOF_Example1;
 var
 Phile : file of Integer;
 I : Integer;
```

```
begin
 Open(Phile, 'Test.Data'); {Open file} <-----EOF is True
 {Write data}
 Phile^:=10;
 Put(Phile);
 Phile^:=20;
 Put(Phile);
 Phile^:=30;
 Put(Phile); <----- EOF is True
 I:= 0;
 {Read data}
 while not(EOF(Phile)) do <----- EOF is still True and loop not executed
  begin
   Seek(Phile,1); <---------------------------
   I:=I + 1;      <----------------Not Done
   Writeln(Phile^) <----------------------------
  end
end.
```

One way of correcting this problem is to make sure that the file is "rewound" (set the file pointer back to the start of the file) before any attempts are made to read it. One way to do this is to close and then reopen the file before the while loop.

```
program EOF_Example1;
 var
  Phile : file of Integer;
  I : Integer;
begin
 Open(Phile, 'Test.Data'); {Open file}
 {Write data}
 Phile^:= 10;
 Put(Phile);
 Phile^:= 20;
 Put(Phile);
 Phile^:= 30;
 Put(Phile);
 I:= 0 ;
 {Set file pointer back to start}
 Close(Phile);
 Open(Phile);
   {Read data}
 while not(EOF(Phile)) do
```

```
   begin
    Seek(Phile,I);
    I := I + 1;
    Writeln(Phile^)
   end
  end.
```

Another method of correcting the problem is to perform a Seek to record zero record before attempting to read the file.

A second major problem with this program is the order in which the statements appear within the While loop.

```
while not(EOF(Phile)) do
 begin
  Seek(Phile,I);
  I := I + 1;
  Writeln(Phile^)
 end
```

The Seek procedure does not set EOF to True until an attempt is made to seek a record past the last record in the file. In this program, that would be an attempt to seek the fourth record (record number 3). However, when that Seek is performed, the value of the file pointer Phile^ becomes undefined, and the attempt to use it in the Writeln statement produces an error. This subtle bug also occurs if a Get is performed instead of a Seek. This can be demonstrated by running the program as it is using the Step option and setting the Observe window to watch the value of the file pointer and the values of the functions FilePos and EOF. The wonderful debugging tools of Macintosh Pascal should be used often to help demonstrate the complex structures of Pascal.

Looking at our example, which is an attempt to read all the records in the file, we can take advantage of the fact that when we rewind the file back to the start, either with a Seek or Open, the contents of the first record (record number zero) are automatically placed into the file pointer. This value can be used as the first value tested by the While loop. Then by reversing the position of the Writeln and Seek statements, the first value can be used before the next value is read from the file.

```
Open(Phile,'Test.Data); {Also reads the first record}
while not(EOF(Phile)) do
 begin
  Writeln(Phile^)
  I := I + 1;
  Seek(Phile,I);
 end
```

```
program EOF_Example1;
 var
  Phile : file of Integer;
  I : Integer;
begin
 Open(Phile, 'Test.Data'); {Open file}
 {Write data}
 Phile^ := 10;
 Put(Phile);
 Phile^ := 20;
 Put(Phile);
 Phile^ := 30;
 Put(Phile);
 I := 0;
 {Set file pointer back to start}
 Close(Phile);
 Open(Phile, 'Test.Data); {Also reads the first record}
 while not(EOF(Phile)) do
  begin
   Writeln(Phile^)
   I := I + 1;
   Seek(Phile,I);
  end
end.
```

The Haiku Writer

A more sophisticated application utilizing random files is the Haiku generator program. A Haiku is a Japanese lyric poem that is 17 syllables long and often points to a thing in nature that has moved the poet. A modern example might be:

The Macintosh computer is the Apple of my eye, forever.

Since the form of the Haiku is so simple, it makes for easy computer generation. The program operates by using a file, built by the user, which holds words and information about the words. The record structure is called WordRec:

```
WordRec = record
 Word : string[10];
 Part : PartType;
 Syl : Integer
end;
```

Along with a string holding the word is a field holding the number of syllables in the word. There is also a field holding the part of speech of the word expressed as an enumerated data type named PartType. The components of PartType are:

```
PartType = (verb, noun, adj, any);
```

The poems are created by randomly selecting a record number, seeking that record, and seeing if the word held in that record fits into the poem in terms of both number of syllables (doesn't make the poem over 17) and part of speech. Determining whether the part of speech fits is one of the more interesting aspects of the program, and it is done with the use of sets. A set called PartSet is declared to be of PartType. This means that the possible members of the set are the possible values of PartType: verb, noun, adj, and any. An array called Grammars is then declared with PartType as its index type. This means that there are four elements in the array: Grammars[verb], Grammars[noun], Grammars[adj], and Grammars[any]. Each of these elements is then assigned a set of the possible grammatical parts of speech that may legally follow that part of speech. For instance, the element Grammar [noun] is assigned the set:

```
Grammars[adj] := [adj, noun];
```

This set represents the possible parts of speech that may legally follow an adjective. Assume that an adjective is the part of speech of the word just accepted into the poem. Then, when a new word is picked from the file, its part of speech is checked with the use of the **in** operator to see if it is a member of the set of the parts of speech that can follow an adjective. A special part of speech called "Any," which contains all the parts of speech, is used to allow any part of speech to be used as the first word in the poem.

The pseudo code for the program is:

```
   Initialize Grammars and other variables
   Open file
   Display menu to user
   get input
   case input of
1: Get word from user
   Advance to end of file
   write word to file
```

```
2: Found length of file
   Pick a random number
   Get that record
   Does it fit Grammar?
   Will adding this word total <= 17 syllables
3: Close file and exit
```

A second level of refinement adds the procedures needed and some of the programming structures.

```
   procedure AddWord
   begin
   prompt user for word, part of speech, and number of syllables
   advance file to end
   Put record in file
   end
   procedure WritePoem
   begin
   Find number of records in file
   Get a random record
   If part of speech in Grammar of previous word,
     Then Will the word fit the syllable count
       Then Print word
     end
   Initialize grammars and variables
   repeat
   display menu
   get choice
   case choice of
 '1': AddWord
 '2': WritePoem
 '3': close file and Done := True
until Done;
```

The third level of refinement produces the actual Pascal code for the program. The following are some notes about the program.

- The length of the file is determined by first doing a Seek with a record number of MaxInt. This sets the file position to the end of the file, one past the last record in the file. The procedure FilePos can then be used to find the record number. When the file position is at the end of the file, the value returned by FilePos is one more than the record number of the last record. Since record numbers start with zero, it also represents the number of records in the file. Subtracting one from this value will produce the record number of the last record of the file. A random value within the range of the record numbers in the file can be generated by calling the QuickDraw random number function Random and then performing a Mod operation with the number of records in the file.
- In the procedure AddWord the user is required to enter the word (as a string), the number of syllables (as an integer), and the part of speech (as a PartType). This routine takes advantage of Macintosh Pascal's ability to read and write values of an enumerated type as a string. No type checking is done in this routine as long as the user enters the part of speech exactly, there is no problem. If an illegal value is entered, the program will crash. Normally, a programmer will protect against this by reading the value as a string and then using a case statement to assign the proper value to the enumerated variable. The program does an input validity check on the menu selection entered by the user. The value is read as a character and a membership test is performed on a set of legal input values.

```
program HaikuWriter;
 type
  PartType = (verb, noun, adj, any);
  PartSet = set of PartType;
  WordRec = record
   Word : string[10];
   Part : PartType;
   Syl : Integer
  end;
var
 PoemWord, Previous : WordRec;
 WordFile : file of WordRec;
 Done : Boolean;
 InSet : set of Char;
 Grammars : array[PartType] of PartSet;
 Ch : Char;
 SylCount : Integer;
```

```
procedure AddWord;
begin
 Write('Word to Add --> ');
 Readln(PoemWord.Word);
 Write('How many Syllables --> ');
 Readln(PoemWord.Syl);
 Write('Which part of speech (noun, adj, or verb) --> ');
 Readln(PoemWord.Part);
 WordFile^ := PoemWord; {Assign record to file pointer}
 Put(WordFile)
end; {AddWord}
procedure WritePoem;
 var
 Pick, Size : Integer;
begin
 Seek(WordFile, Maxint); {Move to end of file}
 Size := FilePos(WordFile); {Find number of records}
 Previous.Part := Any;
repeat
 Pick := Random mod Size - 1; {Pick a random record}
 Seek(WordFile, Pick); {Get it!}
 If WordFile^.Part In Grammars[Previous.Part] then {Grammar check}
  If SylCount + WordFile^.Syl <= 17 then
   begin
    Writeln(WordFile^.Word); {Print word}
    Previous := WordFile^; {New word becomes old}
    SylCount := SylCount + WordFile^.Syl;
  end;
 until SylCount = 17;
end;{WritePoem}
begin
 {Initialize variables and grammars}
 InSet := ['1', '2', '3'];
 SylCount := 0;
 Grammars[Noun] := [adj, verb];
 Grammars[adj] := [ adj, noun];
 Grammars[verb] := [adj, noun];
 Grammars[any] := [adj, noun, verb];
 Done := False;
 Open(WordFile, 'Word.Data');
repeat
 Writeln('1 : Add a word'); {Display menu}
 Writeln('2 : Write a Poem');
 Writeln('3 : Quit');
```

```
repeat
 Writeln;
 Write('Selection --> ');
 Read(Ch);
 Writeln
until Ch in InSet;
case Ch of
 '1' :
 AddWord;
 '2' :
 WritePoem;
 '3' :
 Done := True;
end
until Done;
Close(WordFile)
end.
```

Indexed Sequential Access Method

Many applications require a program to search through a file for specific information. Depending on the number of records in the file, the size of each record, and the number of times it is searched, a tremendous degradation of program speed may occur. Several programming techniques have been developed to allow faster data file access. For instance, if the file field that is most constantly searched is kept in sort order, file searches can be done with a binary search algorithm. However, if there are many additions or deletions from the file, constant sorting of the file will negate any speed gained in searching.

Another popular technique is the use of an index into the file, sometimes called ISAM for indexed sequential access method. ISAM requires the maintenance of an index to the file in an array. The array contains a duplication of one of the fields in the file to be searched plus the record number of the corresponding record in the file (Figure 3.6). This field, called the *key field*, is the part of the record that is most often (or the only field) searched for. When a search is requested, the key field in the index is searched. Since the index is held in memory, this search—done either sequentially or with a binary search if the array is sorted—is quick. The search is complete when either a match is found (providing the record number) or when there are no more positions in the array to be examined (the desired record doesn't exist).

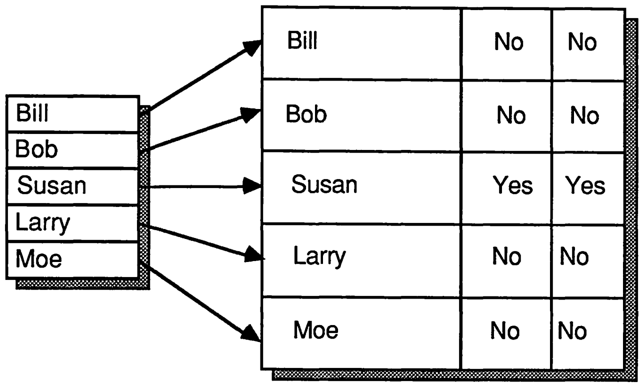


Figure 3.6 An ISAM System

The following program uses an ISAM technique. This program maintains a file holding a disk index for a disk library—which turns out to be a pretty useful application if you have nearly as many disks as I do. The record structure used to hold the information for a disk is quite interesting; it incorporates an array in a record.

```
type
 DiskRecord = record
  DiskName : string[30];
  Files : array[1..10] of string[20]
 end;
```

This essentially creates a record with 11 fields, the disk name and an array which holds 10 file names. The record can be pictured as shown in Figure 3.7.

A collection of these records is held in a file named DiskFile, which is associated with an external file called 'Disk.Data'. The user can add new disk records to the file or search for a particular disk record by providing its name.

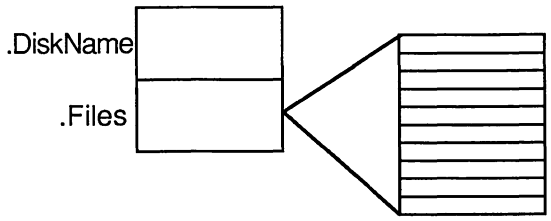


Figure 3.7 The record type DiskRec

When the program is started, the first thing done is the building of the index array. This is done by reading the records sequentially and assigning the DiskName field of each record to a position in the array called Isam. Isam is a one-dimensional array in which the record number is the same as the array index for each DiskName in the array. To facilitate this, the array is declared to start with zero as its first subscript:

```
Isam : array[0..99] of string[30];
```

All searching is performed on this array in a sequential fashion. Since the record number is encoded as the array position, the array cannot be sorted to allow binary searching. Alternatively, the array could have contained the record number in a second field, and then it could be sorted.

The basic structure of the program consists of three procedures that build the index, search the array, and add a new record to the file.

```
program DiskIndex;
 type
  DiskRecord = record
  DiskName : string[30];
  Files : array[1..10] of string[20]
  end;
  InputSet = set of Char;
 var
  ThreeSet : InputSet;
  I, Ct : Integer;
  ArrayEnd : Integer;
  DiskFile : file of DiskRecord;
  InputRec : DiskRecord;
  Isam : array[0..99] of string[30];
  Temp : string[20];
  Done : Boolean;
  Ch : Char;
 procedure BuildIndex;
 var
  I : Integer;
 begin
  Open(DiskFile, 'Disk.data');
  if EOF(DiskFile) = False then {Check if file exists}
  begin
   Ct := 0;
   Seek(DiskFile, Ct);
  begin
   repeat
    Isam[Ct] := DiskFile^.DiskName;
    Ct := Ct + 1;
    Seek(DiskFile, Ct);
    until Eof(DiskFile);
   ArrayEnd := Ct - 1;
   end
  end
  else
   ArrayEnd := 0
 end; {BuildIndex}
 procedure Review;
  var
   I : Integer;
   Found : Boolean;
 begin
  Page(Output);
  Writeln('Enter name of disk to review');
  Readln(Temp);
  I := 0;
 repeat
  if Isam[I] = Temp then
   Found := True
  else
   I: = I + 1;
 until (I = Ct) or (Found = True);
 if Found = True then
 begin
  Seek(DiskFile, I);
  for I := 1 to 10 do
  Writeln(I : 2, ':', DiskFile^.Files[I])
 end
else
 Write('That disk is not in the index, sorry');
end; {procedure Review}
```

```
procedure Add;
begin
 with InputRec do
  begin
  Page(Output);
  Write('Enter Name of disk: ');
  Readln(DiskName);
  writeln('Enter file names, enter return to stop');
  I:=1;
  Done := False;
  repeat
   Write('Enter a file name:');
   Readln(Temp);
   if Temp <> ''then
   begin
    Files[I] := Temp;
    I := I + 1
   end
  else
   Done := True;
  until Done;
  Seek(DiskFile, MaxInt);
  DiskFile^ := InputRec;
  Put(DiskFile);
  Isam[ArrayEnd] := DiskName;
  ArrayEnd := ArrayEnd + 1
 end; {with}
end; {procedure Add}
begin
 ThreeSet := ['1', '2', '3'];
 BuildIndex;
 repeat
  repeat
  Writeln('1: Review a disk');
  Writeln('2: Add a disk');
  Writeln('3: Quit');
  Writeln;
  Readln(Ch);
 until Ch in ThreeSet;
 case Ch of
  '1':
  Review;
  '2':
  Add;
```

```
   otherwise
   ;
  end; {case}
  until Ch = '3'
 end.
```

This program assumes a decent amount of free memory space to work with since the index requires 31 bytes for each record in the file. This number equals the 30 characters in each string plus one. A string occupies an extra byte in memory to hold the length of the string (the reason a string is limited to 255 bytes). In our program, since the array has 100 elements, this preallocates 3100 bytes of memory. Should memory space be tight, the program could also be written using a separate disk file to hold the index information. Since this file would be much smaller than the data file, searching it would take substantially less time. Alternatively, the memory space for the array could be dynamically allocated after checking the number of records in the file. More space could then be added to the array as needed during processing.

The program could also be expanded to include several other functions. Probably the most desirable feature to add is the ability to delete a record from the file. To accomplish this, all the records in the file following the deleted record would have to be shifted one position up in the file. This type of file manipulation is generally a waste of time and resources. A better way is to mark the record to be deleted with a code indicating that the record has been deleted. The position in the array should also be marked as deleted. This does not physically remove the record from the file, but when storing the array back to the file, any record marked deleted is not written and thus removed forever.

CHAPTER 4 Events

Macintosh Pascal exists as a dichotomy. On one hand, Macintosh Pascal operates as an ordinary Pascal system; it runs standard Pascal programs using ordinary Pascal structures such as files, Readln and Writeln. On the other hand, Macintosh Pascal has to effectively work with the complex Macintosh Pascal operating environment with which it constantly interacts. This situation has some interesting consequences. For instance, one reason that Macintosh Pascal is a highly desirable system to program with is that the programmer need not worry much about dealing with the Toolbox and operating system; however, Macintosh Pascal does allow several Toolbox features to seep through to the programmer. One of the most important of these features is event handling. Events are the Macintosh's way of responding to actions taken by the user—clicking the mouse button or typing on the keyboard. These events can be passed through Macintosh Pascal and on to your program.

This chapter explores the event-handling system, the structure of events, the Macintosh routines used to handle events, and the use of these routines in a Macintosh Pascal program.

Why Use Events?

Events can relieve some sticky programming situations that may occur. An example of this type of situation is a program that allows either keyboard or mouse input at the same time. This is difficult to achieve since execution of a Read or Readln statement causes the program to sit and wait for the input to be entered on the keyboard; it does not allow anything else to happen while it is waiting. Macintosh Pascal is very patient and will gladly sit and wait forever.

```
Read(Ch); <-----The program will wait here
GetMouse(X,Y);
```

In the preceding two lines of Pascal code, the GetMouse procedure will not be performed until input is entered on the keyboard. If you want the mouse input before or instead, you're out of luck.

Event Types

Two separate sets of low-level Macintosh routines are used to manage events. They are the operating system's Event Manager, which interfaces with the Macintosh's hardware, and the Toolbox's Event Manager, which works with the application programs. The operating system's Toolbox Manager responds to user actions such as clicking the mouse or clicking a key. When one of these user responses takes place information about it is placed into a list known as the Event Queue (Figure 4.1). The Event Queue is a first-in, first-out (FIFO) structure in which the first event to occur is placed at the head of the list, and it is the first event to be processed.

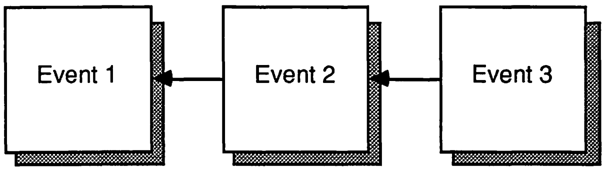


Figure 4.1 The Event Queue

There are 16 different classifications of events that are tracked by the operating system's Event Manager. The most important ones fall into these categories.

- Mouse Events—There are separate events for when the mouse button is depressed (Mouse Down Event) and released (Mouse Up Event).
- Keyboard Events—There are separate events for when a key is depressed (Keyboard Down Event), when a key is held down (Auto Key Event), and when a key is released (Key Up Event).
- Null Event—No event has occurred.

The following event categories are of less importance to Macintosh Pascal programmers and are included for completeness.

- Disk Event—An event is generated (Disk Insertion Event) when a disk is inserted into a disk drive.
- Window Events—There are separate events generated whenever a window is made active (Activate Event), when a window is made inactive (Deactivate Event) and when the active window needs to be redrawn (Update Event).
- Network Events—An event can be generated by an AppleTalk device hooked up to the Macintosh's serial port.
- Application Events—Up to four different types of applications-defined events are supported.

Each type of event has its own event code.

Table 4.1

Event	Code
Null Event	0
Mouse Down Event	1
Mouse Up Event	2
Key Down Event	3
Key Up Event	4
Auto-key Event	5
Update Event	6
Disk Insertion Event	7
Activate Event	8
Network Event	10
Device Driver Event	11
Application1 Event	12
Application2 Event	13
Application3 Event	14
Application4 Event	15

Event Records

When an event is detected, it is posted to the Event Queue by adding an event record containing all the pertinent information about that event. The information included in the event record is:

1. The type of event (event code)
2. The time the event occurred expressed in ticks (1/60th's of a second) since the system start-up occurred.
3. The location of the mouse, expressed in global coordinates, at the time of the event.
4. The state of the mouse button and the modifier keys in the keyboard at the time of the event.
5. Additional information particular to the type of event that occurred.

Event records are a predefined Macintosh Pascal data type defined as:

```
type
 EventRecord = record
  What : Integer;
  Message : LongInt;
  When : LongInt;
  Where : Point;
  Modifiers : Integer
 end; {Event Record}
```

This record definition does not have to be defined in your program to be used since it is predeclared. For instance, to declare an event record named Event:

```
var
Event : EventRecord;
```

The fields in an event record have the following meanings (Figure 4.2):

What—indicates the type of events with the event codes listed above.

When—holds the time of the event as the number of ticks since system start-up.

Where—holds the mouse location, expressed as a point, at the time of the event.

Modifier—holds the state of the modifier keys and the mouse button expressed as the sum of the predeclared Macintosh Pascal constants.

```
const
 ActiveFlag = 1;
 btnState = 128;
 cmdKey = 256;
 shiftKey = 512;
 alphaLock = 1024;
 optionKey = 2048;
```

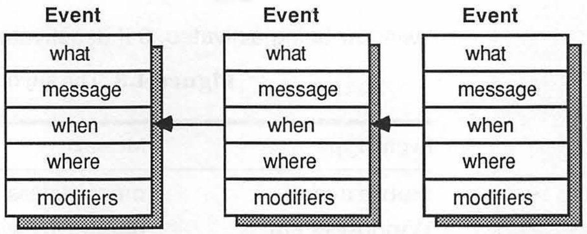


Figure 4.2 Event records in the event queue

The following are also predefined Macintosh Pascal constants that may be used without declaring them in your program (Figure 4.3). These constants represent the modifier flags held by the modifier field of the event record.

Notice that the btnState bit is set(value of 1) when the mouse button is up but the bits for the modifier keys are set if the keys are down. If more than one of these conditions exists, the value of the modifier field is the sum of the bits set. For instance, if at the time of the event both the shift and the option keys are being held down, the value of the modifier field will be the value of shiftKey + optionKey (512 + 2048) which equals 2560.

Message — The message field is a longint that conveys additional information about the event. The context of the message changes with the type of event and can be summarized as follows:

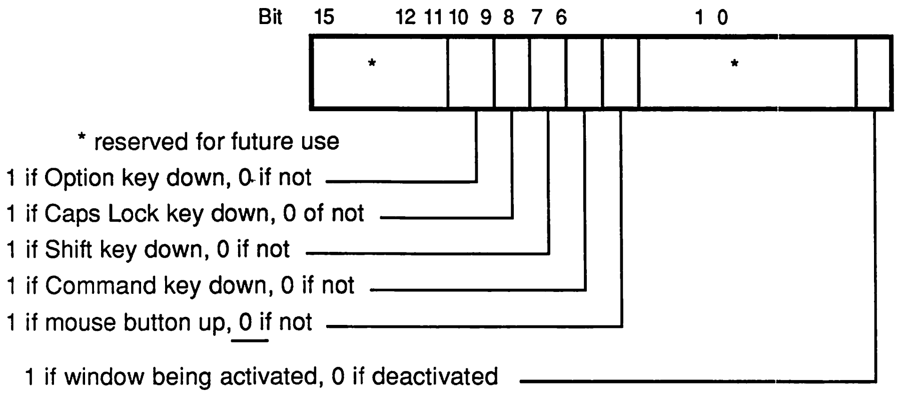

Bit 15 12 11 10 9 8 7 6 1 0
*
*
* reserved for future use
1 if Option key down, 0 if not
1 if Caps Lock key down, 0 of not
1 if Shift key down, 0 if not
1 if Command key down, 0 if not
1 if mouse button up, 0 if not
1 if window being activated, 0 if deactivated

Figure 4.3 The keyboard modifiers

Event Type	Message
Mouse event	meaningless
Window event	pointer to window
Keyboard	character code and key code

Keyboard Events

When a keyboard event occurs (key up, auto, or down), the message field of the event record contains two pieces of information held together: the ASCII character code for the character and the key code (Figure 4.4).

How do these values differ? The ASCII code represents the actual character selected by the user and is affected by the modifier keys. For instance, the code for a capital C is different then a lower-case *c*. The key code (Figure 4.5) represents the key that was depressed on the keyboard and is the same regardless of any modifiers pressed simultaneously.

Either of the two values held in the keyboard event record message field can be separated with the proper use of a mask.

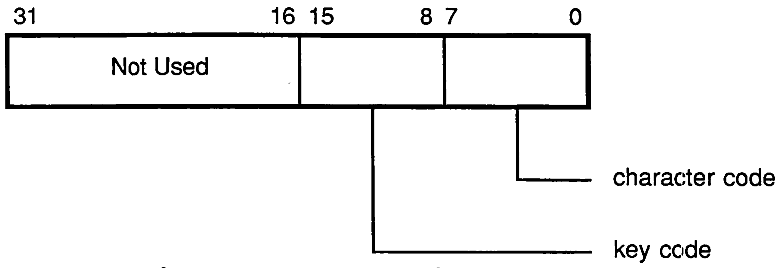


Figure 4.4 Event message for keyboard events

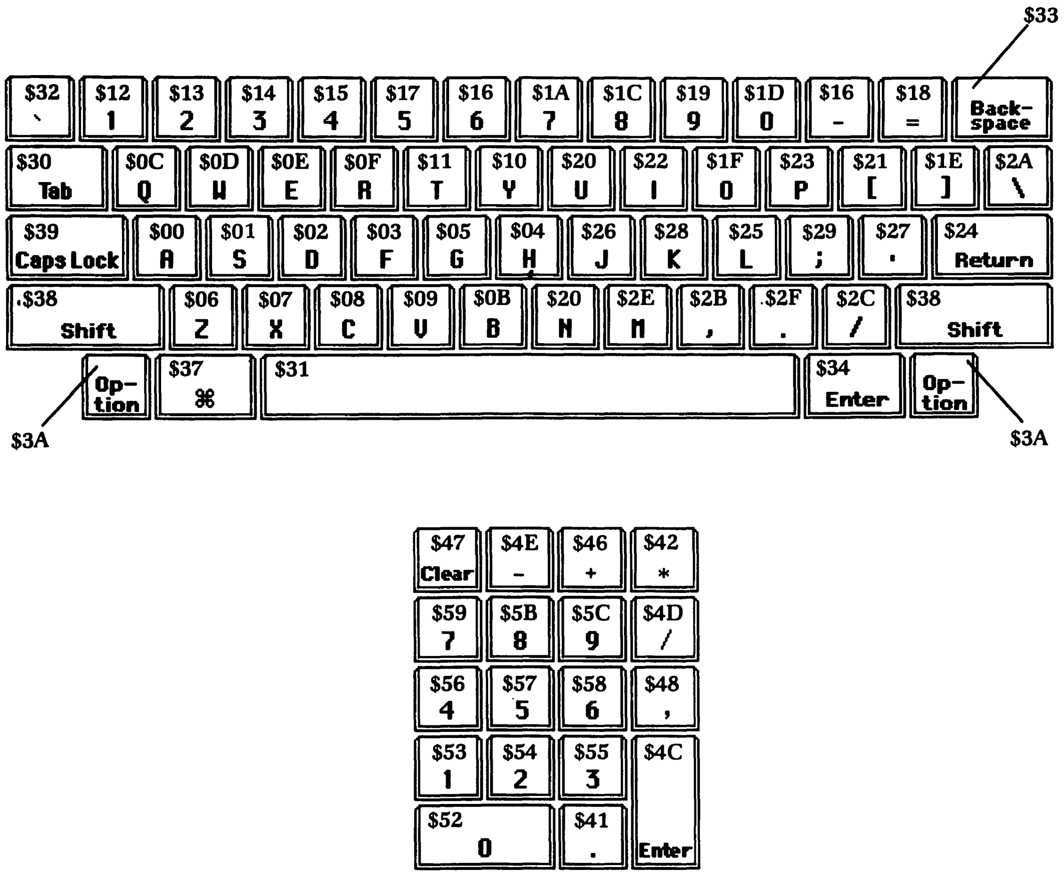


Figure 4.5 Macintosh key codes

Using Bit Masks

Using a mask is an ancient programming technique left over from the days when all programming was done in machine language. It involves taking a variable, such as an integer, and viewing the individual bits inside of it. Normally, there is no need to do this while programming in a high-level language, but in certain situations, a single variable is used to encode different pieces of information by logically dividing the variable into several fields. Such is the case with the event record message field when it holds a keyboard event. For example, when the keyboard event involved typing a lower-case *b*, the message field would look like this:

```
0000 1011 0110 0010
0    B    6    2 (Hex value)
```

The contents of this long integer may appear strange. We are used to thinking of the value of a long integer variable as a decimal value, but it is actually stored as a series of 32 binary digits (bits). Binary values are very often expressed as hexadecimal (hex) values (base 16). If you are unfamiliar with binary and hex or need to brush up on them, Appendix C provides a short review.

In order to separate the two discrete values held in the long integer, it is necessary to operate directly on the bits themselves. This requires using the logical AND operator and a properly selected mask value. When two bits are ANDed, the result is:

First Bit	Operator	Second bit	Result
0	AND	0	gives 0
1	AND	1	gives 0
0	AND	1	gives 0
1	AND	1	gives 1

Notice that when the second bit is zero, the result is always zero regardless of the value of the first bit; but when the second bit is 1, the result is always the same as the value of the first bit. This fact can be used to screen out the bits we are not interested in and to let the bits we wish to examine filter through. Specifically, in order to isolate the rightmost eight bits of the message, we can AND it with a bit pattern that is designed to only allow those eight bits to filter through. This type of bit pattern is known as a mask.

```
0000 1011 0110 0010
0000 0000 1111 1111   < =mask
-------------------
0000 0000 0110 0010
```

The value of the result is the same as the value of the rightmost eight bits, thus the value is isolated.

In Macintosh Pascal we can perform an AND with the BitAnd function. It's form is:

```
function BitAnd(long1, long2 : LongInt) : LongInt;
```

where long1 and long2 are both long integers.

Specifically, we could use one of the two long integers as our value and the other as the mask in either decimal or hexadecimal.

```
BitAnd(Event.message, 255)
```

or

```
BitAnd(Event.message,$FF)
```

The isolation of the second set of eight bits is slightly more complicated. First, let's look at the mask needed to filter these bits.

```
0000 1011 0110 0010
1111 1111 0000 0000   <--- mask
0000 1011 0000 0000
```

The value of this mask is decimal 65280, or hexadecimal $FF00. The result of the AND isolates the eight bits we are interested in, but they are not in the proper position to be used as an arithmetic value; that is, they are not the rightmost eight bits of a variable. They must now be shifted eight bits to the right. The Macintosh Pascal BitShift function is used for this purpose.

```
function BitShift(long : LongInt; count : Integer) : LongInt;
```

where long is a long integer value and count is the direction of the shift and the number of bits to move. The direction is determined by the sign of count: positive shifts to the left, negative shifts to the right. The number of bit positions to move in the desired direction is the absolute value of count **mod** 32. This means that if the value of count is less than 32, it is simply count number of bits.

In our example, to shift the value eight bits to the right, we can use:

```
L := BitAnd(E.Message, $FF00);
ASCIICode := BitShift(L, -8); {Shift 8 bits to the right}
```

The remaining bit operations are:

```
BitOr(Long1, Long2)
```

returns the logical OR of two long integers. The table of results of the logical OR operation is :

First Bit		Second Bit	
0	OR	0	gives 0
1	OR	0	gives 1
0	OR	1	gives 1
1	OR	1	gives 1

```
BitXOr(Long1, Long2)
```

returns the logical Exclusive OR of two long integers. The table of results of the logical Exclusive OR operation is:

First Bit		Second Bit	
0	XOR	0	gives 0
1	XOR	0	gives 1
0	XOR	1	gives 1
1	XOR	1	gives 0

```
BitNot(Long)
```

returns the logical NOT of a long integer. The table of results of the NOT operation is:

BitNOT	0	gives	1
BitNOT	1	gives	0

Using Events

So far in this chapter a lot has been said about events, but no way to access them from a Macintosh Pascal program has been presented. To utilize events, the Event Queue is searched for an event of the desired type. If it is found, it is removed from the queue and passed to the program as a parameter. In Macintosh Pascal as in any program written for the Macintosh, this is performed by the function GetNextEvent, a direct implementation of a Toolbox routine. GetNextEvent causes the Toolbox Event Manager to search the Event Queue for an event of a specific type. If an event is found, the event record is removed from the queue and passed to the program in a variable parameter.

```
function GetNextEvent(Mask : Integer; var Event : EventRecord) : Boolean;
```

where:

mask is an integer used to specify what kind of events are to be removed from the Event Queue

event is a variable of type EventRecord that receives the event record from the Event Queue as a variable parameter.

GetNextEvent is a function that returns two values. The result of the function is a Boolean value, True, if an event record is found and False otherwise. If the result is True, the actual event record is passed to the function as a variable parameter.

The mask used as the first parameter in the function describes what event type to pull off the Event Queue. The mask is the sum of the event masks of the specific event types that you want passed to your program. The event masks have been conveniently defined as Macintosh Pascal constants.

```
const
  NullMask = 1;
  MDownMask = 2;
  MUpMask = 4;
  KeyDownMask = 8;
  KeyUpMask = 16;
  AutoKeyMask = 32;
  UpdateMask = 64;
  DiskMask = 128;
  ActiveMask = 256;
  AbortMask = 512;
  ReserveMask = 1024;
  DriverMask = 2048;
  App1Mask = 4096;
  App2Mask = 8192;
  App3Mask = 16384;
  App4Mask = 32768;
```

Since these constants are predeclared, they are recognized by Macintosh Pascal without needing to declare them in your program. For instance, if you want to receive all mouse events (up and down), you would use a mask of MDownMask + MUpEvents, 2 + 4, or 6.

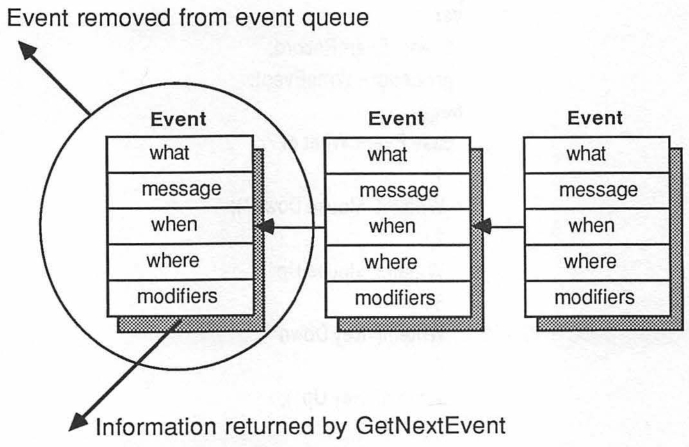


Figure 4.6 The event masks

Remember, GetNextEvent returns only the first event on the queue that matches the mask. Since the Event Queue is a FIFO structure, the first event of that type to have occurred is the first sent to the program.

Programming with GetNextEvent

Using events changes the usual nature of the program you are writing. Event-driven programs often consist of a loop which continually calls GetNextEvent until an event occurs and then processes it.

The following simple example of an event-driven program simply waits for an event to occur and then displays the event type in the Text window.

```
program FirstEvent;
 var
  Event : EventRecord;
begin
 while not( GetNextEvent(62, Event) do
  Writeln('Waiting for an event');
 Writeln(Event.What)
end.
```

When this program is run the 'Waiting for an event' message is displayed by the while loop until an event occurs. At that point, the value returned by GetNextEvent is True (then reversed for use as the condition of the while loop) and the loop is terminated. Notice that when the program is running, the mere repositioning of the mouse does not create an event. This program can be improved with the use of a procedure that uses a case statement to display a message about the event code.

```
program FirstEvent;
 var
  Event : EventRecord;
  procedure WriteEvent;
 begin
  case Event.What of
   1:
   Writeln('Mouse Down');
   2:
   Writeln('Mouse Up');
   3:
   Writeln('Key Down');
   4:
   Writeln('Key Up');
   5:
   Writeln('Auto Key');
  end; {case}
```

```
end; {procedure WriteEvent}
begin
 while not(GetNextEvent(62, Event) do
    Writeln('Waiting for an event'); {only statement in loop}
 Writeln(Event.What);
 WriteEvent
end.
```

The next example program displays the message field of the event record for a keyboard down event. Unlike the previous example where the loop terminated at the first event that occurred, this program will allow several successive events to occur. A slightly different programming technique will be needed to allow the program to eventually terminate. This is normally not a problem when programming in Macintosh Pascal because the Pause menu is always available to halt a wayward program. However, once GetNextEvent is called, Macintosh Pascal's own event-handling capability is disabled with the Pause menu highlighted, yet useless since Macintosh Pascal will never sense the mouse down event in the menu bar. It has been passed to your program. Once the running program terminates, Macintosh Pascal regains its lost event-handling capability. Our new programming example implements a timer by using a variable that is incremented once for each iteration of a repeat loop. The loop will terminate when the timer reaches a predetermined value.

```
program EventTwo;
 var
 Timer : Integer;
 Event : Eventrecord;
 L : Longint;
begin
 Timer := 0;
 repeat
  if GetNextEvent(127, E) then
   begin
    L := BitShift(BitAnd(Event.Message, $FF00), -8);
    Writeln(Event.Message, L, BitAnd(Event.Message, $ff))
   end;
  Timer := Timer + 1;
 until Timer = 1000;
end.
```

Notice that this program utilizes a repeat loop rather than a while loop, and GetNextEvent is called from inside an If statement. This is more typical of event-handling programs on the Macintosh.

Our next programming example involves mouse down events. When any event occurs, the Where field of the event record contains the location of the mouse at the time of the event, expressed in a point. If the event record is Event then the position of the mouse would be held in the Where field of the event record, which is a point. In our example the fields are Event.Where.H and Event.Where.v. The following program waits for a mouse event and then displays the location of the mouse in the Text window. Notice that the coordinates displayed are screen coordinates with the upper left-hand corner of the screen being point 0,0.

```
program MouseEvent;
 var
  Timer : integer;
  Event : Eventrecord;
 begin
  repeat
   if GetNextEvent(6, Event) then
    begin
    Writeln(Event.Where.H, Event.Where.V)
    end;
   Timer := Timer + 1;
  until Timer = 1000;
end.
```

EventAvail

The Toolbox Event Manager has a second function that is called EventAvail similar to GetNextEvent. EventAvail (Figure 4.7) scans the Event Queue just like GetNextEvent, and if the event is found, it only reports back to the program of it's existence but does not remove the event from the queue. This means that a second call to EventAvail or GetNextEvent with the same event mask will return the same event. EventAvail is often used as a way to look at items in the Event Queue without removing them so that they can be processed at a later time. The form of EventAvail is:

```
function EventAvail(Mask : Integer; var Event : EventRecord) : Boolean;
```

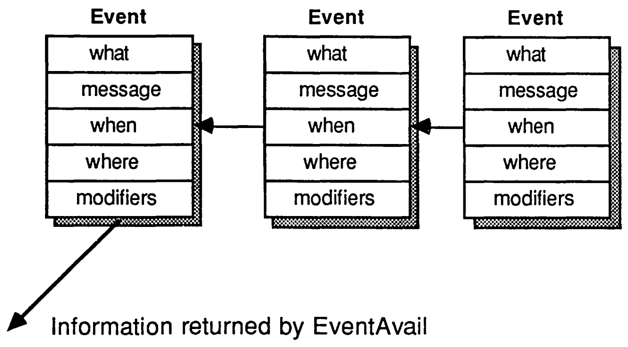


Figure 4.7 The EventAvail

The next programming example demonstrates the differences between Event-Avail and GetNextEvent. The program has two event loops, the first using EventAvail and the second GetNextEvent. When an event is detected the Event.What field is displayed in the Text window along with the output of the WriteEvent procedure. To help differentiate the output from each loop, a message is displayed prior to entering both. When the program is executed you will notice that the first event discovered by EventAvail is the only event that is displayed. This is because each time EventAvail is called it starts scanning at the start of the Event Queue. Any other event of the same event type will remain undiscovered. When the GetNextEvent loop starts, the same event revealed by EventAvail will be the first event returned by GetNextEvent.

Notice that the timer used for the EventAvail is shorter than that for GetNextEvent since 1000 calls to EventAvail will return the same event 1000 times, but 100 calls to GetNextEvent will return the event only once.

```
program LookAtEvents;
 var
  Timer : Integer;
  Event : EventRecord;
 procedure WriteEvent;
 begin
  case Event.What of
   1 :
   Writeln('Mouse Down');
   2 :
   Writeln('Mouse Up');
   3 :
   Writeln('Key Down');
   4 :
   Writeln('Key Up');
```

```
    5:
    Writeln('Auto Key');
   end; {case}
  end; {procedure WriteEvent}
  begin
   Writeln('**EventAvail loop**');
  repeat
   if EventAvail($FF,Event) then
    begin
     Writeln(Event.What);
     WriteEvent
    end
    Timer := Timer + 1;
   until Timer = 10;
   Timer := 0;
   Writeln('**Get next event loop**');
   repeat
    if GetNextEvent($FF,Event) then
     begin
      Writeln(Event.What);
      WriteEvent;
     end;
    Timer := Timer + 1;
   until Timer = 1000;
 end.
```

Flush Events

The procedure FlushEvents is used to remove events from the Event Queue without passing them back to the program. This Operating System Event Manager routine is commonly used to make sure that no stray events are sitting in the Event Queue from the time prior to the execution of your program. The form of FlushEvents is:

```
procedure FlushEvents(EventMask, StopMask : Integer)
```

Where:

EventMask is the sum of the event masks for the type of events to remove from the queue.

StopMask is the event mask of the type of event at which FlushEvents will stop.

FlushEvents moves down the Event Queue, removing all the events of the specified type up to but not including the first occurrence of the event type specified as the StopMask. If the Event Queue contains no events of the type specified, then no action is taken. A StopMask of 0 will remove all events of the specified type. To remove all events of all types from the Event Queue (to make sure that it is empty), use an EventMask that represents all events and a StopMask of 0:

```
FlushEvents($FFFF,0) {removes all events from queue}
```

CHAPTER

5

QuickDraw Programming Techniques

If you have spent any time working with Macintosh Pascal you have no doubt included QuickDraw routines in your program. The QuickDraw graphics package was originally developed for the now defunct Apple Lisa computer and was transported to the Macintosh. It forms the foundation on which the entire Macintosh user interface is based and is arguably the most impressive part of the advanced technology that is the Macintosh.

QuickDraw is built into the Macintosh's read only memory along with the Toolbox and operating system routines. It is directly supported by Macintosh Pascal, meaning that all of the QuickDraw routines and data types are accepted by the interpreter as though they were part of standard Pascal. This does not mean that all of the QuickDraw procedures and functions can be used with impunity. There are several areas of QuickDraw that should not be experimented with since doing so may interfere with the way Macintosh Pascal interfaces with QuickDraw and the operating system. These areas will be pointed out as they are discussed.

Since this book deals with advanced Macintosh Pascal techniques and features, it is assumed that you have some knowledge of the fundamental concepts related to QuickDraw. This includes the QuickDraw coordinate system, points, rectangles, the five drawing operations (Frame, Erase, Paint, Fill, and Invert), and calculations with rectangles (most notably the PtInRect function). All these simple concepts will be used repeatedly throughout the remainder of this book.

This chapter discusses specific QuickDraw programming techniques using many of the topics just mentioned and Chapter 7 covers some of the more advanced QuickDraw topics. These programming techniques highlight the use of QuickDraw to provide useful graphic interfaces for your programs. This includes using QuickDraw to simulate the standard Macintosh user-interface features such as pushbuttons and control buttons. These techniques include examples that use event handling.

The GridEdit Program

Our first programming techniques involve the use of a large number of rectangles and the PtInRect function. The GridEdit program (Figure 5.1) is a building block of a program that edits graphical elements such as the cursor. This program displays an eight-by-eight grid of rectangles and allows the user to select which rectangles are white or black. Eventually, this program will provide a function similar to the FatBits operation in the MacPaint program, but at this point the GridEdit simply tracks which grid elements are ON (black) or OFF (white).

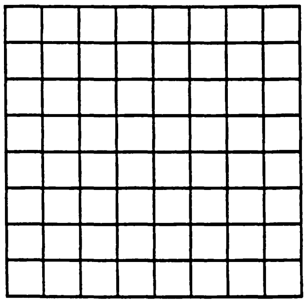


Figure 5.1 The grid from GridEdit

The eight-by-eight grid is constructed by declaring a two-dimensional array of rectangles.

```
type
 GridSize = 1 .. 8;
var
 Grid : array [GridSize, GridSize] of Rect;
```

Each rectangle is defined as a ten-by-ten point square with the top leftmost square starting at point (20, 20). Figure 5.2 shows the coordinates for all the squares which are defined as follows:

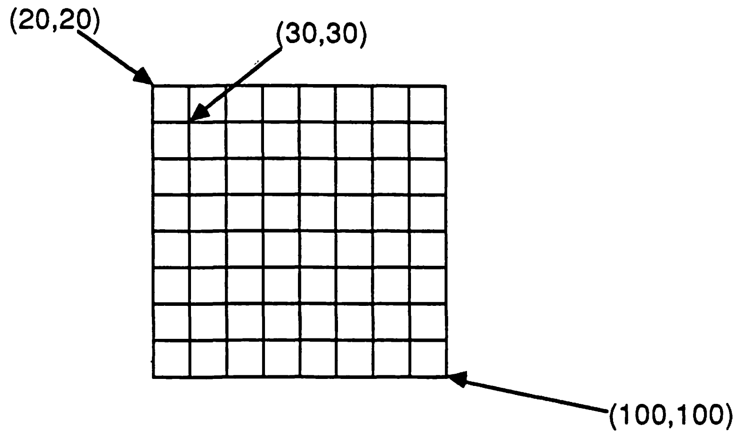


Figure 5.2 The coordinates of the GridEdit grid

If we think of the grid as eight columns and eight rows then the following formulas define the coordinates for each of the rectangles.

Upper Left point: 10 + Row * 10, 10 + Column * 20

Lower Right point: 20 + Row * 10, 20 + Column * 10

Two nested For loops are used to initialize the rectangles.

```
for Col := 1 to 8 do
 for Row := 1 to 8 do
  begin
   SetRect(Grid[Row, Col], 10 + Row * 10, 10 + Col * 10, 20 + Row * 10, 20 + Col * 10);
   FrameRect(Grid[Row, Col]);
 end;
```

A loop then waits for a mouse event. Then the nested loops are used with the PtInRect function to test all the rectangles to see where the mouse was clicked.

```
repeat
until button;
 GetMouse(Pt.H, Pt.V);
 for Col := 1 to 8 do
  for Row := 1 to 8 do
   if ptInRect(Pt, Grid[Col, Row]) then
```

In order to track the state of each rectangle (white or black) a second array of Boolean values is also needed.

```
Mark : array [GridSize, GridSize] of Boolean;
```

By convention, False will represent an OFF (or white) rectangle and True represents an ON (or black) rectangle. When a white rectangle is selected it is turned black with FillRect; when a black rectangle is selected it is turned white with FillRect, and then FrameRect is called to restore its outline. The corresponding position in the Mark array is then switched.

```
if PtInRect(Pt, Grid[Col, Row]) then
 begin
  Mark[Col, Row] := not (Mark[Col, Row]);
  if Mark[Col, Row] then
   FillRect(Grid[Col, Row], black)
  else
   begin
    FillRect(Grid[Col, Row], white);
    FrameRect(Grid[Col, Row]);
   end
end;
```

To terminate the program, a larger rectangle is displayed below the grid. A click in this box will stop the program from accepting any new mouse input. The program then displays, in the Text window, the corresponding values in the array Mark.

```
for Row := 1 to 8 do
 begin
  for Col := 1 to 8 do
   Write(Mark[Col, Row], ' ');
  Writeln
 end;
```

Later we will discuss how this data can be put to further use. Here is the program all together.

```
program GridEdit;
 type
  GridSize = 1..8;
 var
  Grid : array[GridSize, GridSize] of Rect;
  Mark : array[GridSize, GridSize] of Boolean;
  R : Rect;
  X, Y : Integer;
  Row, Col : Integer;
  Pt : Point;
  StopRect : Rect;
 begin
  SetRect(StopRect, 150, 150, 180, 180);
  FrameRect(StopRect);
  for Col := 1 to 8 do
  for Row := 1 to 8 do
   begin
    Mark[Row, Col] := False;
    SetRect(Grid[Row, Col], 10 + Row * 10, 10 + Col * 10, 20 + Row * 10, 20 + Col * 10);
    FrameRect(Grid[Row, Col]);
   end;
 repeat
  repeat
  until Button;
  GetMouse(Pt.H, Pt.V);
  for Col := 1 to 8 do
   for Row := 1 to 8 do
   if PtInRect(Pt, Grid[Col, Row]) then
    begin
    Mark[Col, Row] := not (Mark[Col, Row]);
    if Mark[Col, Row] then
    FillRect(Grid[Col, Row], black)
   else
   begin
    FillRect(Grid[Col, Row], white);
    FrameRect(Grid[Col, Row]);
    end
   end;
 until ptInRect(Pt, StopRect);
 for Row := 1 to 8 do
 begin
  for Col := 1 to 8 do
   Write(Mark[Col, Row], ' ');
  Writeln
 end;
end.
```

Drawing Other Shapes

QuickDraw supports the drawing of other simple shapes in addition to rectangles, notably, ovals and round-cornered rectangles.

Ovals

Drawing ovals is simple once you understand rectangles. An oval is defined as the largest oval that will fit inside a given rectangle as shown in Figure 5.3.

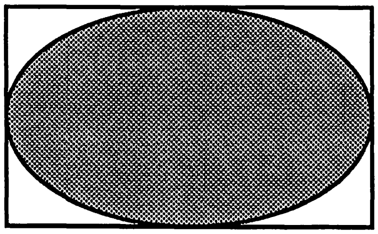


Figure 5.3 An oval inscribed inside a rectangle

To draw an oval, you define its enclosing rectangle and then use one of the oval-drawing routines.

```
procedure FrameOval(R : Rect)
```

draws the outline of the oval enclosed in the rectangle R.

```
procedure PaintOval(R : Rect)
```

paints an oval just inside the specified rectangle.

```
procedure InvertOval(R : Rect)
```

inverts the pixels in the oval enclosed by the rectangle R.

```
procedure FillOval(R : Rect; Pat : Pattern)
```

Fills the oval enclosed by the rectangle R with the pattern Pat.

In addition to these routines Macintosh Pascal includes two routines not normally part of QuickDraw.

```
procedure PaintCircle (X, Y, R : Integer);
```

paints a circle of radius R with its center as point (X, Y).

```
procedure InvertCircle (X, Y, R : Integer);
```

inverts the pixels in the circle of radius R whose center is at point (X,Y).

The rectangle manipulation routines can all be used with ovals since the defining structures of ovals are rectangles.

Round-Cornered Rectangles

A round-cornered rectangle (Figure 5.4) is a rectangle whose corners are rounded by an oval inscribed inside them that determines the radius of the curve.

There are four round-cornered rectangle drawing procedures analogous to the rectangle drawing procedures.

```
procedure FrameRoundRect(R : Rect; ovalWidth, ovalHeight : Integer)
```

draws the outline of the defined round-cornered rectangle.

```
procedure PaintRoundRect (R : Rect; ovalWidth, ovalHeight : Integer)
```

paints the defined round-cornered rectangle.

```
procedure InvertRoundRect (R : Rect; ovalWidth, ovalHeight : Integer )
```

inverts the pixels in the defined round-cornered rectangle.

```
procedure FillRoundRect (R : Rect; ovalWidth, ovalHeight : Integer; Pat : Pattern)
```

fills the defined round-cornered rectangle with the pattern Pat.

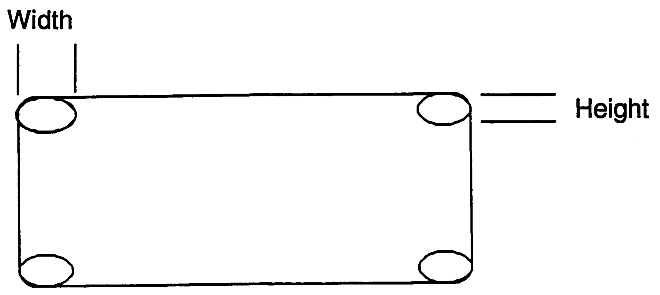


Figure 5.4 A round-cornered rectangle

Simple Animation Techniques

Using rectangles, animation effects can be created on the Drawing window. The basis of animation is the quick redrawing of the same shape in a slightly altered position.

As a building block of an animation program, let's look at a program that draws a series of rectangles across the screen.

```
program RectSeries;
 var
  R : Rect;
  K, Dh : Integer;
 begin
  Dh := 0;
  for K := 1 to 100 do
  begin
   SetRect(R, 10 + Dh, 10 + Dh, 30 + Dh, 40 + Dh);
   FrameRect(R);
   Dh := Dh + 10
  end
end.
```

The program RectSeries declares a rectangle, draws its outline, and then redefines the rectangle displaced by 10 points both to the right and down (Figure 5.5).

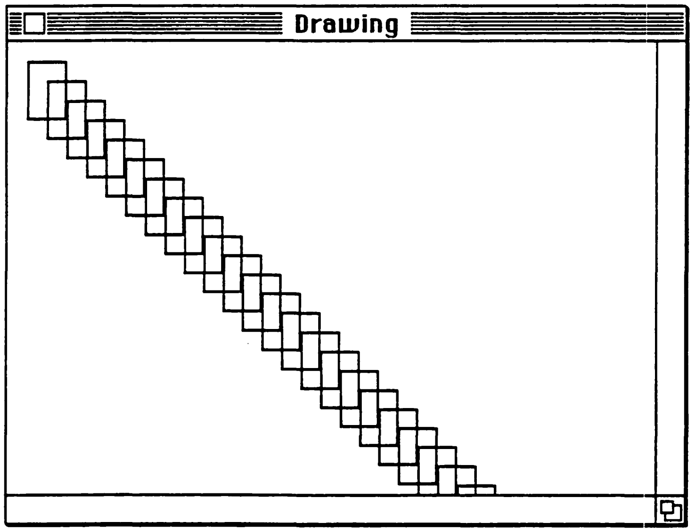


Figure 5.5 The rectangles drawn by SeriesRect

Since we wish to move a single rectangle across the screen rath
draw a group of rectangles, we must now erase the rectangle on the screen before drawing the displaced rectangle. This is exactly what is done in the enhanced program now called AnimaRect (Figure 5.6). It is also more effective if the rectangle is painted rather than just framed.

```
program AnimaRect;
 var
  R : Rect;
  K, Dh : Integer;
begin
 Dh := 0;
 for K := 1 to 100 do
  begin
   EraseRect(R);
   SetRect(R, 10 + Dh, 10 + Dh, 30 + Dh, 40 + Dh);
   PaintRect(R);
   Dh := Dh + 2 {Speed control}
  end
 end.
```

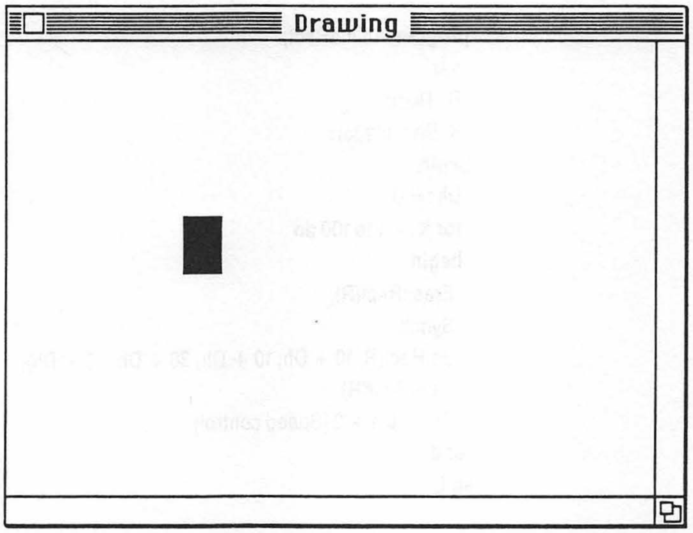


Figure 5.6 A snapshot from AnimaRect

If you run AnimaRect there are some interesting things to note. The speed of the rectangle is controlled by how much the rectangle is displaced each time. The larger the displacement, the faster the speed; but as the speed is picked up, the resolution of shape is decreased since there is less time to see the object. Alter the displacement value and watch the results. The flicker in the shape should also be noticeable. This flicker occurs because the drawing of the rectangle is not synchronized with the electron beam refreshing the Macintosh's screen. The human eye is not capable of sensing motion that occurs faster than 50 to 60 times a second and, coincidentally, the contents of the Macintosh's screen are redrawn at 60 times a second. Any drawing not coordinated with this cycle will appear to flicker, or it may display a scanning bar going from the top of the screen to the bottom. This is a particular problem with Macintosh Pascal because of the delay involved in interpreting the Pascal statements.

The Synch procedure is used to combat this problem. Synch holds control of the program and doesn't return it until the electron beam has reached the top of the screen and is about to redraw the entire screen. The procedure takes no parameters. Inserting the Synch procedure before the drawing routine will eliminate the flicker. This is not an exact science and the programmer may have to play around with different numbers of Synch calls in different positions in the program to get the desired effect.

```
program AnimaRect;
 var
  R : Rect;
  K, Dh : Integer;
begin
 Dh := 0;
 for K := 1 to 100 do
 begin
  EraseRect(R);
  Synch;
  SetRect(R, 10 + Dh, 10 + Dh, 30 + Dh, 40 + Dh);
  PaintRect(R);
  Dh := Dh + 2 {Speed control}
 end
end.
```

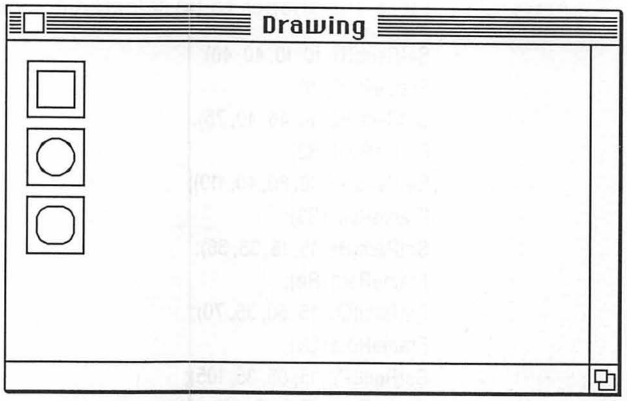


Figure 5.7 The PaulPaint screen

The PaulPaint Program

PaulPaint is a program that allows freehand drawing of rectangles, ovals, and round-cornered rectangles, loosely (very loosely) based on MacPaint. The program displays a choice of three shapes (rectangle, oval, and round-cornered rectangle) on the left side of the Drawing window (Figure 5.7). Clicking in any of the shapes selects that shape as the one to be drawn. A shape is drawn by clicking at the upper left-hand corner of the shape and then clicking again in the lower right-hand corner.

You will find the structure of the program to be quite interesting since the meaning of the mouse input is different, depending on the context that the program is in. In programs of higher complexity it is helpful to examine pseudocode for the program first.

```
Initialize menus
Repeat
  Repeat
  Until mouse button pressed
Test to see if cursor is in menu
If it wasn't then
  begin
  Repeat
    Animate rectangle
  Until mouse button pressed
DrawShape
Until Forever.
```

First, the shapes to be selected are defined and drawn.

```
SetRect(R1, 10, 10, 40, 40);
FrameRect(R1);
SetRect(R2, 10, 45, 40, 75);
FrameRect (R2);
SetRect(R3, 10, 80, 40, 110);
FrameRect (R3);
SetRect(Re, 15, 15, 35, 35);
FrameRect (Re);
SetRect(Ov, 15, 50, 35, 70);
FrameRect (Ov);
SetRect(Rc, 15, 85, 35, 105);
FrameRoundRect(Rc, 15, 15);
```

The program then loops until the mouse button is pressed, at which time the rectangles around the shapes are tested to see if the mouse click was inside of them.

```
repeat
until button;
GetMouse(Pt.H, Pt.V);
if PtinRect(Pt, R1) then
 begin
  DrawMode := Rct;
  Select := True
 end;
if PtinRect(Pt, R2) then
 begin
  DrawMode := Oval;
  Select := True
 end;
if PtinRect(Pt, R3) then
 begin
  DrawMode := Round;
  Select := True
 end;
```

If no shape was selected, then the click was intended to be the upper left-hand corner of the shape. It cannot be the lower point due to the context of the program. That point is saved, and then the program follows the movement of the mouse, drawing the rectangle defined by the current cursor position as its lower right-hand corner. Then it quickly erases it as the cursor is moved. This provides an animation effect. Once the mouse button is clicked, the final size of the rectangle is known and the selected shape is drawn inside it.

```
begin
 Top := Pt;
 repeat
  GetMouse(Bottom.H, Bottom.V);
  setRect(Draw, Top.H, Top.V, Bottom.H, Bottom.V);  FrameRect(Draw);
  EraseRect(Draw);
 until button;
 GetMouse(Bottom.H, Bottom.V);
 SetRect(Draw, Top.H, Top.V, Bottom.H, Bottom.V);
 case DrawMode of
  Rct :
   FrameRect(Draw);
  Oval :
   FrameOval(Draw);
  Round :
   FrameRoundRect(Draw, 9, 9);
 end; {Case}
end
```

Here is the entire program.

```
program PaulPaint;
 type
  Mode = (Rct, Oval, Round);
 var
  R1, R2, R3, Re, Ov, Rc : Rect;
  DrawMode : Mode;
  Pt : Point;
  Select : Boolean;
  Top, Bottom : Point;
  Draw : Rect;
```

```
begin
 Select := False;
 SetRect(R1, 10, 10, 40, 40);
 FrameRect(R1);
 SetRect(R2, 10, 45, 40, 75);
 FrameRect (R2);
 SetRect(R3, 10, 80, 40, 110);
 FrameRect (R3);
 SetRect(Re, 15, 15, 35, 35);
 FrameRect (Re);
 SetRect(Ov, 15, 50, 35, 70);
 FrameRect (Ov);
 SetRect(Rc, 15, 85, 35, 105);
 FrameRoundRect(Rc, 15, 15);
 repeat
  repeat
  until button;
  GetMouse(Pt.H, Pt.V);
  if PtinRect(Pt, R1) then
   begin
    DrawMode := Rct;
    Select := True
   end;
  if PtinRect(Pt, R2) then
   begin
    DrawMode := Oval;
    Select := True
   end;
  if PtinRect(Pt, R3) then
   begin
    DrawMode := Round;
    Select := True
   end;
   if Select then
    Select := False
   else
   begin
    Top := Pt;
```

```
        repeat
         GetMouse(Bottom.H, Bottom.V);
         SetRect(Draw, Top.H, Top.V, Bottom.H, Bottom.V);
         FrameRect(Draw);
         EraseRect(Draw);
        until button;
        GetMouse(Bottom.H, Bottom.V);
        SetRect(Draw, Top.H, Top.V, Bottom.H, Bottom.V);
        case DrawMode of
         Rct : FrameRect(Draw);
        Oval :
         FrameOval(Draw);
        Round :
         FrameRoundRect(Draw, 9, 9);
        end; {Case}
       end
      until false
     end.
```

PaulPaint Revisited

A second way of writing the PaulPaint program is to employ event handling. The advantage is improved response to mouse input. There is more direct handling of mouse down events with GetNextEvent rather than the Button function, which checks the Event Queue itself. The basic structure of the program changes little except substituting of GetNextEvent calls for the Button and GetMouse functions. This results in improved response to the mouse clicks. The GetNextEvents are set up only to detect a mouse down event with an event mask of two. This is because pulling the mouse up events off the Event Queue will confuse the operation of a program that was originally written with the Button routine (which only detects mouse downs). When the GetNextEvent detects a mouse down, the mouse location is held in the Where field of the event record. One call to GetMouse is still used to animate the movement of the rectangle across the screen since no event occurs as the mouse moves. Another change in the program is the inclusion of a fourth rectangle called "Stop," used to stop the program since Macintosh Pascal's event handling is shut off by the use of GetNextEvent. A much maligned Goto statement is used to branch the program to the end.

```
program PaulPaintII;
 label
  99;
 type
  Mode = (Rct, Oval, Round);
 var
  R1, R2, R3, Re, Ov, Rc, Stop : rect;
  DrawMode : Mode;
  Pt : Point;
  Select : Boolean;
  Top, Bottom : Point;
  Draw : Rect;
  E : EventRecord;
begin
 Select := False;
 setRect(R1, 10, 10, 40, 40);
 frameRect(R1);
 setRect(R2, 10, 45, 40, 75);
 frameRect(R2);
 setRect(R3, 10, 80, 40, 110);
 frameRect(R3);
 SetRect(Stop, 10, 115, 40, 145);
 FrameRect(Stop);
 MoveTo(12, 135);
 DrawString('Stop');
 setRect(Re, 15, 15, 35, 35);
 frameRect(Re);
 setRect(Ov, 15, 50, 35, 70);
 frameOval(Ov);
 setRect(Rc, 15, 85, 35, 105);
 frameRoundRect(Rc, 15, 15);
 repeat
  repeat
  until GetNextEvent(2, E);
  GlobalToLocal(E.Where);
  Pt := E.Where;
  if PtinRect(Pt, R1) then
   begin
    DrawMode := Rct;
    Select := True
  end;
 if PtinRect(Pt, R2) then
  begin
   DrawMode := Oval;
   Select := True
  end;
```

```
if PtinRect(Pt, R3) then
 begin
  DrawMode := Round;
  Select := True
 end;
 if PtinRect(Pt, Stop) then
  goto 99;
 if Select then
  Select := False
 else
  begin
   Top := Pt;
 repeat
  GetMouse(Bottom.H, Bottom.V);
  setRect(Draw, Top.H, Top.V, Bottom.H, Bottom.V);
  FrameRect(Draw);
  EraseRect(Draw);
  until GetNextEvent(2, E);
  GlobalToLocal(E.Where);
  Bottom := E.Where;
  GetMouse(Bottom.H, Bottom.V);
  setRect(Draw, Top.H, Top.V, Bottom.H, Bottom.V);
  case DrawMode of
  Rct :
  FrameRect(Draw);
  Oval :
   FrameOval(Draw);
   Round :
   FrameRoundRect(Draw, 9, 9);
   otherwise
   ;
   end; {Case}
  end
 until false;
99 :
 end.
```

Text-Drawing Routines

QuickDraw has a set of routines that write text data to the GrafPort. It is actually through these routines that *all* text output is done on the Macintosh, but this is transparent to the Macintosh Pascal user in every situation except when sending text information to the Drawing window.

The two basic text-drawing routines are DrawChar and DrawString. DrawChar draws one character at the current Pen location and DrawString draws a string at the current Pen location.

```
procedure DrawChar (Ch : Char);
procedure DrawString ( S : string);
```

Both of the procedures start their output at the current Pen location. If this is not where you want the output to go, you are not stuck. The Pen location can be altered with the MoveTo procedure

```
procedure MoveTo( h, v : Integer);
```

which moves the Pen location to point (h,v) without writing anything. For instance, in order to write at location (10,20), the following instructions can be used.

```
MoveTo(10, 20);
DrawString('Good Morning');
```

A second procedure Move is used to perform a relative move of the Pen location.

```
procedure Move( dh, dv : Integer);
```

Move adds dh to the current horizontal and dv to the current vertical Pen. If the current Pen location is (100,115), then the statement

```
Move( -10, 5);
```

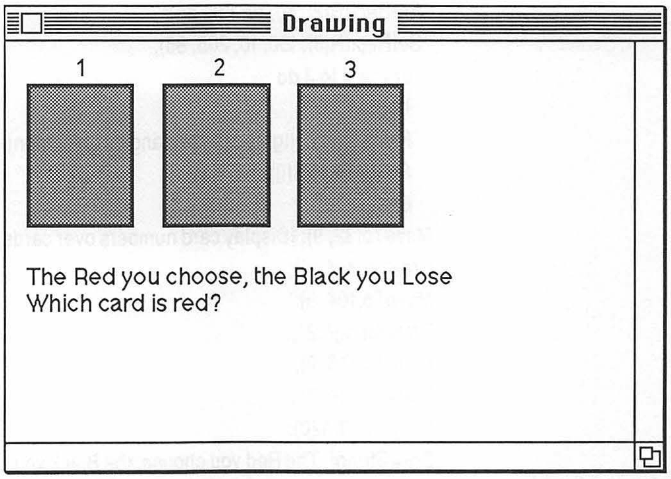
changes the current Pen location to (90, 120).

The Three-Card Monte Program

This program simulates the popular New York City street game Three-Card Monte. If you have never been to New York you may not be familiar with the game, which is designed to efficiently and quickly separate people from their money. Its premise is simple. The dealer has three cards, two of a black suit and one of a red suit. The player wages even money against the dealer that he or she can pick the red card after the three have been shuffled and placed on the table. Conceptually simple, the game (which the author has never played) is made more difficult with the use of a shill, who for five minutes wins handfuls of $20 bills from the seemingly "klutzy" dealer. But, when a pigeon who thinks he can make a quick buck puts up some money, a sleight-of-hand is used to take the player. I have seen dealers so good that when the shill bends the red card (with the dealer's back turned) in order to bring the "real" greed out in a player, the dealer does a sleight-of-hand during the shuffle that unbends the red and bends a black, leaving the player speechless and out $20.

Anyway, the program, which is not this malicious, displays three filled rectangles as cards and randomly picks one to be the red card (Figure 5.8). The player clicks on the card he or she chooses.

The technique of finding a mouse click in a rectangle should be old hat to you by now. What is interesting is the use of DrawString to prompt the player and display text inside of the rectangles.



```
The Red you choose, the Black you Lose
Which card is red?
```

Figure 5.8 The Three-Card Monte program

The pseudocode is:

```
Draw cards
Repeat
Until mouse button is pressed
Randomly pick red card
Find cursor position
Was selected card red card
Flip cards
```

```
program ThreeCardMonte;
 type
  Color = (red, black);
  CardRange = 1..3;
  NumOfCards = 3;
 var
  Card : array[ CardRange ] of Color;
  R : array[ CardRange ] of Rect;
  CardNum, RedCard, I : Integer;
  ChIn : Char;
  Pt : Point;
 begin
  SetRect(R[1], 10, 10, 65, 85); {set up rectangles}
  SetRect(R[2], 80, 10, 135, 85);
  SetRect(R[3], 150, 10, 205, 85);
  for I := 1 to 3 do
   begin
   FillRect(R[I], ltgray); {fill rectangles on screen}
   FrameRect(R[I]);
   end;
  MoveTo(32, 9); {Display card numbers over cards}
  DrawString('1');
  MoveTo(104, 9);
  DrawString('2');
  MoveTo(176, 9);
  DrawString('3');
  MoveTo(10, 120);
  DrawString('The Red you choose, the Black you Lose');
  MoveTo(10, 135);
  DrawString('Which card is red?'); {prompt the user}
```

```
repeat
until button; {wait for a mouse click}
GetMouse(Pt.h, Pt.v);
for I := 1 to NumOfCards do
 if PtInRect(Pt, R[I]) then {find clicked card}
  CardNum := I;
 for I := 1 to NumOfCards do
  Card[I] := Black;
 RedCard := TickCount mod 3 + 1; {now pick red card}
 Card[RedCard] := Red;
 for I := 1 to NumOfCards do {turn cards over}
  begin
   EraseRect(R[I]);
   FrameRect(R[I]);
   if Card[I] = Black then
    begin {display cards}
    Moveto(15 + 70 * (I - 1), 50);
    DrawString('BLACK')
    end
  else
    begin
    Moveto(15 + 70 * (I - 1), 50);
    DrawString('RED')
    end
 end;
MoveTo(20, 150);
if CardNum = RedCard then {congrats to the winner}
 DrawString('A WINNER')
else {taunt loser}
 DrawString('A LOSER, I LIKE LOSERS')
end.
```

Now that we have explored both event handling and a subsection of QuickDraw, we can combine the two to simulate Macintosh Toolbox pushbuttons and radio buttons.

Simulating Pushbuttons

Pushbuttons are a familiar part of the Macintosh User Interface. Unfortunately, Macintosh Pascal does not allow direct access to the use of pushbuttons, but it is not difficult to simulate them with QuickDraw routines.

A pushbutton is essentially a round-cornered rectangle with the name of the button displayed inside. When the mouse is clicked inside the button, it is inverted until the button is released.

As an example, the pushbuttons named Start and Stop (Figure 5.9) will be used.

Essentially, each pushbutton is a round-cornered rectangle. The Quick-Draw routines will be used to define the rectangle, draw the round-cornered rectangle and place the strings inside them. This is quite straightforward programming.

```
var
 Start, Stop : Rect;
  .
  .
 SetRect(Start, 20, 70, 80, 90);
 SetRect(Stop, 90, 70, 150, 90);
 FrameRoundRect(start, 10, 10);
 MoveTo(33, 85);
 DrawString('Start');
 FrameRoundRect(stop, 10, 10);
 MoveTo(103, 85);
 DrawString('Stop');
```

Figure 5.9 The example pushbuttons

To detect a mouse click inside of one of the buttons, a Repeat. . Until Button loop could be used. This, however, is not the most desirable way to operate since as the program is waiting inside the loop, no other operations can take place. A more preferable way to operate is to use an event loop. The loop will wait for a mouse down event in one of the buttons, but the loop is also capable of processing other tasks or events that occur. The form of the loop is:

```
repeat
 if GetNextEvent(2, E) then...
until False;
```

The simplest construction of the loop will detect a mouse down event, find out if the mouse location was inside a button's rectangle, and then invert that button.

```
repeat
 if GetNextEvent(2, E) then
  begin
  GlobalToLocal(E.Where);
  if PtInRect(E.Where, Start) then
   InvertRoundRect(Start,10,10)
 else
  if PtInRect(E.Where, Stop) then
   InvertRoundRect(Stop,10,10)
until...
```

This program segment tests where the mouse down event occurred by checking both rectangles defining the buttons. Unlike GetMouse, the coordinates returned in the Where field of the event record are in global coordinates (where the top left of the screen is the origin) not the local coordinates of the Drawing window. Global coordinates can be converted to local coordinates with the GlobalToLocal procedure.

```
procedure GlobalToLocal(var Pt : Point);
```

GlobalToLocal takes the point passed as a parameter and converts it to the local coordinates of the current GrafPort (which in Macintosh Pascal is the Drawing window), returning the new point as a variable parameter.

Note

The original point is lost since it is replaced by the new local point.

Once the location of the mouse click is determined in local coordinates it is tested to see if it is contained in either button. If it is, the button is inverted with InvertRoundRect.

This does not completely mimic the action of a button as it is only inverted while the mouse button is held down with the cursor inside the button. When the mouse button is released the rectangle, is "uninverted."

This can be added to the example by using a second repeat loop which terminates when the mouse up event occurs.

```
if ptInRect(E.Where, Start) then
 begin
  InvertRoundRect(Start, 10, 10);
 repeat
 until GetNextEvent(4, E); {Waits for a mouse up}
 InvertRoundRect(Start, 10, 10);
```

The following program demonstrates the simulated pushbuttons. A timing loop is used to control the execution of the program when Macintosh Pascal's regular event handling is turned off by the calls to GetNextEvent. An enumerated type, called ButtonType, is declared to track the button that was last pressed.

```
program ButtonSim;
 type
  ButtonType = (StartButton, StopButton, None);
 var
  Start, Stop : Rect;
  E : EventRecord;
  Ct : Integer;
  Pressed : ButtonType;
 begin
  Pressed := None;
  SetRect(Start, 20, 70, 80, 90);
  SetRect(Stop, 90, 70, 150, 90);
  FrameRoundRect(start, 10, 10);
  TextFont(0);
  MoveTo(33, 85);
  DrawString('Start');
  FrameRoundRect(stop, 10, 10);
  MoveTo(103, 85);
  DrawString('Stop');
  Ct := 0;
```

```
    repeat
    Ct := Ct + 1;
    if GetNextEvent(2, E) then
     begin
     GlobalToLocal(E.Where);
     if ptInRect(E.Where, Start) then
      begin
        InvertRoundRect(Start, 10, 10);
        repeat
        until GetNextEvent(4, E);
        InvertRoundRect(Start, 10, 10);
        Pressed := StartButton;
      end;
   if ptInRect(E.Where, Stop) then
      begin
        InvertRoundRect(Stop, 10, 10);
        repeat
        until GetNextEvent(4, E);
        InvertRoundRect(Stop, 10, 10);
        Pressed := StopButton;
      end
   end   until Ct = 500;
  Writeln(Pressed);
 end.
```

The simulation of push buttons can be further expanded by including both active and inactive buttons. An inactive button is displayed slightly dimmed and no action takes places when the mouse is clicked inside it. Simulating inactive buttons requires both displaying them differently from active buttons and programming so that no action takes place when the mouse is clicked in them.

The first question to address is how to display a dimmed button. In actual buttons displayed by the Toolbox, the name of the button is written in gray, but there is no way to do this with QuickDraw. Any method that differentiates active from inactive buttons can be used. One possibility is to display the name of an active button in a different font than an inactive. The font drawn by QuickDraw can be controlled with the TextFont procedure.

```
procedure TextFont(Font : Integer)
```

The TextFont procedure sets the font (or typeface) to be used by QuickDraw in displaying text. The font is indicated by an integer. All the fonts are assigned a number in a sequence defined by Apple Computer. The system font (Chicago) is numbered 0 and the application font (almost always Geneva) is numbered 1. The entire sequence is:

systemfont = 0

applFont = 1

NewYork = 2

Geneva = 3

Monaco = 4

Venice = 5

London = 6

Athens = 7

SanFran = 8

Toronto = 9

Notice that Geneva actually has two font numbers 1 and 3.

We can write the ButtonSim program so that only one of the two buttons are active at a time. This makes sense since the buttons are titled Start and Stop, which implies that their actions are mutually exclusive. We can also tie to the buttons some action such as tracking how much time has elapsed between button clicks. This can be done with calls to the TickCount function.

The initial setting of the program is to have the Start button active and the Stop button inactive. Active button names will be displayed in the system font (Chicago) and inactive button names in Geneva. When the Start button is clicked, TickCount is called to mark the start time, and the Start button is made inactive and the Stop button made active. To change a button from active to inactive or the reverse, the button is erased and then redrawn. Thus the code to draw the button should be placed into a procedure to eliminate duplication. The procedure can also receive a parameter to indicate which font to use in drawing the name. An example of such a procedure is DrawStart, which erases and then draws the Start button.

```
procedure DrawStart (Fnt : Integer);
begin
 TextFont(Fnt);
 MoveTo(33, 85);
 EraseRoundRect(start, 10, 10);
 FrameRoundRect(start, 10, 10);
 DrawString('Start')
end;
```

In order to make an inactive button inactive a enumerated type called StatusType is declared.

```
type
 StatusType = (On, Off);
```

A variable of StatusType, Status, is used to track the current state of the program. When the value of Status is ON, the Start button is active. When Status is OFF, the Stop button is active. Before a button's action is performed, Status is checked; for instance,

```
if ptInRect(E.Where, Start) and (Status = On) then
```

When the active Stop button is pushed, TickCount is called again and the elapsed time is computed. The outermost Repeat loop is then terminated.

```
program ButtonSim2;
 type
  ButtonType = (StartButton, StopButton, None);
  StatusType = (On, Off);
 var
  Start, Stop : Rect;
  E : EventRecord;
  Ct : Integer;
  Pressed : ButtonType;
  Status : StatusType;
  StartTime, StopTime : LongInt;
 procedure DrawStart (Fnt : Integer);
 begin
  TextFont(Fnt);
  MoveTo(33, 85);
  EraseRoundRect(start, 10, 10);
  FrameRoundRect(start, 10, 10);
  DrawString('Start')
 end;
 procedure DrawStop (Fnt : Integer);
 begin
  TextFont(Fnt);
  EraseRoundRect(stop, 10, 10);
  FrameRoundRect(stop, 10, 10);
  MoveTo(103, 85);
  DrawString('Stop')
 end;
```

```
begin
 Pressed := None;
 Status := On;
 SetRect(Start, 20, 70, 80, 90);
 SetRect(Stop, 90, 70, 150, 90);
 DrawStart(0); {Chicago}
 DrawStop(1); {Geneva}
 Ct := 0;
 repeat
  Ct := Ct + 1;
  if GetNextEvent(2, E) then
   begin
    GlobalToLocal(E.Where);
     if ptInRect(E.Where, Start) and (Status = On) then
    begin
    InvertRoundRect(Start, 10, 10);
    repeat
    until GetNextEvent(4, E);
    InvertRoundRect(Start, 10, 10);
    GlobalToLocal(E.Where);
    Status := Off;
    Pressed := StartButton;
    DrawStop(0);
    DrawStart(1);
    StartTime := TickCount;
    end;
 if ptInRect(E.Where, Stop) and (Status = Off) then
  begin
   InvertRoundRect(Stop, 10, 10);
   repeat
   until GetNextEvent(4, E);
   InvertRoundRect(Stop, 10, 10);
   Pressed := StopButton;
   DrawStop(1);
   DrawStart(0);
   StopTime := TickCount;
   end {for}
  end;
 until Pressed = StopButton;
 Writeln((StopTime - StartTime) / 60 : 10 : 2, 'seconds');
end.
```

Radio Buttons

The second type of user interface controls we will simulate is radio buttons. Radio buttons are different than pushbuttons in that they maintain an OFF or ON setting. Typically, several radio buttons are grouped together, acting like the buttons on a car radio where only one can be pressed at a time. Pressing another button pops out the current one.

The six buttons displayed in Figure 5.10 are used as our example throughout this discussion. The simulation of radio buttons requires two steps: drawing the buttons and coordinating their action.

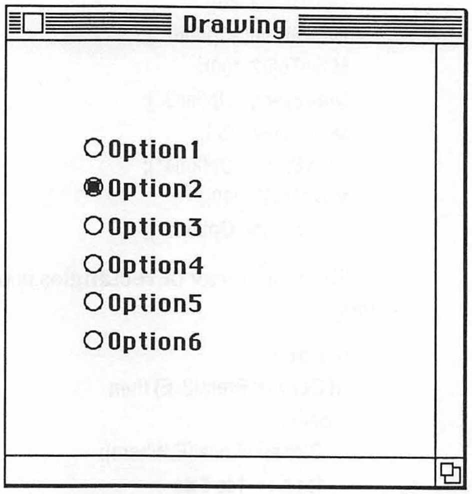


Figure 5.10 Radio buttons

The basic control structure for simulating a set of radio buttons is an array of type Rectangle.

```
var
  C : array[1..6] of Rectangle;
```

Each individual button is drawn in two parts: first the oval and then the button's name. The position of the first button in our group is (40,50) and the rectangle describing the oval is a 10-pixel square. The text is drawn next to the oval in the system font.

```
TextFont(0);
SetRect(C[1], 40, 50, 50, 60);
FrameOval(C[1]);
MoveTo(52, 60);
DrawString('Option1');
```

Each of the next buttons will be displayed 20 pixels down. A For loop could be used to provide more efficient code for setting and displaying the rectangles.

```
for K := 1 to 6 do
 begin
  SetRect(C[K], 40, 50 + (K - 1)*20, 50, 60 + (K - 1)*20);
  FrameOval(C[K])
 end;
```

The buttons' names can then be displayed with;

```
MoveTo(52, 60);
DrawString('Option1');
MoveTo(52, 80);
DrawString('Option2');
MoveTo(52, 100);
DrawString('Option3');
MoveTo(52, 120);
DrawString('Option4');
MoveTo(52, 140);
DrawString('Option5');
```

Since an array of rectangles is used, searching for a mouse click in an oval is easy.

```
repeat
 if GetNextEvent(2, E) then
  begin
   GlobalToLocal(E.Where);
   for K := 1 to 6 do
    if ptInRect(E.Where, C[K]) then
     Clicked := K
  end;
 until False;
```

Before a mouse click can be processed, some strategy must be devised to draw and erase the dot inside the button currently ON. A dot can be drawn inside an oval by defining a smaller oval inside and then inverting it. This can be done by shrinking the rectangle with InsetRect, inverting the oval, and then using Insetrect to stretch the oval back to original size.

```
InsetRect(C[K], 2, 2);
InvertOval(C[K]);
InsetRect(C[K], - 2, - 2);
```

If the number of the button currently set ON is held in Clicked, then the following loop will find a mouse click and then change the look of that button.

```
repeat
 if GetNextEvent(2, E) then
  begin
   GlobalToLocal(E.Where);
   for K := 1 to 6 do
    if ptInRect(E.Where, C[K]) then
     begin
      {Remove current dot}
      InsetRect(C[Clicked], 2, 2);
      InvertOval(C[Clicked]);
      InsetRect(C[Clicked], -2, -2);
      {Set new button}
      InsetRect(C[K], 2, 2);
      InvertOval(C[K]);
      InsetRect(C[K], -2, -2);
```

To use the simulated radio buttons in a program, one button should be initialized as being ON prior to the event loop. The following program fully demonstrates simulating radio buttons. For efficiency, a procedure is used to invert the dot.

```
procedure DoDot (K : Integer);
 begin
  InsetRect(C[K], 2, 2);
  InvertOval(C[K]);
  InsetRect(C[K], -2, -2);
end;
```

Here is the program in full.

```
program RadioButton;
 var
  C : array[1..6] of Rect;
  K, Clicked : Integer;
  E : EventRecord;
  Ct : Integer;
 procedure DoDot (K : Integer);
 begin
  InsetRect(C[K], 2, 2);
  InvertOval(C[K]);
  InsetRect(C[K], -2, -2);
 end;
```

```
begin
 Ct := 1;
 for K := 1 to 6 do
  begin
   SetRect(C[K], 40, 30 + K * 20, 50, 40 + K * 20);
   FrameOval(C[K])
  end;
 TextFont(0);
 MoveTo(52, 60);
 DrawString('Option1');
 MoveTo(52, 80);
 DrawString('Option2');
 MoveTo(52, 100);
 DrawString('Option3');
 MoveTo(52, 120);
 DrawString('Option4');
 MoveTo(52, 140);
 DrawString('Option5');
 MoveTo(52, 160);
 DrawString('Option6');
 Clicked := 1; {Initial setting}
 DoDot(Clicked);
 repeat
 if GetNextEvent(2, E) then
  begin
   GlobalToLocal(E.Where);
   for K := 1 to 6 do
    if ptInRect(E.Where, C[K]) then
     begin
      DoDot(Clicked);
      DoDot(K);
      Clicked := K;
     end
  end;
  Ct := Ct + 1
 until Ct = 500
end.
```

One last touch can be added to the program to enhance the simulation. In actual radio buttons a mouse click anywhere inside the oval or the button's label will select the button—unlike our simulation where the cursor must be in the oval itself. This feature can be added by defining a second set of rectangles which enclose both the rectangle defining the oval and the label (Figure 5.11). This array of rectangles (Called R) is then searched for the mouse click.

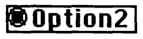


Figure 5.11 Rectangle enclosing oval and label

```
for K := 1 to 6 do
If ptInRect(E.Where, R[K]) then...
```

The size of these rectangles is defined by the length of the label. You can either guess at the length or use the StringWidth function.

```
function StringWidth(Str:string);
```

The StringWidth function returns the length in pixels of the given string, assuming the last type font to be selected. If the font or drawing characteristics are changed after calling StringWidth but before drawing, the results will not be accurate.

Since all the labels in the program are seven-characters long they are approximately the same length as 50 pixels in the system font. Note that since the system font is proportionately spaced, it may not always be the case that strings of the same number of characters have the same length in pixels. In a proportionately-spaced font, characters only use the amount of space needed instead of the even spacing for all characters found in monotonic fonts. However, in our case, six out of the seven letters are the same so all the lengths are almost all the same. The new rectangles can be defined with:

```
for K := 1 to 6 do {define second rectangles} SetRect(R[K], 40, 30 + K * 20, 102, 40 + K * 20);
```

In programs where the labels vary in length, StringWidth can be used to define the rectangle accurately. For example:

```
SetRect(R[1], 40, 50, 40 + StringWidth('Option1'), 60);
SetRect(R[2], 40, 70, 40 + StringWidth('Option2'), 80);
SetRect(R[3], 40, 90, 40 + StringWidth('Option1'), 100);
SetRect(R[4], 40, 110, 40 + StringWidth('Option1'), 120);
SetRect(R[5], 40, 130, 40 + StringWidth('Option1'), 140);
SetRect(R[6], 40, 150, 40 + StringWidth('Option1'), 160);
```

```
program RadioButton;
 var
  R, C : array[1..6] of Rect;
  K, Clicked : Integer;
  E : EventRecord;
  Ct : Integer;
 procedure DoDot (K : Integer);
 begin
  InsetRect(C[K], 2, 2);
  InvertOval(C[K]);
  InsetRect(C[K], -2, -2);
 end;
begin
  Ct := 1;
 for K := 1 to 6 do
  begin
   SetRect(C[K], 40, 30 + K * 20, 50, 40 + K * 20);
   FrameOval(C[K])
  end;
 for K := 1 to 6 do {define second rectangles}
  SetRect(R[K], 40, 30 + K * 20, 102, 40 + K * 20);
      TextFont(0);
 MoveTo(52, 60);
 DrawString('Option1');
 MoveTo(52, 80);
 DrawString('Option2');
 MoveTo(52, 100);
 DrawString('Option3');
 MoveTo(52, 120);
 DrawString('Option4');
 MoveTo(52, 140);
 DrawString('Option5');
 MoveTo(52, 160);
 DrawString('Option6');
 Clicked := 1; {Initial setting}
 DoDot(Clicked);
 repeat
  if GetNextEvent(2, E) then
   begin
    GlobalToLocal(E.Where);
    for K := 1 to 6 do
    if ptInRect(E.Where, R[K]) then
      begin
       DoDot(Clicked);
       DoDot(K);
       Clicked := K;
      end
  end;
 Ct := Ct + 1;
 until Ct = 500
end.
```

CHAPTER

6

The InLine Routines—Accessing the Toolbox

In earlier chapters we saw how extensions to the standard Pascal language allow Macintosh Pascal to use routines from the Macintosh User Interface Toolbox. The function GetNextEvent and the procedures SetText and GetDateTime are three examples of how Toolbox routines can be called as though they were Pascal intrinsics such as Writeln and Readln. Unfortunately, Macintosh Pascal provides for the use of only a small portion of the capability of the Toolbox by this method (although all the QuickDraw routines are supported). Most of the Toolbox features that every Macintosh user is familiar with, such as pulldown menus and windows, cannot be used.

To allow use of the majority of the Toolbox the designers of Macintosh Pascal provided the InLine routines. The InLine routines, three functions and one procedure, allow programmers to step out of Macintosh Pascal to include the Toolbox routines directly in their programs and thus develop "real" Macintosh applications with menus, windows, pushbuttons, and the like. When the Macintosh Pascal interpreter encounters an InLine statement, it branches directly to the ROM address of the Toolbox routine involved. This chapter explores four of the Toolbox's managers. Specifically, the Window Manager, Control Manager, Menu Manager, and TextEdit package. Some memory-management routines are thrown in for good measure.

The power of the InLines does not come without a price. Since the InLines are not standard Pascal extensions like the built-in routines that access QuickDraw, they provide absolutely none of Pascal's type checking or error trapping. This means Macintosh Pascal will not warn you if a routine is used improperly or if the parameters are of the wrong data type, and it will not provide a graceful exit after errors. Instead, when an error occurs, the entire Macintosh Pascal environment may "bomb," leaving you with no choice but to reboot the system, erasing your program. So it is imperative that when programming with InLines, you regularly save your program before you try running it. The warning is important enough to repeat.

SAVE YOUR PROGRAM OFTEN!

If you love your Macintosh and don't wish to harm it in a fit of anger after losing three hours of work, heed this advice.

There are three InLine functions because Toolbox functions return one of three types of values, Boolean(1 byte long), Integer(2 bytes), and LongInt(4 bytes). The InLine functions are:

BInLineF

used for Toolbox functions that return a one-byte value.

WInLineF

used for Toolbox functions that return a two-byte value.

LInLineF

used for Toolbox functions that return a four-byte value.

Similarly, any ToolBox procedure can be invoked with:

InLineP

used for all Toolbox procedures.

The InLines can be thought of as Macintosh Pascal's way of encapsulating calls to Toolbox routines. All of the InLine routines, when called, require at least one parameter to be passed to them, the ROM address of the routine, expressed in hexadecimal (Base 16). A hexadecimal address is denoted with a dollar sign $. A complete list of the routines covered in this book is in Appendix B. The routines are organized in the ROM with the help of a branch table containing a pointer to each of the routines. The addresses used to access a routine are actually the position of the routine's entry in the branch table. It is done this way to assure compatability after future changes in the ROM routines. Toolbox routines are accessed by referencing the appropriate ROM address, which is actually the entry in the branch table, which in turn branches execution to the actual routine. The last action taken by each routine is a mechanism to branch back to the Macintosh Pascal program. Many of the Toolbox routines also require parameters to be passed to them. This is done by listing these parameters after the address ROM of the routine in the InLine call.

Menus

The first examples of programming with InLines are menus. Pulldown menus are probably the most distinguishing feature of the Macintosh, but without using InLines there is no way to integrate menus into your Macintosh Pascal programs.

Pulldown menus are implemented via routines that are part of the Toolbox's Menu Manager. They are displayed on the screen in the menu bar which runs 20 pixels wide along the top of the Macintosh's screen. It is in the menu bar that the menu titles are displayed. The choices provided by a menu are known as menu items (Figure 6.1).

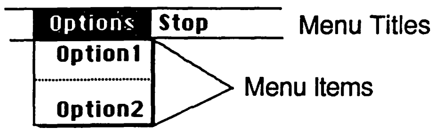


Figure 6.1 Components of a menu

Internally, the Menu Manager maintains linked lists which consist of handles to one or more menus. The current menu list contains handles to all the menus in the menu bar being displayed. The Toolbox routines that are part of the Menu Manager are:

Type	Name	Address	Returns
Function	GetMenuBar	$A93B	LongInt
Procedure	ClearMenuBar	$A934	
Procedure	SetMenuBar	$A93C	
Function	MenuSelect	$A93D	LongInt
Procedure	InsertMenu	$A935	
Procedure	DrawMenuBar	$A937	
Procedure	DisposeMenu	$A932	
Procedure	AppendMenu	$A933	
Function	NewMenu	$A934	LongInt
Procedure	HiLiteMenu	$A938	

The best menu routine to start exploring the Menu Manager is the function GetMenuBar.

```
function GetMenuBar : Handle      Address - $A93B
```

The purpose of GetMenuBar is to create a copy of the current menu list and return a reference to it in a handle. Since the size of a handle is 4 bytes (the same as a LongInt), GetMenuBar is invoked with LInLineF.

For example:

```
var
 SaveMenu : Handle;
   :
   :
begin
 SaveMenu := LInLineF($A93B);
```

This is our first demonstration of the use of an InLine. Notice that since GetMenuBar has no parameters itself, the only parameter used in the InLine call is the address of the GetMenuBar routine. This sequence of instructions will save a reference to the current menu list in the variable SaveMenu, which is declared as a Handle. Since Handle is not defined in Macintosh Pascal it is equated to a LongInt, also a 4-byte data type. SaveMenu could have also been declared to be a LongInt-byte data type. No data-type incompatability error would occur because Macintosh Pascal does no type checking anytime an InLine routine is used.

Instead of using the address of the Toolbox routine as the parameter for the InLine, it is better programming style to declare a constant containing the address of the routine and use the constant in the InLine.

```
const
 Handle = LongInt;
 GetMenuBar = $A93B;
  :
  :
begin
 SaveMenu := FInLine(GetMenuBar);
```

This programming style is far more readable and is highly preferred.

Now that we have saved a reference to the current menu list, it can be restored after we are done. Let's now clear the Macintosh Pascal menu from the screen.

```
procedure ClearMenuBar     Address - $A934
```

ClearMenuBar removes the current list from the Menu Manager so you can start fresh with a new menu bar. ClearMenuBar only removes the current menu from memory; it does not remove them from the screen. The next procedure we will see does that. Notice that ClearMenuBar is a procedure—not a function—and therefore returns no value.

```
procedure DrawMenuBar     Address - $A937
```

DrawMenuBar redraws the menu bar on the screen according to the current menu list. If the current menu list contains no entries then the menu bar is blanked out.

Combining the three menu routines the following program will save the Macintosh Pascal menu list and then clear the screen of all menus.

```
program MenuClear;
const
 GetMenuBar = $A93B;
 ClearMenuBar = $A934
 DrawMenuBar = $A932;
 Handle = LongInt;
var
 SaveMenu : Handle;
begin
 SaveMenu := FInLineF(GetMenuBar);
 InLineP(ClearMenuBar);
 InLineP(DrawMenuBar)
end.
```

If you run this program you will discover a small problem: after the program executes, you are left with no menus on the screen and thus no way to use Macintosh Pascal. Not being able to access the Macintosh Pascal menus might be desirable if you are designing an application and want to prevent the user from accessing the Macintosh Pascal environment; otherwise, you must reboot the computer by turning it off and then on. It is generally easier to install the plastic programmer's switch (included with the computer) on the left side of the Macintosh and use that to reboot the system. Instructions to do so are in the Macintosh owner's guide. I hope you heeded the warning to save your program before you run it. If you didn't, you will next time. The next menu routine discussed can be used to restore the Macintosh Pascal menus which were saved in the variable SaveMenu.

```
procedure SetMenuBar(menuList : Handle)      Address - $A93C
```

SetMenuBar is the first routine that takes a parameter, a Handle to a menu list. In our example, this is held in the variable SaveMenu. This routine replaces the current menu list with the one pointed to by the parameter menuList. The parameter is listed and separated by a comma after the routine address.

```
InLineP(SetMenuBar, SaveMenu);
```

This routine does not draw the new menu bar on the screen; another routine, DrawMenu, is used for this purpose. The following program expands the previous one by waiting and then restoring the Macintosh Pascal menu bar.

```
program MenuClear;
const
 GetMenuBar = $A93B;
 ClearMenuBar = $A934;
 DrawMenuBar = $A932;
 Handle = LongInt;
var
 K : Integer;
 SaveMenu : Handle;
begin
 SaveMenu := FInLineF(GetMenuBar);
 InLineP(ClearMenuBar);
 InLineP(DrawMenuBar);
 for K := 1 to 1000 do; {Wait loop}
   InLineP(SetMenuBar, SaveMenu);
 InLineP(DrawMenuBar)
end.
```

We have now seen the mechanism to save the Macintosh Pascal menu list, erase it, and then restore it to the screen. Now we are ready to build our own menus. The two menus in Figure 6.2 will serve as our example.

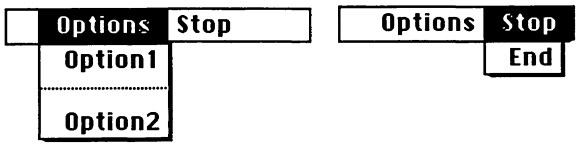


Figure 6.2 Sample menus

The function NewMenu creates a new, empty menu that is not installed in the menu list.

function NewMenu(menuID : Integer; menuTitle : **string**) : MenuHandle   Address - $A931

The parameter menuID is an integer value greater than zero used to identify the menu and determine where in the menu bar it will be displayed. Menu numbers 1 through 9 are reserved for use by Macintosh Pascal and should never be used by a program. To create our sample menus the following NewMenu calls are used.

```
var
 Menu1, Menu2 : LongInt;
  :
  :
 Menu1 := FInLineF(NewMenu,10, 'Options');
 Menu2 := FInLineF(NewMenu, 20, 'End');
```

The ID for the Options menu is 10 and for the End menu, 20. Both of the menus are empty at this point. Items are placed into a menu with the AppendMenu procedure.

procedure AppendMenu(theMenu : MenuHandle; item : **string**)   Address - $A933

AppendMenu adds items to the end of a menu. Continuing with our example, the items can be placed into Menu2 with:

```
InLineP(AppendMenu, Menu2, 'End');
```

Since Menu1 has only one item, only one call to AppendMenu is needed. Menu1 appears to have only two items but actually has three. Let's look at it again (Figure 6.3).

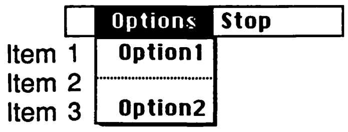


Figure 6.3 Menu items

The dotted line between Option1 and Option2 is actually considered to be a separate menu item that is permanently disabled; that is, it can never be selected by the user.

The dotted line between Option1 and Option2 is added to the menu by the second AppendMenu call which specifies a two-character sequence '(– ' used to specify that a dotted line be placed in that position as the item. The left parenthesis is the item disable operator.

```
InLineP(AppendMenu, Menu1, 'Option1');
InLineP(AppendMenu, Menu1, '(-');
InLineP(AppendMenu, Menu1, 'Option2');
```

Once items have been appended to it the menu is placed into the menu list with the InsertMenu routine.

```
procedure InsertMenu(theMenu : MenuHandle; beforeID : Integer)   Address - $A935
```

The parameters used are the menu handles of the menu and the ID number of the menu it is to be inserted before in the menu list. An ID of 0 means place the menu at the end of the menu list. Since this is where we want the two menus to be placed, the calls to InsertMenu are:

```
InLineP(InsertMenu, Menu1, 0);
InLineP(InsertMenu, Menu2, 0);
```

To draw the two new menus on the screen a call to DrawMenu is used.

```
InLineP(DrawMenu);
procedure DrawMenu      Address - $A932
```

DrawMenu displays on the screen the current menu list. No parameters other than the ROM address are passed because the current menu list is assumed.

A small program to demonstrate all of the menu concepts covered so far follows.

```
program MenuDemo;
 const
  GetMenuBar = $A93B;
  ClearMenuBar = $A934;
  DrawMenuBar = $A932;
  AppendMenu = $A933;
  NewMenu = $A931;
  SetMenuBar = $A93C;
  Handle = LongInt;
var
 K : Integer;
 SaveMenu : Handle;
```

```
begin
 SaveMenu := InLineF(GetMenuBar);
 InLineP(ClearMenuBar);
 InLineP(DrawMenuBar);
 InLineP(NewMenu,10, 'Options');
 InLineP(NewMenu, 20, 'End');
 InLineP(AppendMenu, Menu2, 'End');
 InLineP(AppendMenu, Menu1, 'Option1');
 InLineP(AppendMenu, Menu1, '(- ');
 InLineP(AppendMenu, Menu1, 'Option2');
 InLineP(InsertMenu, Menu1, 0);
 InLineP(InsertMenu, Menu2, 0);
 InLineP(AppendMenu, Menu2, 'End');
 repeat
 until WaitMouseUp;
 InLineP(SetMenuBar, SaveMenu);
 InLineP(DrawMenuBar)
end.
```

Remember to save the program before you run it. The repeat loop is used to maintain the menus on the screen until the mouse button is released.

If you run the MenuDemo program, you can see the installed menus on the screen, but the program has no ability to sense the mouse. To accomplish this, event handling must be combined with the MenuSelect function. When a mouse down event is detected it can be tested to see if it occurred in the menu bar. If so, the movement of the mouse can be tracked with the MenuSelect procedure. When the mouse button is released, the number of the menu item is returned.

```
function MenuSelect(startPt : Point) : LongInt     Address - $A93D
```

If MenuSelect is called after a mouse down event occurs in the menu bar, the routine will track the position of the cursor, highlighting menu items under the cursor. If the mouse button is released over an item, MenuSelect returns information about which item in which menu was selected. The information is passed in a LongInt whose high-order word contains the menuID and the low-order word contains the item number (Figure 6.4). If no item was selected, a zero is returned.

MenuID	Item Number
10	2

Figure 6.4 LongInt returned by MenuSelect

In order to dissect the LongInt returned, the Toolbox functions HiWord and LoWord, which are built into Macintosh Pascal, can be used. The function HiWord takes a LongInt and returns the value of the high-order word, LoWord returns the value of the low-order word.

```
procedure HiLiteMenu(menuID : Integer)     Address - $A93B
```

When MenuSelect returns a positive value it leaves the title of the selected menu highlighted. The procedure HiLiteMenu will reverse the highlighting if called with a parameter of 0. HiLiteMenu can be used to highlight any other menu in the menu bar by passing it the menuID of the menu title to be highlighted.

The following program demonstrates all the menu routines used together with an event loop. When a mouse down event occurs, MenuSelect is called and the values returned in the LongInt are displayed in the Text window. When an item is selected, the event loop is terminated and the Macintosh Pascal menu is restored.

```
program MenuDriver;
 const
 GetMenuBar = $A93B;
 ClearMenuBar = $A934;
 NewMenu = $A931;
 AppendMenu = $A933;
 InsertMenu = $A935;
 DrawMenuBar = $A937;
 DisposeMenu = $A932;
 SetMenuBar = $A93C;
 MenuSelect = $A93D;
 HiLiteMenu = $A938;
 type
 MenuHandle = LongInt;
 var
 OldMenu, MyMenu, MyMenu2 : MenuHandle;
 E : EventRecord;
 X, Y : Integer;
 Result : LongInt;
 P : Point;
```

```
begin
 InitCursor;
 OldMenu := LInLineF(GetMenuBar);
 InLineP(ClearMenuBar);
 MyMenu := LInLineF(NewMenu, 10, 'Options');
 InLineP(AppendMenu, MyMenu, 'Option1');
 InLineP(AppendMenu, MyMenu, '(-');
 InLineP(AppendMenu, MyMenu, 'Option2');
 InLineP(InsertMenu, MyMenu, 0);
 MyMenu2 := LInLineF(NewMenu, 12, 'Stop');
 InLineP(AppendMenu, MyMenu2, 'End');
 InLineP(InsertMenu, MyMenu2, 0);
 InLineP(DrawMenuBar);
 repeat
  if GetNextEvent(2, E) then
   begin
    Result := LInLineF(MenuSelect, E.where);
    X := HiWord(Result);
    Y := LoWord(Result);
    Writeln(X, Y, Result);
    if X > 0 then
     InLineP(HiLiteMenu, 0)
   end
  until X > 0;
  InLineP(ClearMenuBar);
  InLineP(DisposeMenu, MyMenu);
  InLineP(SetMenuBar, OldMenu);
  InLineP(DrawMenuBar)
 end.
```

Three additional menu routines that control menu appearance —DisableItem, EnableItem, and CheckItem—are of interest.

procedure DisableItem(theMenu : MenuHandle; item : Integer);   Address - $A932

DisableItem disables the specified item in the specified menu. Disabled menu items are displayed dimmed and are not highlighted when the cursor moves over them. Thus, they cannot be selected. If the value of the item parameter is 0 the entire menu is disabled.

procedure EnableItem (theMenu : MenuHandle; item : Integer);   Address - $A939

EnableItem reverses the effects of DisableItem, enabling a disabled menu item. If the item parameter is 0 the entire menu is enabled.

```
procedure CheckItem( theMenu : MenuHandle; item : Integer; checked : Boolean)
  Address - $A945
```

CheckItem places or removes a check mark to the left of the specified menu item.

Windows

The second part of the Macintosh User Interface we will examine is the Window Manager. The similarity of the windows displayed by different Macintosh programs can be attributed to the Window Manager, whose routines display and manipulate windows. The Macintosh Pascal program itself uses the Window Manager to maintain five windows: Program, Text, Drawing, Observe, and Instant.

To review the elements of a window let's look at Macintosh Pascal's Drawing window (Figure 6.5).

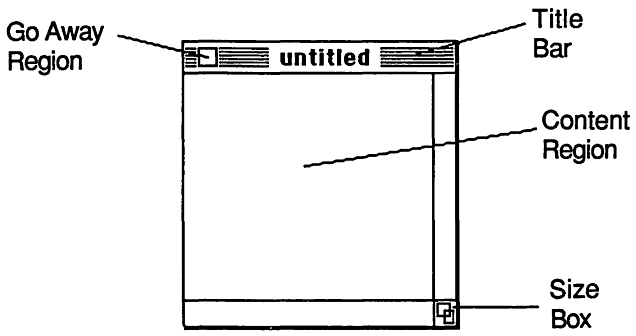


Figure 6.5 The elements of a window

The Window Manager allows six different styles of windows to be drawn. The Drawing window, pictured above, is known as a DocumentProc. The six styles are shown in Figure 6.6.

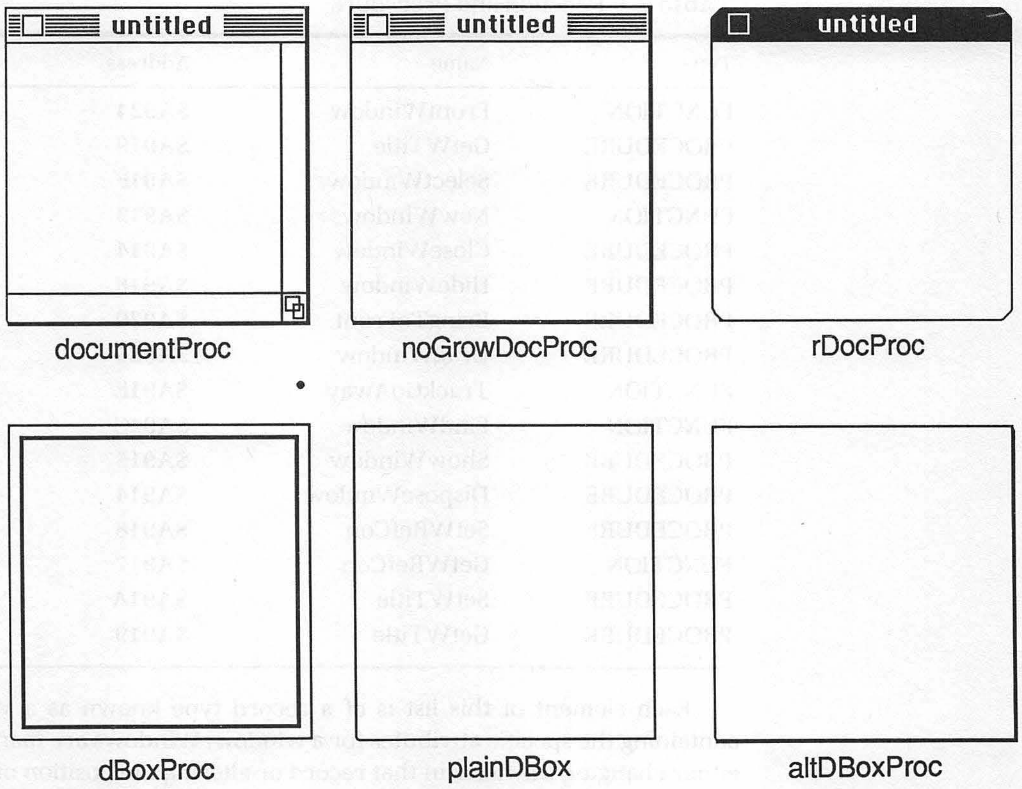


Figure 6.6 Window types

The Window Manager maintains a linked list of information about each window that is open. This list is what determines the position of the windows on the screen and their specific attributes. The Window Manager routines are used to manipulate the information stored in the list. The routines we will examine are listed in Table 6.1.

A good way to start exploring the Window Manager is by manipulating the windows that Macintosh Pascal already has displayed on the screen. The function FrontWindow will return a pointer to the window record of the active window on the screen.

```
function FrontWindow : WindowPtr     Address - $A904
```

Table 6.1 Function and Procedure

Type	Name	Address	Returns
FUNCTION	FrontWindow	$A924	LONGINT
PROCEDURE	GetWTitle	$A919	
PROCEDURE	SelectWindow	$A91F	
FUNCTION	NewWindow	$A913	LONGINT
PROCEDURE	CloseWindow	$A914	
PROCEDURE	HideWindow	$A916	
PROCEDURE	BringToFront	$A920	
PROCEDURE	DragWindow	$A925	
FUNCTION	TrackGoAway	$A91E	BOOLEAN
FUNCTION	FindWindow	$A92C	INTEGER
PROCEDURE	ShowWindow	$A915	
PROCEDURE	DisposeWindow	$A914	
PROCEDURE	SetWRefCon	$A918	
FUNCTION	GetWRefCon	$A917	LONGINT
PROCEDURE	SetWTitle	$A91A	
PROCEDURE	GetWTitle	$A919	

Each element of this list is of a record type known as a WindowRec, containing the specific attributes for a window. Windows are manipulated by either changing the fields in that record or altering the position of the record in the list. This is not normally done directly but rather through the Window Manager routines. The window records are referenced through a pointer to them. To accomplish this, the function looks at the window list and finds which window is active. It then returns a copy of the pointer to that window. The following is a simple program demonstrating FrontWindow.

```
program FindFrontWindow;
 type
  FrontWindow = $A924;
  WindowPtr = LongInt;
 var
  ActiveWindow : WindowPtr;
 begin
  ActiveWindow := FInLineF(FrontWindow);
  Writeln(ActiveWindow)
 end.
```

Run this program and then run it again after changing the active window. The value displayed in the Text window is the address held in the window pointer, expressed in a decimal that changes when the active window changes. Run the program twice without changing the active window and notice that the value of the pointer to the active window doesn't change. This demonstrates a subtle fact about the Window Manager and the Macintosh Pascal environment. The window list is independent of the program currently running under Macintosh Pascal, and Macintosh Pascal programs can manipulate the Macintosh Pascal windows just as if they were created by the running program itself.

The procedure GetWTitle takes a pointer to a window as its parameter. It finds the entry for that window in the window list and then returns the title of the window.

```
procedure GetWTitle(theWindow : WindowPtr; var Title : string)
```

This procedure operates in a slightly different manner than any other Toolbox routine we have seen in the way that a value is returned. GetWTitle returns a string containing the window's title by passing it back to the procedure as a variable parameter. In order for this mechanism to operate properly, the parameter that receives the value must be preceded with the pass-by reference operator, the at sign (@). The following program gets a pointer to the active window and then displays the window's title.

```
program FindFrontWindowTitle;
 type
  FrontWindow = $A924;
  GetWTitle = $A919;
  WindowPtr = LongInt;
 var
  ActiveWindow : WindowPtr;
  WinName : string;
 begin
  ActiveWindow := FrontWindow;
  GetWTitle(ActiveWindow, @WinName);
  Writeln(WinName)
 end.
```

A much more sophisticated way to interact with the Macintosh Pascal windows is the function FindWindow used to locate the window the cursor was positioned in when the mouse button was clicked.

```
    function FindWindow (thePt: Point; var whichWindow: WindowPtr):
Integer;
Address - $A92C
```

The function is passed the point where the mouse is clicked in global coordinates and returns a pointer to the proper window as a variable parameter. We can combine FindWindow with an event loop to track which window the mouse is clicked in. When using an event loop it is convenient to have a mechanism to exit the loop. In the following program the repeat loop is terminated when a variable, incremented in every iteration of the loop, reaches 50.

```
program ClickandTell;
 const
  GetWtitle = $A919;
  FindWindow = $A92C;
 var
  MyWindow : LongInt;
  B : Boolean;
  Str : string;
  E : EventRecord;
  Count, Code : Integer;
 begin
  Count := 0;
  repeat
   Count := Count + 1;
   if GetNextEvent(2, E) then
    begin
     Code := WInLineF(FindWindow, E.Where, @MyWindow);
      if MyWindow <> 0 then
       begin
        InLineP(GetWtitle, MyWindow, @Str);
        Writeln('-----', Str, '------')
       end;
  end;
  Writeln('Loop iteration', Count);
 until Count = 50;
end.
```

Position the Text window on the screen so that it is not obscured by the other windows and run the program. Clicking the mouse in a window will cause the name of the window to appear in the Text window.

Since FindWindow is a function, a value is returned by it. In the preceding program the value was assigned to the Integer variable Code. That value is useful in a way that will be discussed shortly.

Program ClickandTell is admittedly rather dull since the window where the mouse is clicked is not made active; that is, highlighted and brought to the front of the screen. This provides a clue to how much of the windowing functions are actually performed by the applications software itself rather than the Window Manager. When you are editing a program in Macintosh Pascal, the Macintosh Pascal interpreter handles the window functions, but when your program is running it must handle the windows itself. The procedure SelectWindow takes a window pointer and makes that window the active window.

```
procedure SelectWindow (theWindow : WindowPtr)      Address - $A91F
```

We can now add SelectWindow to our program and allow it to highlight and bring forward the window clicked on.

```
 program ClickandShowandTell;
  const
   GetWTitle = $A919;
   FindWindow = $A92C;
   SelectWindow = $A91F;
  var
   MyWindow : LongInt;
   B : Boolean;
   Str : string;
   E : EventRecord;
   Count, Code : Integer;
   begin
   Count := 0;
   repeat
   Count := Count + 1;
   if GetNextEvent(2, E) then
   begin
    Code := WInLineF(FindWindow, E.Where, @MyWindow);
     if MyWindow <> 0 then
      begin
       InLineP(SelectWindow,MyWindow);
       InLineP(GetWtitle, MyWindow, @Str);
       Writeln('------', Str, '------')
      end;
     end;
    Writeln('Loop iteration', Count);
   until Count = 50;
end.
```

More on FindWindow

As was mentioned previously, the procedure FindWindow does more than just return a pointer to the selected window. FindWindow also returns, as the value of the function, a code for where in a window the cursor was when the mouse was clicked. The possible values returned by FindWindow are:

6 — in the go-away box of the window selected (must be the active window)

5 — in the grow region of the window selected (must be the active window)

4 — in the draw region of the window

3 — in the content region of the window

2 — in the system window

1 — in the menu bar

0 — in the desktop (none of the above)

By testing the value returned by FindWindow, a program can determine the intention of the user by checking where the mouse was positioned then taking appropriate action. For instance, if the click is in the content region, the window can be highlighted; if the click is in the go-away box of the active window, the program can close the window with the procedure HideWindow.

The procedure HideWindow takes the window pointed to by the parameter and removes it from the screen.

```
procedure HideWindow (theWindow : WindowPtr)      Address - $A916
```

This procedure does not destroy references to the window and it can later be re-opened with the procedure ShowWindow.

The following program implements both the highlight and go-away features. It is substantially similar to the previous examples except there are changes inside the repeat loop. To increase efficiency, a case statement is used to test the code returned by FindWindow. If the click event was in the content window, the same action as before is taken; but if the click is in the go-away box, a call is made to HideWindow.

```
program WindowsGalore;
 const
  HideWindow = $A916;
  CloseWindow = $A92D;
  GetWtitle = $A919;
  SelectWindow = $A91F;
  FindWindow = $A92C;
  inContent = 3;
  InGoAway = 6;
```

```
var
  MyWindow : LongInt;
  B : Boolean;
  Str : String;
  E : EventRecord;
  Count, Code : Integer;
  begin
   Count := 0;
   repeat
    Count := Count + 1;
    if GetNextEvent(2, E) then
     begin
      Code := WInLineF(FindWindow, E.Where, @MyWindow);
      case Code of
       inContent :
       begin
        InLineP(GetWtitle, MyWindow, @str);
        InLineP(SelectWindow, MyWindow);
        Writeln(Code, Str);
       end;
     inGoAway :
      InLineP(HideWindow, MyWindow);
     end; {Case}
    end;
   Writeln('Loop iteration', Count);
  until Count = 50;
 end.
```

If you run the program you might notice that if you release the mouse button with the cursor outside of the go-away box, the window still closes. This does not conform to the standards of the Macintosh User Interface which specifies that a window is closed only if the mouse button is released inside the go-away box. The function TrackGoAway is used to track the cursor when a mouse down event occurs inside the go-away box by checking where the cursor is positioned when the mouse up event occurs.

function TrackGoAway (theWindow : WindowPtr; thePt : Point) : Boolean   Address - $A91E

The parameters to the function are the pointer to the window and the point on the screen where the mouse down event occurs, expressed in global coordinates. The Where field from the event record holds this information. The function returns True value if the mouse up occurs inside the go-away box and False if it occurs outside the go-away box. If the result is True, HideWindow can then be called.

```
program WindowsWithTrackGoAway;
 const
  NewWindow = $A913;
  FrontWindow = $A924;
  HideWindow = $A916;
  CloseWindow = $A92D;
  GetWtitle = $A919;
  SelectWindow = $A91F;
  DragWindow = $A925;
  FindWindow = $A92C;
  TrackGoAway = $A91E;
  inContent = 3;
  InGoAway = 6;
 var
  Go : Boolean;
  MyWindow : LongInt;
  B : Boolean;
  Str : string;
  E : EventRecord;
  Count, Code : Integer;
 begin
  Count := 0;
  repeat
   Count := Count + 1;
   if GetNextEvent(2, E) then
    begin
     Code := WInLineF(FindWindow, E.Where, @MyWindow);
     case Code of
      inContent :
      begin
       InLineP(GetWtitle, MyWindow, @Str);
       InLineP(SelectWindow, MyWindow);
       Writeln(Code, Str);
      end;
     InGoAway :
        begin
         Go := BlnLineF(TrackGoAway, MyWindow, E.Where);
         if Go then
          InLineP(HideWindow, MyWindow);
        end;
       otherwise
       ;
       end; {Case}
      end;
      Writeln('Loop iteration', Count);
     until Count = 50;
    end.
```

If you run the program you will notice that TrackGoAway also takes responsibility for highlighting the go-away box when the cursor is moved inside of it. Notice that TrackGoAway is the first Toolbox routine we have seen that returns a byte value and thus is invoked with BInLineF.

The DragWindow procedure is used to allow the window to be moved around the screen by clicking the mouse in the window's drag bar and then dragging the mouse to a new position.

```
procedure DragWindow (theWindow : WindowPtr; startPt : Point; boundsRect : Rect)
 Address - $A925
```

The parameters used are the pointer to the window to be moved, the starting point where the mouse down event occurred (from the event record), and a rectangle describing the area inside which the window can be dragged. If the mouse button is released with the window outside of the rectangle, DragWindow returns without moving the window or making it active. This is used to prevent a window from being placed over an area of the screen such as the menu bar.

```
program OldExamplewithMoveAdded;
 const
  NewWindow = $A913;
  FrontWindow = $A924;
  HideWindow = $A916;
  CloseWindow = $A92D;
  GetWtitle = $A919;
  SelectWindow = $A91F;
  DragWindow = $A925;
  FindWindow = $A92C;
  TrackGoAway = $A91E;
  inDrag = 4;
  inContent = 3;
  InGoAway = 6;
```

```
var
 Go : Boolean;
 DragRect : Rect;
 MyWindow : LongInt;
 B : Boolean;
 Str : string;
 E : EventRecord;
 Count, Code : Integer;
begin
 SetRect(DragRect, 5, 5, 500, 330);
 Count := 0;
 repeat
 Count := Count + 1;
 if GetNextEvent(2, E) then
   begin
    Code := WInLineF(FindWindow, E.Where, @MyWindow);
    case Code of
    inContent :
     begin
      InLineP(GetWtitle, MyWindow, @str);
      InLineP(SelectWindow, MyWindow);
      Writeln(Code, Str);
     end;
    InGoAway :
     begin
      Go := BInLineF(TrackGoAway, MyWindow, E.Where);
      if Go then
       InLineP(HideWindow, MyWindow);
     end;
    inDrag :
     begin
      InLineP(SelectWindow, MyWindow);
      Writeln('in drag routine');
      InLineP(DragWindow, MyWindow, E.Where, dragRect);
     end;
   otherwise
    ;
   end; {Case}
   end;
  Writeln('Loop iteration', Count);
 until Count = 50;
end.
```

This revised version of our program now supports moving windows around on the screen. The code returned if the mouse down event is in the title bar is 4. This has been assigned to the constant inDrag and it has been added to the case statement. If the mouse is clicked in the title bar a call is made to SelectWindow to make it the active window and then DragWindow is called.

Programming Example—Windows and Menus

We have seen how to program using two of the Toolbox's managers: menu and window. However, coordinating the implementation of Toolbox features becomes more complex as more sophistication is attempted. The following program is our first attempt to produce a true Macintosh-style application using both menus and windows. The program will display a menu giving the user the ability to close the active window on the screen or exit the program. A third option, open the formerly active window, is also displayed but disabled. If the window is closed then the close window option is disabled and the open window option is then enabled. To develop a program of this complexity, it is best to first pseudocode.

```
Initialize values
Set up new menu
Save old menu
Install new menu
Event loop
  GetNextEvent (looking for mouse down)
  Find out where the event was (FindWindow)
  case event location of
  inMenuBar :
  Find out which item was selected (MenuSelect, HiWord, LoWord)
  case which item selected of
    Open the window :
      Display the window (ShowWindow, SelectWindow)
      Disable 'Open window' option in menu (DisableItem)
      Enable 'Close window' option in menu (EnableItem)
  Exit :
      Restore the Macintosh Pascal Menu (SetMenuBar, DrawMenuBar)
      Set Flag to exit event loop
    Remove highlight from menu (HiLiteMenu)
  end {item selected case}
```

```
   inDrag :
      Drag the window around (DragWindow)
   Otherwise;
   Do nothing
until exit flag from event loop
```

Now the program itself.

```
program WindowsandMenus;
 const
  NewWindow = $A913; {Window routines}
  FrontWindow = $A924;
  HideWindow = $A916;
  CloseWindow = $A92D;
  GetWtitle = $A919;
  SelectWindow = $A91F;
  ShowWindow = $A915;
  DragWindow = $A925;
  FindWindow = $A92C;
  TrackGoAway = $A91E;
  GrowWindow = $A92B;
  SizeWindow = $A91D;
  GetMenuBar = $A93B; {Menu routines}
  SetMenuBar = $A93C;
  MenuSelect = $A93D;
  InsertMenu = $A935;
  DrawMenuBar = $A937;
  AppendMenu = $A933;
  NewMenu = $A931;
  ClearMenuBar = $A934;
  EnableItem = $A939;
  DisableItem = $A93A;
  HiLiteMenu = $A938;
  inMenuBar = 1; {FindWindow constants}
  inContent = 3;
  inDrag = 4;
  inGrow = 5;
  InGoAway = 6;
```

```
var
 Go : Boolean;
 SizeRect, DragRect : Rect;
 FrontSave, SaveMenu, MyMenu, GrowBy, MyWindow, Item : LongInt;
 Stop, B : Boolean;
 Str : string;
 E : EventRecord;
 H, W, Count, Code : Integer;
procedure SetUpMenu;
begin
 SaveMenu := LInLineF(GetMenuBar);
 InLineP(ClearMenuBar);
 MyMenu := LInLineF(NewMenu, 10, 'Window');
 InLineP(AppendMenu, MyMenu, 'Open it');
 InLineP(AppendMenu, MyMenu, 'Close it');
 InLineP(AppendMenu, MyMenu, '(-');
 InLineP(AppendMenu, MyMenu, 'Quit and Restore');
 InLineP(InsertMenu, MyMenu, 0);
 InLineP(DrawMenuBar);
 InLineP(DisableItem, MyMenu, 1); {Disable Open it}
end;{SetUpMenu}
procedure DoMenuCommand;
begin
 Item := LInLineF(MenuSelect, E.Where);
 if HiWord(Item) = 10 then
 case LoWord(item) of
  1 : {Open It}
  begin
   InLineP(ShowWindow, FrontSave);
   InLineP(SelectWindow, FrontSave);
   InLineP(DisableItem, MyMenu, 1);
   InLineP(EnableItem, MyMenu, 2)
  end;
  2 : {Close It}
  begin
   InLineP(HideWindow, FrontSave);
   InLineP(EnableItem, MyMenu, 1);
   InLineP(DisableItem, MyMenu, 2)
  end;
```

```
    4 : {Quit}
    begin
    InLineP(ClearMenuBar);
    InLineP(SetMenuBar, SaveMenu);
    InLineP(DrawMenuBar);
    Stop := True;
    end;
   end; {Case}
   InLineP(HiLiteMenu, 0);
  end;
  begin
   SetRect(DragRect, 5, 5, 500, 330);
   SizeRect.Top := 5;
   SizeRect.Left := 5;
   SizeRect.Bottom := 200;
   SizeRect.Right := 400;
   Stop := False;
   SetUpMenu;
   Count := 0;
   FrontSave := LInLineF(FrontWindow);
   repeat
    if GetNextEvent(2, E) then
    begin
     Code := WInLineF(FindWindow, E.Where, @MyWindow);
     case Code of
      inMenuBar :
      DoMenuCommand;
     inDrag :
     begin
      InLineP(SelectWindow, MyWindow);
      InLineP(DragWindow, MyWindow, E.Where, dragRect);
     end;
    otherwise
    ;
   end; {Case}
  end;
 Writeln('Loop running');
 until Stop = True;
end.
```

Programmer Defined Windows

The Window Manager routines we have been dealing with are all used to manipulate existing windows which, in our examples, have been the windows displayed by Macintosh Pascal. The NewWindow function is used to create new windows.

```
function NewWindow (wStorage : Ptr; boundsRect : Rect; title : string;
  visible : Boolean; procID : Integer; behind : WindowPtr;
  goAwayFlag : Boolean; refCon : LongInt) : WindowPtr;
Address - $A913
```

The NewWindow function provides a way for an application to create its own window, which can then be manipulated in the exact same manner as we have been manipulating the Macintosh Pascal windows. NewWindow is quite complex to use because some of the data types required are not predefined in Macintosh Pascal and because eight different parameters are used.

The significance of the parameters passed to NewWindow are:

wStorage. wStorage is a pointer to the place in memory where the window record will be stored by the Window Manager. An area large enough to hold the window record must be allocated by the program and a pointer to that area is passed as the parameter. Specifically, this is done by declaring a type called WindowRecord defined as:

```
WindowRecord = array[1 .. 78] of Integer;
```

This allocates 312 bytes of contiguous memory that is used by the Window Manager to store the descriptive information about the window created. Secondly, a pointer to a WindowRecord is also needed to be able to refer to it indirectly:

```
WindowPtr = ^WindowRecord;
```

boundRect. boundRect is a rectangle expressed in global coordinates. It is used to define the size and location of the window that will be created.

title. The title parameter is a string that contains the name that is displayed in the window's title bar.

visible. Visible is a boolean value that determines if the window is displayed on the screen; True to display the window, False not to.

procID. This is an integer value which describes the predefined window types to be created. The possible values are:

0 — a standard document window

1 — alert box or modal dialog box

2 — plain box

3 — plain box with shadow

4 — document window without size box

16 — round-cornered window

behind. This is a pointer to the window that the new window is displayed behind. If the new window is to be the front window then a – 1 is passed as a pointer using the built-in function Pointer(– 1).

goAwayFlag. This field is True if you want a goAway region in your window, False otherwise.

RefCon. This is a field used only by your program for whatever purpose you please. This is a convenient way to associate a value with a window.

A pointer is returned by NewWindow to the window being created. This can be used just as the pointers to the existing Macintosh Pascal windows were used in all the previous examples.

The use of NewWindow is best demonstrated in a program. The following program opens a window, displays a short message in it, and then closes it after the user hits the mouse button (giving you time to examine what you have created). Let's look at the NewWindow call used:

```
if LInLineF(NewWindow, TheWindow, BoundsRect, 'My Window', True, 0, Pointer(-1), False,
0 + 0) = Ord(TheWindow) then;
```

Notice that the function call is contained inside an If statement. This is one of the small programming "fudges" necessary when using InLines. The rationale for this is that as a function, NewWindow returns a value which is a pointer to the window record. This value is not needed since it is the same as the value sent in the wRecord parameter. (It is needed in a variant of this NewWindow call not possible from Macintosh Pascal.) But the value must be used in some way in order not to cause a syntax error generated by an illegal use of a function. One safe way of handling this is to compare the value to the parameter set, since they will be the same, and then take no action in the If statement. Another possible solution is to call NewWindow from inside a Write statement which will open the window and then print the address held in the pointer in the Text window. Imaginative programmers will develop other ways of doing this. Once the window is open, it must be set as the current GrafPort with the SetPort procedure in order for QuickDraw output to be sent to it rather than Macintosh Pascal's Drawing window. When the window is no longer needed, it must be closed, otherwise a system error will occur.

Also look at the RefCon parameter which is passed as a value of 0 + 0 instead of just 0. This is done because the Toolbox expects a LongInt value. Since the result of all arithmetic operations on integers returns a LongInt, the addition of the two zeros will produce the proper parameter. Passing just 0 will cause a system error to occur.

```
program BabysFirstWindow;
 const
  NewWindow = $A913;
  CloseWindow = $A92D;
  SetPort = $A873;
 type
  WindowRecord = array[1..78] of Integer;
  WindowPtr = ^WindowRecord;
 var
  WindowStorage : WindowRecord;
  TheWindow : WindowPtr;
  BoundsRect : Rect;
 begin
  SetRect(BoundsRect, 3, 40, 307, 239);
  TheWindow := @WindowStorage;
  if LInLineF(NewWindow, TheWindow, BoundsRect, 'My Window', True, 0, Pointer(-1),
  False, 0 + 0) = Ord(TheWindow) then
  ;
  InlineP(SetPort, TheWindow);
  Moveto(20, 40);
  TextFont(0);
  TextSize(24);
  DrawString('I did it' I did it!');
  repeat
  until button;
  InLineP(CloseWindow, TheWindow)
end.
```

Opening windows proves to be one of the trickiest Toolbox operations but also one of the most rewarding to the programmer. Remember to save your programs before running.

Miscellaneous Window Routines

Several other miscellaneous window manager procedures and functions are useful when programming with windows.

procedure ShowWindow(theWindow : WindowPtr); Address - $A915

The ShowWindow procedure makes visible the window pointed to by theWindow. The window is not made active and is not brought to the front of the screen.

procedure CloseWindow (theWindow : WindowPtr); Address - $A914

The CloseWindow procedure removes theWindow from the screen and then deletes its window record and recovers the memory space. If your application creates windows, be sure to dispose of them with CloseWindow before returning to the Macintosh Pascal environment.

procedure SetWRefCon (theWindow : WindowPtr; Data : LongInt); Address - $A918

The SetWRefCon procedure sets theWindow's RefCon field to the value held in Data. Remember to pass the Data parameter as a Long Integer.

procedure GetWRefCon (theWindow : WindowPtr) : LongInt; Address - $A917

The GetWRefCon procedure retrieves the value in theWindow's RefCon field.

procedure SetWTitle (theWindow : WindowPtr; Title : **string**); Address - $A91A

The SetWTitle procedure sets the title of theWindow to the string held in Title. The window is then redrawn.

procedure GetWTitle (theWindow : WindowPtr; **var** Title : **string**);
Address - $A919

The GetWTitle procedure returns the title of theWindow as the value of the Title parameter. Remember to use the pointer formation operator (@) in front of that parameter.

The Controls Manager

In Chapter 5, we saw how you can simulate Toolbox control by clever use of QuickDraw routines, but as that Motown song from the sixties goes, "There's nothing like the real thing baby." So, in this chapter we will explore using the Toolbox to produce three types of controls; radio buttons, check boxes, and pushbuttons (Figure 6.7).

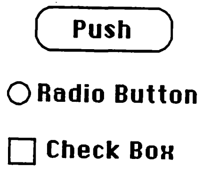


Figure 6.7 Radio button, check box and pushbutton

You are already familiar with radio buttons and pushbuttons from our simulation of them. Check boxes are meant to function like radio buttons in that they have two states, ON or OFF (checked or not checked), but are usually displayed alone rather than coordinated in groups.

Controls are handled by the Toolbox's Control Manager, which displays and manages them. All three types of controls are handled by the same set of Toolbox routines. They are all created by the NewControl function and a mouse click inside one of them, located with the FindControl function. Finally, the action of the particular type of control is handled with TrackControl.

The Control Manager maintains a record for each control, initialized with New Control. These control records are linked together on a linked list with a control list being specific to a particular window. There is no need to allocate space in your program for control records like there is for window records because they are set up by the Toolbox on the system heap and all reference to the record is done via a handle returned by NewWindow. The control record holds a link to both the next record in the linked list and to the window it belongs to. It also holds the attributes of that control. There are several attributes that a particular control may have associated with it. A control has a value that is its setting. For instance, in a radio button a value of one means that the button is ON and a value of zero means OFF. Along with its current value, a control record holds the maximum and minimum value for that control. Other attributes include whether or not the control is visible, its title, and if it is active or highlighted. Each type of control has a different appearance when it is highlighted. For instance, a pushbutton has a dark ring around it.

The Control Manager routines we will examine are:

Type	Name	Address	Returns
Function	NewControl	$A9C6	LONGINT
Function	FindControl	$A96C	INTEGER
Function	TrackControl	$A968	INTEGER
Procedure	SetCtrlValue	$A963	
Procedure	HiLiteControl	$A95D	
Procedure	DisposeControl	$A955	
Procedure	KillControls	$A956	
Procedure	HideControl	$A958	
Procedure	ShowControl	$A957	
Procedure	DrawControls	$A969	
Procedure	SizeControl	$A95C	
Function	GetCtlValue	$A960	INTEGER
Procedure	SetCtlMin	$A964	
Function	GetCtlMin	$A961	INTEGER
Procedure	SetCtlMax	$A965	
Function	GetCtlMax	$A962	INTEGER
Procedure	SetCRefCon	$A95B	
Function	GetCRefCon	$A95A	LONGINT

function NewControl(theWindow : WindowPtr; boundsRect : Rect; title : **string**; visible : Boolean; value : Integer; min, max : Integer; procID : Integer; refCon : LongInt);
Address - $A954

Where:

theWindow

is the pointer to the window that the control will be displayed in.

boundsRect

is a rectangle that defines the size and location of the control. Pushbuttons are drawn to fit the rectangle exactly, and there should be at least 20 points between the top and the bottom. For check boxes and radio buttons there should be at least 16 points between the top and bottom coordinates.

title

is the control's title which is displayed inside the control. Make sure that the title will fit inside the defining rectangle of the control. Use StringWidth to make sure.

```
visible
```

should be True if you want the Control Manager to draw the control when it is defined. It can always be drawn at a different time if you use False instead.

```
value
```

is the initial and current setting of the control.

```
min, max
```

are the possible range of settings. For controls that have ON-OFF states, such as radio buttons and check boxes, these values should be 0 and 1. For pushbuttons, which have no settings, 0 should be used for min, max, and value. If other values are used they are ignored anyway.

```
ProcID
```

is the way you tell the Control Manager which standard type of the control is being defined. The values are:

```
pushButton = 0;
checkBox = 1;
radioButton = 2;
```

Since Macintosh Pascal does not support controls these values *are not* defined in Macintosh Pascal.

```
RefCon
```

is a reference value that can be used by your program and has no effect on the control's appearance or operation. Note that since this parameter is a LongInt, you must either pass a variable declared as a LongInt or force a LongInt to be passed with the technique demonstrated previously.

Examples of the use of NewControl are:

For a pushbutton:

```
SetRect(PushRect, 20,20,50,70);
  Push := Pointer(LInLineF(NewControl, TheWindow, PushRect, 'Test1', True, 1, 0, 1, 1, 0 +
 0));
```

For a check box:

```
SetRect(CheckRect, 20,20,170,40);
 Check := Pointer(LInLineF(NewControl, TheWindow, CheckRect, 'Test1', True, 1, 0, 1, 1, 0
 + 0));
```

For a radio button:

```
SetRect(RadioRect, 20,20,170,40);
  Radio := Pointer(LInLineF(NewControl, TheWindow, RadioRect, 'Test1', True, 1, 0, 1, 2, 0 + 0));
```

Since radio buttons will be very often used in a group it is effective to use an array of rectangles and control handles as demonstrated in the following definitions.

```
CRect : array[1..6] of Rect;
Radio : array[1..6] of Handle;
Begin
  SetRect(CRect[1], 20, 20, 170, 40);
  SetRect(CRect[2], 20, 40, 170, 60);
  SetRect(CRect[3], 20, 60, 170, 80);
  SetRect(CRect[4], 20, 80, 170, 100);
  SetRect(CRect[5], 20, 100, 170, 120);
  SetRect(CRect[6], 20, 120, 170, 140);
  Radio[1] := Pointer(LInLineF(NewControl, TheWindow, CRect[1], 'Test1', True, 1, 0, 1, 2, 0 +
  0));
  Radio[2] := Pointer(LInLineF(NewControl, TheWindow, CRect[2], 'Test2', True, 0, 0, 1,
  Pointer, 0 + 0));
  Radio[3] := Pointer(LInLineF(NewControl, TheWindow, CRect[3], 'Test3', True, 0, 0, 1,
  Pointer, 0 + 0));
  Radio[4] := Pointer(LInLineF(NewControl, TheWindow, CRect[4], 'Test4', True, 0, 0, 1,
  Pointer, 0 + 0));
  Radio[5] := Pointer(LInLineF(NewControl, TheWindow, CRect[5], 'Test5', True, 0, 0, 1,
  Pointer, 0 + 0));
  Radio[6] := Pointer(LInLineF(NewControl, TheWindow, CRect[6], 'Test6', True, 0, 0, 1,
  Pointer, 0 + 0));
```

Essential to the use of controls is event handling. When a mouse down event is detected, it is the program's responsibility to call the FindControl function to determine if a control is involved, and if so, which one.

```
function FindControl (thePoint : Point; theWindow : WindowPtr; var whichControl :
ControlHandle) : Integer;   Address - $A96C
```

The FindControl function reports if a mouse down event occurred in any of the controls in a window. The parameters are:

thePoint

is the point where the mouse click occurred—usually obtained from the event record. This point must be in local coordinates so if the Where field from an event record is used, it must first be converted to local coordinates with GlobalToLocal.

theWindow

is the handle to the window where the controls are.

whichControl

is the handle to the control involved and is returned by the function. The address operator (@) *must be used* with this parameter.

The function returns two values. The first is an integer as the value of the function; it is either 0 if no control was selected or a part code if a control was selected. The second value returned is the whichControl parameter, which is passed to the program as a variable parameter. In order to receive variable parameters when using InLines, the address operator must be added to the actual parameter used in the function call. This is because when working outside the Pascal environment, variable parameters are passed as addresses rather than values.

The part code is a set value for each type of control used. If the mouse down was inside a pushbutton, it is 10; inside a radio button or check box, eleven. This is a way of determining what of category of control was selected.

Once a control handle is found, more work has to be performed by the program. This can be divided into two steps. First, the mouse action has to be tracked so that the control works as defined. For instance, when a mouse down occurs in a pushbutton, the button is inverted until it is released or the cursor is moved outside the button. With a radio button, the oval is highlighted until the button is released or moved outside the controls-defining rectangle. Unlike when we had to simulate this action ourselves, the Control Manager's TrackControl function handles this. The second thing that must be done is that the action tied to the button must be taken. When the Print pushbutton is pressed, printing should occur. Less obviously, with radio buttons the dot must be removed from the old button and placed in the new button. The Toolbox does not perform this function for your program because it cannot assume that all the radio buttons in a window are logically connected, and many times they may not be. This is done with the CtlValue procedure. Since an Integer is returned, FindControl is called with WInLineF.

function TrackControl (theControl : ControlHandle; startPt : Point; actionProc : ProcPtr) : Integer   Address - $A968

Where:

theControl

is the handle to the control where the mouse down event occurred. This will usually be obtained from FindControl.

startPoint

is the point where the mouse button was pressed, expressed in local coordinates. This will probably come from the Where field in the event record and if so, must first be converted to local coordinates with GlobalToLocal.

actionProc

is a pointer to a procedure that defines some action to be taken as long as the mouse button is held down. This is not used with simple controls, such as the ones being discussed, and as such is passed as **nil**.

The TrackControl function follows the movement of the mouse and responds in an appropriate manner for that type of control until the mouse button is released. If highlighting is appropriate, this is done by Track-Control. When the mouse button is released the function returns either the part code, if the cursor was still in the control it originally was in, or 0 otherwise. Since an Integer is returned, TrackControl is called with WInLineF.

The following several lines of code demonstrate how FindControl and TrackControl might be used together to determine which radio button was selected and then tracking it. First, FindControl determines if the mouse down event occurred inside a control. Both a part code and a control handle are returned. If the part code is greater than zero, TrackControl is called with the control handle returned. The value returned by Track-Control will tell us if any action has to be taken. If the value returned was 11, we then known that a button was selected. When used in a window with several types of controls, a Case statement can be used to check the part code returned.

```
if WinLineF(FindControl, E.Where, TheWindow, @whControl) > 0 then
begin
 InControl := WInLineF(TrackControl, whControl, E.Where, nil);
 if InControl > 0 then
```

The changing of the ON radio button is done with the SetCtrlValue procedure.

```
procedure SetCtrlValue (theControl : ControlHandle; theValue : Integer);   Address - $A963
```

The SetCtrlValue function changes the control's current value to theValue and then redraws the control to reflect the change. When a radio button or check boxes' value is 1, it appears ON; when it is 0, it appears OFF. The control value has no meaning to a pushbutton. Usually, when working with a group of radio buttons, one will be set OFF with another immediately set ON. We will maintain a variable that holds the number of the button ON. The following program displays a window with the six radio buttons that have been defined. It then manages any mouse clicks in a button. The program implements a timer to shut the program down after a predetermined value is hit.

```
program RadioRadio;
 const
  NewWindow = $A913;
  CloseWindow = $A92D;
  SetPort = $A873;
  NewControl = $A954;
  DrawControls = $A969;
  DisposeControl = $A955;
  FindControl = $A96C;
  TrackControl = $A968;
  SetCtlValue = $A963;
  RadioButProc = 2;
 type
  WindowRecord = array[1..78] of Integer;
  WindowPtr = ^WindowRecord;
  Ptr = ^LongInt;
  Handle = ^Ptr;
 var
  WindowStorage : WindowRecord;
  TheWindow : WindowPtr;
  ScrollBarRect, Button, BoundsRect : Rect;
  CRect : array[1..6] of Rect;
  Radio : array[1..6] of Handle;
  whControl : Handle;
  K, J : Integer;
  E : EventRecord;
  CtlOn, InControl, whichpart : Integer;
```

```
begin
 K := 1;
 CtlOn := 1;
 InitCursor;
 SetRect(BoundsRect, 3, 40, 407, 239);
 SetRect(ScrollBarRect, 20, 20, 170, 40);
 SetRect(CRect[1], 20, 20, 170, 40);
 SetRect(CRect[2], 20, 40, 170, 60);
 SetRect(CRect[3], 20, 60, 170, 80);
 SetRect(CRect[4], 20, 80, 170, 100);
 SetRect(CRect[5], 20, 100, 170, 120);
 SetRect(CRect[6], 20, 120, 170, 140);
 TheWindow := @WindowStorage;
 if LInLineF(NewWindow, TheWindow, BoundsRect, 'My Window', True, 0,
 Pointer(-1), False, 0 + 0) = Ord(TheWindow) then
 ;
 InLineP(SetPort, theWindow);
 Radio[1] := Pointer (LInLineF(NewControl, TheWindow, CRect[1], 'Test1',
 True, 1, 0, 1, RadioButProc, 0 + 0));
 Radio[2] := Pointer (LInLineF(NewControl, TheWindow, CRect[2], 'Test2',
 True, 0, 0, 1, RadioButProc, 0 + 0));
 Radio[3] := Pointer (LInLineF(NewControl, TheWindow, CRect[3], 'Test3',
 True, 0, 0, 1, RadioButProc, 0 + 0));
 Radio[4] := Pointer (LInLineF(NewControl, TheWindow, CRect[4], 'Test4',
 True, 0, 0, 1, RadioButProc, 0 + 0));
 Radio[5] := Pointer (LInLineF(NewControl, TheWindow, CRect[5], 'Test5',
 True, 0, 0, 1, RadioButProc, 0 + 0));
 Radio[6] := Pointer (LInLineF(NewControl, TheWindow, CRect[6], 'Test6',
 True, 0, 0, 1, RadioButProc, 0 + 0));
 repeat
  K := K + 1;
  if GetNextEvent(2, E) then
   begin
    GlobaltoLocal(E.Where);
     case E.What of
     1 :
     begin
       if WinLineF(FindControl, E.Where, TheWindow, @whControl) > 0 then
```

```
          begin
        InControl := (WInLineF(TrackControl, whControl, E.Where, nil));
         if InControl > 0 then
          for J := 1 to 6 do
           if whControl = Radio[J] then
           begin
             InLineP(SetCtlValue, Radio[CtlOn], 0);
             InLineP(SetCtlValue, whControl, 1);
           CtlOn := J
         end
        end
       end
      end
     end;
 until K = 600;
 Writeln('The set radio button is :', CtlOn);
 InLineP(CloseWindow, TheWindow)
end.
```

The program can be adapted to use a pushbutton to close the window rather than using the timer. This requires that the program be able to differentiate between the two types of controls in the window. This is done after the call to TrackControl by testing the part code returned and using that to determine which type of control was selected.

```
program RadioPush;
 const
  NewWindow = $A913;
  CloseWindow = $A92D;
  SetPort = $A873;
  NewControl = $A954;
  DrawControls = $A969;
  FindControl = $A96C;
  Trackcontrol = $A968;
  SetCtlValue = $A963;
  inButton = 11;
  inRadio = 10;
 type
  WindowRecord = array[1..78] of Integer;
  WindowPtr = ^WindowRecord;
  Ptr = ^LongInt;
  Handle = ^Ptr;
```

```
var
 WindowStorage : WindowRecord;
 TheWindow : WindowPtr;
 CRect : array[1..6] of Rect;
 Radio : array[1..6] of Handle;
 StopRect, Button : Rect;
 Done : Boolean;
 whControl, Stop : Handle;
 J : Integer;
 E : EventRecord;
 CtlOn, InControl, whichpart : Integer;
begin
 K := 1;
 CtlOn := 1;
 InitCursor;
 SetRect(CRect[1], 20, 20, 170, 40);
 SetRect(CRect[2], 20, 40, 170, 60);
 SetRect(CRect[3], 20, 60, 170, 80);
 SetRect(CRect[4], 20, 80, 170, 100);
 SetRect(CRect[5], 20, 100, 170, 120);
 SetRect(CRect[6], 20, 120, 170, 140);
 SetRect(StopRect, 20, 150, 50, 170);
 TheWindow := @WindowStorage;
 if LInLineF(NewWindow, TheWindow, BoundsRect, 'My Window', True, 0, Pointer(-1),
 False, 0 + 0) = ord(TheWindow) then
 ;
 InLineP(SetPort, theWindow);
  Radio[1] := Pointer(LInLineF(NewControl, TheWindow, CRect[1], 'Test1',
 True, 1, 0, 1, 2, 0 + 0));
  Radio[2] := Pointer (LInLineF(NewControl, TheWindow, CRect[2], 'Test2',
 True, 0, 0, 1, 2, 0 + 0));
  Radio[3] := Pointer (LInLineF(NewControl, TheWindow, CRect[3], 'Test3',
 True, 0, 0, 1, 2, 0 + 0));
  Radio[4] := Pointer (LInLineF(NewControl, TheWindow, CRect[4], 'Test4',
 True, 0, 0, 1, 2, 0 + 0));
  Radio[5] := Pointer (LInLineF(NewControl, TheWindow, CRect[5], 'Test5',
 True, 0, 0, 1, 2, 0 + 0));
  Radio[6] := Pointer (LInLineF(NewControl, TheWindow, CRect[6], 'Test6',
 True, 0, 0, 1, 2, 0 + 0));
  Stop := Pointer (LInLineF(NewControl, TheWindow, StopRect, 'Stop',
 True, 0, 0, 0, 0, 0 + 0));
  Done := False;
```

```
   repeat
    if GetNextEvent(2, E) then
    begin
     GlobaltoLocal(E.Where);
     case E.What of
     0:
     ; {Trap for when button outside control}
     1:
     if WinLineF(FindControl, E.Where, TheWindow, @whcontrol) > 0 then
      begin
      InControl := (WInLineF(TrackControl, whControl, E.Where, nil));
      case InControl of
      inButton :
       Done := True;
      inRadio :
       begin
        for J := 1 to 6 do
        if whControl = Radio[J] then
         begin
         InLineP(SetCtlValue, Radio[CtlOn], 0);
         InLineP(SetCtlValue, whControl, 1);
         CtlOn := J
        end {if}
      end {case 11}
    end {Case}
   end {case1}
  end {Big case}
  end; {Big IF}
 until Done = True;
 Writeln('The set radio button is :', CtlOn);
 InLineP(CloseWindow, TheWindow)
end.
```

Other Control Routines

procedure HiLiteControl (the Control : ControlHandle; HiLiteState : Integer); Address - $A95D

The HiliteControl procedure changes the way theControl is highlighted. The parameter hiliteState can be a value from 0 to 255. The default hiliteState of a control is 0, which displays the control as you expect to see it. A hiliteState value equal to the part code of a control will highlight it. For a pushbutton this will highlight the oval; for a radio button this will display the oval inverted, and there is little point doing this. A hiliteState value of 254 or 255 displays the control as inactive. The difference is that a hiliteState of 254 allows you to detect when the mouse button is pressed in the inactive control as opposed to not in any control.

procedure DisposeControl (theControl : ControlHandle); Address - $A955

The DisposeControl procedure removes theControl from the screen and from the window's control list and releases the memory occupied by the control record.

procedure KillControls (theWindow : WindowPtr); Address - $A956

The KillControls routine disposes of all the controls in theWindow as though DisposeControl had been called for each.

procedure HideControl (theControl : ControlHandle); Address - $A958

The HideControl procedure makes the control pointed to by the control handle invisible. It can later be redisplayed by calling ShowControl.

procedure ShowControl (theControl : ControlHandle); Address - $A957

The ShowControl procedure makes theControl visible and draws it on the screen.

procedure DrawControls (theWindow : WindowPtr); Address - $A969

The DrawControl procedure draws all the controls in the control list of theWindow.

procedure SizeControl (theControl : ControlHandle; w, h : Integer); Address - $A95C

The SizeControl procedure changes the size of a controls-defining rectangle. The bottom right corner of the rectangle is moved W pixels horizontally and H pixels vertically (to the right and down). Negative values move the other way. There is no effect on the upper left-hand corner of the rectangle, which is then redrawn on the screen.

function GetCtlValue (theControl : ControlHandle) : Integer; Address - $A960

The GetCtlValue function returns the current value of theControl. When radio buttons are used, this is an alternative way of detecting which button is set ON.

procedure SetCtlMin (theControl : ControlHandle; minValue : Integer); Address - $A964

The SetCtlMin procedure changes the minimum value setting of theControl to minValue. The control is then redrawn.

function GetCtlMin (theControl : ControlHandle) : Integer; Address - $A961

The GetCtlMin function returns the minimum value for theControl.

procedure SetCtlMin (theControl : ControlHandle; minValue : Integer); Address - $A964

The SetCtlMin procedure changes the maximum value of theControl to maxValue. The control is then redrawn.

function GetCtlMax (theControl : ControlHandle) : Integer; Address - $A962

The GetCtlMin function returns the maximum value for theControl.

procedure SetCRefCon (theControl : ControlHandle; data : LongInt); Address - $A95B

The SetCRefCon procedure sets the reference value of a control. This value is just for use by the program; the Toolbox has no interest in it. Note that the value used is a LongInt and must be passed as so.

function GetCRefCon (theControl : ControlHandle) : LongInt; Address - $A95A

The GetCtlMin function returns the reference value for theControl.

Text Editing

The last Toolbox Manager we will explore is the TextEdit package. The designers of the Macintosh realized that text editing was probably the most prevalent operation performed by the user, no matter what type of application program was being run. If the Macintosh was going to be easy to use, then a standardized text editing package most be included in the Toolbox. The Macintosh's TextEdit package is an easy-to-use, mouse-based editing system that supports text operation such as:

- inserting text in a line
- deleting text that is backspaced over
- selecting text by clicking and dragging with the mouse and double-clicking to select a word
- Inverse highlighting of the text selected
- word wrapping (preventing words from being split on two lines)
- cutting, copying, and pasting to and from the clipboard
- left and right justification of text and centering.

All this text-editing power is packaged in probably the easiest-to-use portion of the Toolbox. Although it is powerful, there are some features you may have seen in other programs that are not supported by TextEdit, such as full justification of text (even left and right margins), tabs, and the use of more than one font or style variation at a time.

Text Editing Summary

Type	Name	Address	Returns
Function	TENew	$A9D2	TEHandle
Procedure	TEKey	$A9DC	
Procedure	TEClick	$A9D4	
Procedure	TEDispose	$A9CD	
Function	GetCursor	$A9B9	CursHandle
Procedure	SetCursor	$A851	
Procedure	TEActivate	$A9D9	
Procedure	TEIdle	$A9DA	
Procedure	TECut	$A9D6	
Procedure	TECopy	$A9D5	
Procedure	TEPaste	$A9D8	
Procedure	TEDelete	$A9D7	
Procedure	TEDeactivate	$A9D9	
Procedure	TESetSelect	$A9D1	
Procedure	TESetJust	$A9DF	

The basic data structure used for text editing is a record of type TERec. Like window records, this type is *not* predeclared in Macintosh Pascal and has to be declared as a type in the program so a handle of that type can be declared. Unlike window records it is desirable to declare all the individual fields of a Text Edit record since you may want to access the values in the fields at some point.

The declaration of a TERec is:

```
type
 TERec = record
  DestRect, ViewRect, SelRect : Rect;
  LineHeight, FirstBL : Integer;
  SelPoint : LongInt;
  SelStart, SelEnd, Active : Integer;
  WorkBreak, ClikLoop, ClickTime : LongInt;
  ClickLoc : Integer;
  CaretTime : LongInt;
  CaretState, Just, TElength : Integer;
  HText : Handle;
  RecalBack, RecalLines, ClikStuff, CrOnly : Integer;
  TxFont, TxFace, TxMode, TxSize : Integer;
  InPort, HighHook : Ptr;
  CaretHook : Ptr;
  NLines : Integer;
  LineStarts : array[0..32000] of Integer;
end;
```

As you can see, the record type declaration has 31 separate fields. Most of them are of no consequence to the programmer and are documented in *Inside Macintosh*. The fields that are of interest to us are:

DestRec

and

ViewRec

are two rectangles that are used to indicate where the window the text is displayed and how it is mapped to the display. Both expressed in local coordinates, the ViewRec is the rectangle within which text is visible on the screen; the DestRec, a more obscure concept, is where the text is actually drawn by the Toolbox. If the ViewRec is smaller than the DestRec, the text is clipped before being displayed on the screen.

TxFont, TxFace, TxSize

and

TxMode

are the font number, the style characteristics, the font size, and the pen mode of the text drawn.

HText

is a handle to the text.

TELength

is the number of characters of text stored, starting at HText^^.

NLines

is the number of lines in the text

LineStarts

is an array containing the position of the first character in each line. It is actually a dynamic structure that only uses as much space as needed. It is only declared as large as it is to comply with Pascal's type checking.

Access to a Text Edit record is done via a handle—never directly.

```
type
 TEPtr = ^TERec;
 TEHandle = ^TEPtr;
```

Use of the TextEdit package is quite simple and, like the other Toolbox managers we have explored, event driven. When a mouse down event occurs in a ViewRec, a routine to track the mouse and select text is called. When a key down event occurs, the event record Message field is dissected to find the character and add it to the text. After every call to a TextEDit routine the text is redrawn in the ViewRect

```
function TENew(DestRect, ViewRect) : TEHandle;   Address - $A9D2
```

The TENew function allocates a new Text Edit record and returns a handle to it. This handle is assigned to a variable of type TEHandle. The parameters passed to TENew are the Destination and View rectangles for the text. Each rectangle is in that rectangle's local coordinates. The DestRect must be at least as wide as the first character to be drawn (about 20 pixels). The ViewRect must not be an empty rectangle. The default settings for a new Text Edit record is single spaced, left justified and an insertion point at position 0.

```
procedure TEKey(key : Char; hTE : TEHandle);   Address - $A9DC
```

The TEKey procedure is used to insert a character at the insertion point in a ViewRect or to replace the selected text if text has been selected (the next operation to be covered). The parameters are the character to be entered into the text and the TEHandle for the Text Edit record. The character is usually found in the low word of the Message field of the event record for a key down event. It is up to your program to filter out illegal characters (do input verification).

If the code for a backspace is passed to TEKey, the character to the left of the insertion point is removed.

```
procedure TEClick(startPt : Point; Extend : Boolean; hTE : TEHandle);   Address - $A9D4
```

The procedure TEClick is the real workhorse of the Text Edit package. The routine controls the selecting and highlighting of text selected with the mouse. TEClick should be called anytime a mouse down event occurs in the ViewRect pointed to by hTE. The procedure maintains control of the program until the mouse button is released, highlighting and selecting as the mouse is moved. The other parameter passed to TEClick is the point (in local coordinates) where the mouse down event occurred (found in the event record). The Extend parameter is used to indicate whether the Shift key was held down at the time of the click, meaning that the selected text should be added to that already selected. This returns a True, if the Shift key is down; False otherwise.

The following program uses these three routines to provide some rudimenatry text-editing functions. The program first creates a window and then an edit record in that window. A nested repeat loop is then used to wait for an event. If the event is a key down, a call is made to TEKey to add it to the edit record. If the event is a mouse down, the mouse location is checked to see if it is inside the view rectangle. If it is, TEClick is called to track the mouse and select the text.

```
program TextDemo;
 const
  TENew = $A9D2;
  TEKey = $A9DC;
  TEClick = $A9D4;
  NewWindow = $A913;
  CloseWindow = $A92D;
```

```
type
 Ptr = ^LongInt;
 Handle = ^Ptr;
 WindowRecord = array[1..78] of Integer;
 WindowPtr = ^WindowRecord;
 TERec = record
  DestRect, ViewRect, SelRect : Rect;
  LineHeight, FirstBL : Integer;
  SelPoint : LongInt;
  SelStart, SelEnd, Active : Integer;
  WorkBreak, ClikLoop, ClickTime : LongInt;
  ClickLoc : Integer;
  CaretTime : LongInt;
  CaretState, Just, TElength : Integer;
  HText : Handle;
  RecalBack, RecalLines, ClikStuff, CrOnly : Integer;
  TxFont, TxFace, TxMode, TxSize : Integer;
  InPort, HighHook : Ptr;
  CaretHook : Ptr;
  NLines : Integer;
  LineStarts : array[0..32000] of Integer;
 end;
 TEPtr = ^TERec;
 TEHandle = ^TEPtr;
 var
  WindowStorage : WindowRecord;
  TheWindow : WindowPtr;
  BoundsRect, DestRect, ViewRect, ScrollBarRect : Rect;
  ScrollBar : Handle;
  TEH : TEHandle; Event : EventRecord;
  Mess : Integer;
  Ch : Char;
  Mouse : Point;
 begin
  TheWindow := @WindowStorage; {Create window}
  SetRect(boundsRect, 10, 40, 200, 300);
  if LInLineF(NewWindow, TheWindow, BoundsRect, 'Untitled', True, 0,
  Pointer(-1), False, 0 + 0) = ord(TheWindow) then ;
  InLineP(SetPort, TheWindow);
  SetRect(DestRect, 30, 30, 180, 50); {Set up edit record}
  ViewRect := DestRect;
  FrameRect(ViewRect);
  TEH := Pointer(LInlineF(TENew, DestRect, ViewRect));
  FlushEvents(255, 0);
  InitCursor;
```

```
    repeat
     repeat {Wait for an event}
     until GetNextEvent(15, Event);
     if Event.What = 3 then {Its a key down}
       begin
        Mess := Bitand(255, Event.Message);
        Ch := Chr(Mess);
        InLineP(TeKey, Ch, TEH);
       end
    else
    if event.what = 1 then {Its a mouse down}
    begin
     GlobalToLocal(event.where);
     if PtInRect(Event.Where, ViewRect) then
       begin
        InlineP(TEClick, Event.Where, False, TEH);
       end
    end;
   until Ch = 'z';
   InLineP(TEDispose, TEH);
   InLineP(CloseWindow, TheWindow);
 end.
```

The last action of the program is to clean up by closing the window and disposing of the Text Edit record. This is done with the TEDispose procedure.

procedure TEDispose(hTE : TEHandle); Address - $A9CD

The TEDispose procedure releases the memory occupied by the Text Edit record pointed to by hTE. Only call this procedure when you are done with this information.

Not much more is needed in order to make this small program live up to standardized Macintosh text editing. Two features that have to be added are the blinking caret at the insertion point in the text line and changing the cursor from an arrow to the I-beam when it is moved into the ViewRec. Let's tackle the former first.

We know how to change the cursor from MacPascal's cross hair to the north-northwest arrow by calling the InitCursor routine(supported by Macintosh Pascal), but how can we change it to the I-beam? The resource section of the System File contains the basic cursor pattern used by most applications such as the arrow, the cross hair, and the I-beam. Any of these can be summoned with one of the Toolbox's utilities—the GetCursor function.

function GetCursor (cursorID : Integer) : CursHandle; Address - $A9B9

The GetCursor function returns a handle to the cursor having the given cursorID. The standard cursors are held in the System Resource file.

The cursorID's for the standard cursors are:

Cursor	ID
IBeam	1
cross	2
plus	3
watch	4

To use the I-beam cursor, we first get its handle with GetCursor as follows.

```
var
 IBeam : Handle;
  .
  .
  IBeam := Pointer(LInLineF(GetCursor,1));
```

The same technique can be used with any of the other standard cursors. Any time we wish to display the I-beam, a call to the QuickDraw procedure SetCursor is used.

```
procedure SetCursor (crsr : Cursor);  Address - $A851
```

The SetCursor procedure is used to display the cursor pointer to by crsr. The current cursor can be changed to the I-beam with

```
InLineP(SetCursor, IBeam^);
```

Notice that the parameter used is a pointer rather than a handle because SetCursor uses only one level of indirection. Of course, the cursor can be changed back to the north-northwest arrow with a call to InitCursor.

Now that we know how to set the cursor to the I-beam, it is necessary to be able to detect where the cursor is at any time, especially when it is in the ViewRec. This can be done by constantly calling GetMouse to return the cursor position and then immediately calling PtInRect to see if the cursor is in the ViewRec. This should be done at all time when the program is not processing an event, and it can be handled in the nested Repeat loop in our program, which waits for an event to occur.

```
repeat
 GetMouse(Mouse.H, Mouse.V);
 if PtInRect(Mouse, ViewRect) then
   InLineP(SetCursor, IBeam^)
 else
 InitCursor
until GetNextEvent(15, Event);
```

When the cursor is found to be in the ViewRect it set as the I-beam; otherwise, it is set back to the arrow.

The second feature we wished to add is to provide the blinking caret at the insertion point. This is also quite simple to do and is implemented by a combination of two routines.

procedure TEActivate (hTE : TEHandle) Address - $A9D9

The TEActivate procedure highlights the selection range in the given Text Edit record. If the selection range is an insertion point it displays the blinking caret. This procedure should be called every time the window containing the Edit record becomes active. If you are using only one window, this procedure is called once.

This procedure starts the caret blinking. In order to maintain it, it is necessary to call TEIdle.

procedure TEIdle (hTE : TEHandle); Address - $A9DA

The TEIdle procedure is called to keep the displayed caret at the insertion point blinking. It should be called as often as possible and at least once, each time through the main loop of the program, in order to provide constant blinking. No matter how many times the procedure is called the time between blinks will never be less than the minimum of 30 ticks. The actual time before blinks is set by the user in the Control Panel desk accessory, but calls to TEIdle are needed to implement it.

```
program TestDemo2;
 const
  TENew = $A9D2;
  TEIdle = $A9DA;
  TEActivate = $A9D8;
  TEKey = $A9DC;
  TEClick = $A9D4;
  NewWindow = $A913;
  CloseWindow = $A92D;
  SetPort = $A873;
  SetCursor = $A851;
  GetCursor = $A9B9;
 type
  Ptr = ^LongInt;
  Handle = ^Ptr;
  WindowRecord = array[1..78] of Integer;
  WindowPtr = ^WindowRecord;
  TERec = record
   DestRect, ViewRect, SelRect : Rect;
   LineHeight, FirstBL : Integer;
   SelPoint : LongInt; SelStart, SelEnd, Active : Integer;
   WorkBreak, ClikLoop, ClickTime : LongInt;
   ClickLoc : Integer;
   CaretTime : LongInt;
   CaretState, Just, TElength : Integer;
   HText : Handle;
   RecalBack, RecalLines, ClikStuff, CrOnly : Integer;
   TxFont, TxFace, TxMode, TxSize : Integer;
   InPort, HighHook : Ptr;
   CaretHook : Ptr;
   NLines : Integer;
   LineStarts : array[0..32000] of Integer;
  end;
 TEPtr = ^TERec;
 TEHandle = ^TEPtr;
var
 WindowStorage : WindowRecord;
 TheWindow : WindowPtr;
 BoundsRect, DestRect, ViewRect, ScrollBarRect : Rect;
 ScrollBar : Handle;
 TEH : TEHandle;
 Event : EventRecord;
 I, Mess : Integer;
 Ch : Char;
 IBeam : Handle;
 TopLine, ScrapSize : Integer;
 Mouse : Point;
```

```
begin
 TheWindow := @WindowStorage;
 SetRect(boundsRect, 10, 40, 200, 300);
 if LInLineF(NewWindow, TheWindow, BoundsRect, 'Untitled', True, 0, Pointer(-1), False, 0
+ 0) = ord(TheWindow) then
 ;
 InLineP(SetPort, TheWindow);
 SetRect(DestRect, 30, 30, 180, 50);
 ViewRect := DestRect;
 IBEam := Pointer(LInLineF(GetCursor, 1));
 TEH := Pointer(LInlineF(TENew, DestRect, ViewRect));
 FlushEvents(255, 0);
 InitCursor;
 repeat
  repeat
  GetMouse(Mouse.h, Mouse.v);
  if PtInRect(Mouse, ViewRect) then
   begin
   InLineP(TEActivate, TEH);
   InLineP(SetCursor, IBeam^)
   end
 else
 InitCursor;
 InLineP(TEIdle, TEH);
 until GetNextEvent(15, Event);
 if Event.What = 3 then
 begin
  Mess := BitAnd(255, Event.Message);
  Ch := Chr(Mess);
  InLineP(TeKey, Ch, TEH);
 end
else
if Event.What = 1 then
begin
 GlobalToLocal(Event.Where);
 if PtInRect(Event.Where, ViewRect) then
 begin
  InLineP(TEClick, Event.Where, False, TEH);
 end
end;
InLineP(TEIdle, TEH);
until Ch = '\';
InLineP(TEDispose, TEH);
InLineP(CloseWindow, TheWindow);
end.
```

At this point a fully functional text editing system has been implemented. However, several other features can be added to it. The first is being able to detect if the Shift key is down so that this information can be passed to TEClick.

SHIFT Click

If you refer back to the chapter on event handling you will see that information about the Modifier keys at the time of an event is passed in the Modifiers field of the event record. Specifically, if the Shift key was held down, the eighth bit is set. This value can be extracted from the field by performing a BitAnd using the field with 512 (2^8) as the mask.

```
BitAnd(Event.Modifiers, 512)
```

The value is returned and it can then be compared to 512 to see if that bit was set.

```
if BitAnd(Event.Modifiers, 512) = 512 then
```

To use this value we don't need the aid of an If statement because the parameter passed is a Boolean, and the result of any comparison is a Boolean. This can be placed directly into the expression.

```
if Event.What = 1 then
begin
 GlobalToLocal(Event.Where);
 if PtInRect(Event.Where, ViewRect) then
  begin
   InLineP(TEClick, Event.Where, BitAnd(Event.Modifiers, 512) = 512, TEH);
  end
end;
```

If this is to messy for you, the result of the comparison can be placed in a Boolean variable and that variable can be passed to TEClick.

```
if Event.What = 1 then
begin
 GlobalToLocal(Event.Where);
 if PtInRect(Event.Where, ViewRect) then
  begin
   Shift := (BitAnd(Event.Modifiers, 512) = 512)
   InLineP(TEClick, Event.Where, Shift, TEH);
  end
end;
```

Building the Strings

Next, it would be nice to be able to access the text that has been entered by the user. This text is placed by TextEdit into a packed array that has the field HText as a handle to it. The text is not placed into a string because of the 255-character size limitation of a string. To access the text, a technique to circumvent Pascal type checking must be used. Since HText is declared as a Handle, which is a pointer to a pointer to a LongInt, we cannot use it as though it pointed to an array. This would violate Pascal data type rules. We must first declare a handle to a packed array of characters and then assign the value of HText to that handle. This is demostrated with the following declarations.

```
type
 Chars = packed array[0..10] of Char;
 CharsPtr = ^Chars;
 CharsHandle = ^CharsPtr;
var
 TextData : CharsHandle;
```

If TextData is made equal to HText we can then work with it.

```
Data := TEH^^.HText;
```

Notice that we use TEH^^ since TEH is a handle to the Edit record and not the Edit record itself. Any operation can now be performed on the packed array TextData. An example of this is to display it in the **Text** window demonstrated below.

```
for I := 0 to TEH^^.TELength - 1 do
 Write(Data^^[I]);
```

This For loop uses as its upper boundary the length of the text that is stored in TEH^^TELength. We subtract one from it since we are starting with the zeroth element.

Input Verification

The TEKey procedure is nondiscriminating; it will place any character provided to it into the text, so in certain situations, input verification must be performed prior to calling TEKey. For instance, if only numeric data could be entered, a check to see if the character entered was a number is used. For example:

```
var
 NumSet : set of Integer;
   .
   .
 NumSet := ['0' .. '9'];
if Event.What = 3 then
```

```
begin
 Mess: = BitAnd(255, Event.Message);
 Ch : = Chr(Mess);
 if Ch in NumSet then
  InLineP(TeKey, Ch, TEH)
 else
  SysBeep(3);
end
```

The use of Sets to do input verification is preferred programming style. Notice that if an illegal value is entered, a call to SysBeep is made to alert the user. Other techniques could also be used, such as displaying the illegal text and then flashing it or displaying an error message.

Another aspect of input verification is limiting the number of characters that can be entered into an Edit record. This might be done in situations where there is a maximum number of characters allowed, such as for the name of a file. The current length of the text held in an Edit record is in the TELength field. This can be easily checked before a call is made to TEKey; for Instance, in this example the upper limit to the number of characters that can be entered is held in MaxChars.

```
if Event.What = 3 then {Handle the Key down}
 begin
  Mess : = BitAnd(255, Event.Message);
  Ch : = Chr(Mess);
  if HTE^^.TELength < MaxChars then
   InLineP(TeKey, Ch, TEH)
  else
   SysBeep(3);
 end
```

There is one hitch to this example. It does not allow the one character that should be allowed when the maximum number of characters has been reached, the Backspace. A check to see if the Backspace (ASCII 8) has been typed should be added to the If statement.

```
if Event.What = 3 then {Handle the Key down}
 begin
  Mess : = BitAnd(255, Event.Message);
  Ch : = Chr(Mess);
  if (HTE^^.TELength < MaxChars) or (Ch = Chr(20)) then
   InLineP(TeKey, Ch, TEH)
  else
   SysBeep(3);
 end
```

Other Text Edit Routines

The TextEdit package supports a large range of other text operations including the familar cut, copy, and paste.

procedure TECut(hTE : TEHandle); Address - $A9D6

The TECut procedure removes the selected text from the text specified by the parameter hTE and places it into the TextEdit scrap. The scrap is a temporary storage facility and is not the "desk scrap" used to support transfer of information between programs. If the selection range is the insertion point, the scrap is emptied. This function, along with copy and paste, is very often coordinated with a menu choice.

procedure TECopy (hTE : TEHandle); Address - $A9D5

The TECopy procedure copies the selected text in the text specified by hTE into the TextEdit scrap. erasing anything previously in it. If the selection range is the insertion point, the scrap is emptied.

procedure TEPaste (hTE : TEHandle); Address - $A9D8

The TEPaste procedure (Figure 6.8) replaces the selected text in the text specified by hTE with the contents of the TextEdit scrap and places the insertion point to the right of the inserted text. If the scrap is empty, the selected text is deleted. If the selection range is the insertion point, TEPaste just inserts the text at the insertion point.

procedure TEDelete (hTE : TEHandle); Address - $A9D7

The TEDelete procedure removes the selected text from the text specified by hTE. If the selection range is the insertion point, no action is taken.

procedure TEDeactivate (hTE : TEHandle); Address - $A9D9

The TEDActivate procedure unhighlights the selected part of the text specified by hTE. If the selection range is the insertion point, it is removed. TEDactivate should be called anytime that the window containing the Edit record is made inactive.

procedure TESetSelect (selStart, selEnd : LongInt; hTE : TEHandle); Address - $A9D1

The TESetSelect procedure is used to select and highlight text from inside a program. If you have ever seen a window appear with text already highlighted, you have seen the effect of TESetSelect. The range of text selected is identified as selStart, selEnd where the position of the first character in the text edit record is zero. The maximum range is from 0 to 32767. If selEnd is beyond the last character of the text, the position just past the last character is used.

procedure TESetJust (Just : Integer; hTE : TEHandle); Address - $A9DF

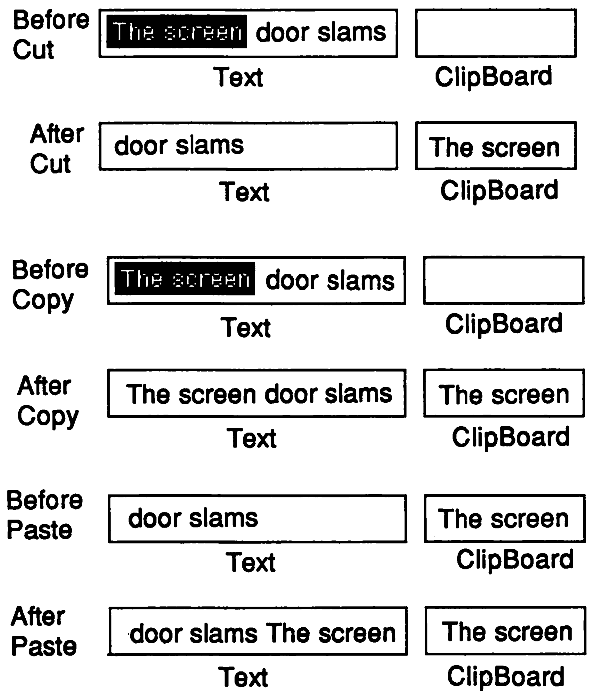


Figure 6.8 Cut, copy, and paste

The TESetJust procedure sets the justification of the text specified by hTE. Three types are supported, indicated by:

Left justified 0

Centered 1

Right justified 1

The default justification is left justified.

The Haiku Writer Revisited

Now that we have examined the four Toolbox features used to create standard Macintosh applications—windows, controls, menus and TextEdit—we can combine all four of these features into a single program cooordinating their actions. As an example, we can add a complete Macintosh interface to a program already developed, such as the Haiku program from Chapter 3. This way, we need not concentrate on the operations of the program, just the handling of the interface.

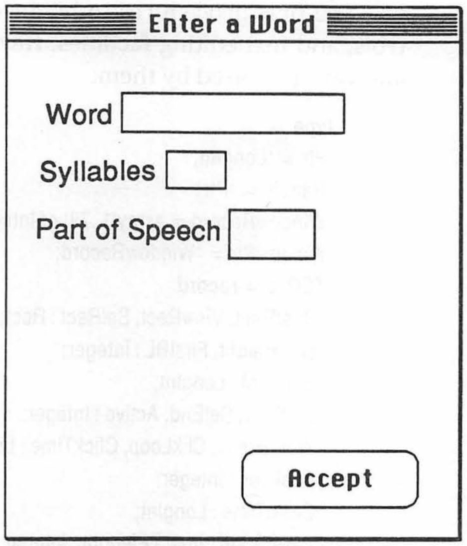


Figure 6.9 The Haiku user interface

The new version of the Haiku program (Figure 6.9) has a user-defined window for entering words, the number of syllables, and the parts of speech into text-editing fields. After a word is typed, it is accepted by clicking a push-button located in the window. A menu is used to permit the poet to clear the fields for further input, create a poem to be displayed in the Text window, or quit the program.

The program was adapted by breaking up its two major functions—adding words and writing a poem—into two procedures names AddWord and WritePoem. These procedures are tied to menu actions, as you will see later. Other procedures need to be added to create the Toolbox features and to manage them. The main program serves as the event handler, calling the appropriate routine.

One major change has been added to the program to speed the writing of a poem and to demonstrate another file technique. In this version, all the words are initially read into memory and stored in an array of type WordRec. To write a poem, the words are randomly chosen from the array rather than the file. This saves the file access time each time a word is randomly selected. Words added by the user are appended to the end of the array. When the program is exited, the file on the disk is erased, and the words in the array are then sent to a new file with the same name.

The first procedures to be developed set up the menus, windows, controls, and text-editing facilities. Here are the procedures and the data types and variables used by them.

```
  type
   Ptr = ^LongInt;
   Handle = ^Ptr;
   WindowRecord = array[1..78] of Integer;
   WindowPtr = ^WindowRecord;
   TERec = record
    DestRect, ViewRect, SelRect : Rect;
    LineHeight, FirstBL : Integer;
    SelPoint : LongInt;
    SelStart, SelEnd, Active : Integer;
    WorkBreak, ClikLoop, ClickTime : LongInt;
    ClickLoc : Integer;
    CaretTime : LongInt;
    CaretState, Just, TElength : Integer;
    HText : Handle;
    RecalBack, RecalLines, ClikStuff, CrOnly : Integer;
    TxFont, TxFace, TxMode, TxSize : Integer;
    InPort, HighHook : Ptr;
    CaretHook : Ptr;
    NLines : Integer;
    LineStarts : array[0..32000] of Integer;
   end;
 TEPtr = ^TERec;
 TEHandle = ^TEPtr;
var
 OldMenuBar, OurMenu : Handle;
 WindowStor1 : WindowRecord;
 EntryWindow : WindowPtr;
 TempRect, EntryWRect, PoemWRect, ClearRect : Rect;
 ViewRect1, ViewRect2, ViewRect3 : Rect;
 TE1, TE2, TE3, CurTE : TEHandle;
 ClearBut : Handle;
```

The InitMenu procedure starts to create the user interface by saving the Macintosh Pascal menu bar and then replacing it with a single menu with the menuID number 100.

```
procedure InitMenu;
 begin
  OldMenuBar := Pointer(LInLineF(GetMenuBar));
  InLineP(ClearMenuBar);
  OurMenu := Pointer(LInLineF(NewMenu, 100, 'Options'));
  InLineP(AppendMenu, OurMenu, 'Add a New Word;Write a Poem;(----;Quit');
  InLineP(InsertMenu, OurMenu, 0);
  InLineP(DrawMenuBar);
 end; {InitMenu}
```

Next the window is created with the InitWindows procedure. It creates and displays the window described earlier.

```
procedure InitWindows;
begin
 EntryWindow := @WindowStor1;
 SetRect(EntryWRect, 20, 45, 250, 300);
 if LInLineF(NewWindow, EntryWindow, EntryWRect, 'Enter a Word', True,
 0, Pointer(-1), False, 0 + 0) = ord(EntryWindow) then
 ;
end; {InitWindows}
```

The InitControls procedure is the most complex of the three and is used to display the interface, creating both a pushbutton and three text-editing records.

In succession, a view rectangle is defined, its text label is drawn, and a surrounding rectangle is drawn around the view record by enlarging the view rectangle by 4 pixels in each direction with InsetRect. You may notice that there are no destination rectangles declared. This is done to save space, since the view rectangles and the destination rectangles are the same.

```
procedure InitControl;
begin
 InLineP(SetPort, EntryWindow);
 SetRect(ClearRect, 120, 210, 210, 240);
 MoveTo(20, 40);
 DrawString('Word');
 SetRect(ViewRect1, 60, 28, 170, 45);
  ClearBut := Pointer(LInLineF(NewControl, EntryWindow, ClearRect,
 'Accept', True, 0, 0, 0, 0, 0 + 0));
 MoveTo(20, 70);
 DrawString('Syllables');
 SetRect(ViewRect2, 83, 58, 110, 74);
 MoveTo(20, 100);
 DrawString('Part of Speech');
 SetRect(ViewRect3, 115, 88, 155, 110);
 TempRect := ViewRect1;
 InsetRect(TempRect, -2, -2);
 FrameRect(TempRect);
 TempRect := ViewRect2;
 InsetRect(TempRect, -2, -2);
 FrameRect(TempRect);
 TempRect := ViewRect3;
 InsetRect(TempRect, -2, -2);
 FrameRect(TempRect);
 TE1 := Pointer(LInlineF(TENew, ViewRect1, ViewRect1));
 TE2 := Pointer(LInlineF(TENew, ViewRect2, ViewRect2));
 TE3 := Pointer(LInlineF(TENew, ViewRect3, ViewRect3));
end;
```

The next portion of the program to be looked at is the main program. In a Macintosh application the main program usually serves two purposes: it acts as an interrupt handler, waiting for events and then dispatching them to routines to handle them; and it tracks the cursor, changing it's shape when and where necessary.

This program is interested in two types of events, mouse down and key down. When a mouse down happens, it could be in one of four contexts: in the pushbutton, in the menu bar, in one of the three text edit view rectangles, or anywhere else on the screen (which, of course, has no meaning to the program). The main program must trap the events and then differentiate between the mouse downs, depending upon their location. The algorithm for handling the events is as follows: when GetNextEvent returns an event, a Case statement is used to differentiate between a mouse down and key down, based on the What field in the event record. If the event was a mouse down, several Toolbox routines are called to find out where it occurred and to track it. First, the FindWindow function is called. As you remember, FindWindow returns both a window pointer and a code that indicates if the mouse down occurred in the menu bar. Since this program only uses one window we are not interested in the window pointer returned—only the code. However, if the program was multi-windowed, we would use the window pointer to activate the window in which the mouse was clicked. In our program, if the window pointer returned does not equal the pointer for the window, we do a SysBeep. If the code returned is 1, which we have declared as the constant InMenuBar, a call is then made to the procedure HandleMenu from which MenuSelect is called, and the values returned are processed.

If the mouse down was not in the menu bar, the next place to check is the controls. The FindControl function is called and returns both a part code and the control handle for the control where the event occurred. In our case, this can only be the pushbutton. If the part code returned is greater than zero, the event was in the control and TrackControl is called to do just that. Track Control also returns a part code for where the mouse up was located. If the code is greater than zero, the user has pushed the button, and the procedure that handles the action of the pushbutton, AddWord, is called. Finally, if the mouse down was neither in the menu bar or the control, we check the text edit view rectangles to see if the event occurred in any of them. This is done by successive calls to PtInRect with each of the rectangles. If the event was in one of the view rectangles, a simple call to TEActivate with the text edit handle is performed.

The other event we are interested in is a key down. The only function a key down can serve is to enter text into the text edit record currently active. This is done in the procedure HandleKey after processing the Message field of the event record by calling TEKey.

The other function of the main program is to maintain the shape of the cursor as the I-beam in the edit record view rectangles and as the arrow elsewhere. This is done with continual calls to GetMouse, checking the mouse position whenever the program is not handling an event. If the mouse is in a text edit rectangle, the cursor is set to the I-beam; otherwise, it is set to the north-northwest arrow.

Here is the entire main program. It begins by calling the routines which set up windows, menus, controls and text editing.

```
begin {main program}
 InitWindows;
 InitMenu;
 InitControl;
 InitGrammars;
 IBeam := Pointer(LInLineF(GetCursor, 1));
 CurTE := TE1;
 InLineP(TEActivate, CurTE);
 Done := False;
 repeat
   repeat
    GetMouse(Mouse.h, Mouse.v);
      if PtInRect(Mouse, ViewRect1) or PtInRect(Mouse, ViewRect2) or
 PtInRect(Mouse, ViewRect3) then
    InLineP(SetCursor, IBeam^)
 else
  InitCursor;
 InLineP(TEIdle, CurTe);
 until GetNextEvent(15, Event);
 GlobalToLocal(Event.Where);
 case Event.What of
 1 : {Mouse down}
 begin
   ClickedRegion := WInLineF(FindWindow, Event.Where, @WhWindow);
   if ClickedRegion = InMenuBar then
    HandleMenu;
   if WInLineF(FindControl, Event.Where, EntryWindow, @WhControl) > 0
   then
    begin
      InControl := (WInLineF(TrackControl, whControl, Event.Where, nil));
      if InControl > 0 then
        AddWord;
    end
```

```
        else
        begin
         InLineP(TeDeactivate, CurTE);
         if PtInRect(Event.Where, ViewRect1) then
           CurTE := TE1;
         if PtInRect(Event.Where, ViewRect2) then
          CurTE := TE2;
         if PtInRect(Event.Where, ViewRect3) then
          CurTE := TE3;
         InLineP(TEActivate, CurTE);
         InLineP(TEClick, Event.Where, False, CurTE);
        end
       end;
      3 : {Key Down}
      HandleKey;
     end; {Case}
    until Done;
    InLineP(CloseWindow, EntryWindow);
    InLineP(DisposeMenu, OurMenu);
    InLineP(SetMenuBar, OldMenuBar);
    InLineP(DrawMenuBar);
    SaveWords;
  end. {program}
```

We can now look at the procedures tied to the user interface. The first one we will examine is called when FindWindow indicates that a mouse down event was located in the menu bar. HandleMenu calls MenuSelect, which returns a Long Integer with the menuID in the high word and the item number in the low word. Since we have only one menu we need not worry about the menuID. A value greater than zero returned by MenuSelect means that an item in the single menu was selected. Declared in the procedure are three constants representing the item numbers of the menu choices. Notice that the item numbers are 1, 2 and 4. The number 3 position in the menu is filled with a dotted line.

The item choice is obtained from the value returned by MenuSelect, with the LoWord function called from a Case statement. The Case statement then branches to a procedure that performs the action tied to that menu item. The last InLineP in HandleMenu is the obligatory HiLiteMenu used to change the menu title to black on white from white on black.

```
procedure HandleMenu;
 const
  Add = 1;
  Write = 2;
  Quit = 4;
 var
  MenuChoice : LongInt;
 begin
  MenuChoice := LInLineF(MenuSelect, Event.Where);
  case LoWord(MenuChoice) of
   Add :
    ClearWord;
   Write :
    WritePoem;
   Quit :
    Done := True;
  otherwise
  ;
 end; {case}
 InLineP(HiLiteMenu, 0);
end;
```

The HandleKey procedure is called every time a key down event is detected. It finds the character typed from the Message field of the event record and then adds it to whichever of the three text edit records is currently activated with TEKey. The procedure also checks to see if the number of characters in the current TE record is at the maximum for that field. These maximum values are 10 characters for the word field, 2 for the number of syllables, and 1 for the part of speech (*a* for adjective, *n* for noun, and *v* for verb).

```
procedure HandleKey;
  var
   MaxLen : Integer;
 begin
  if CurTE = TE1 then
   MaxLen := 10
  else if CurTE = TE2 then
   MaxLen := 2
  else if CurTE = TE3 then
   MaxLen := 1;
  Event.Message := BitAnd(Event.Message, 255);
  Ch := Chr(Event.Message);
  if (CurTE^^.TELength < MaxLen) or (Ch = Chr(8)) then
   InLineP(TeKey, Ch, CurTE)
  else
   SysBeep(5);
 end;
```

Now that we have looked at the procedures that handle the menu and text editing, we can turn our attention to the procedures that perform the actions tied to the menu items and the pushbutton. Two of these procedures you should be familar with because they are part of the old Haiku program.

The AddWord procedure adds a word and its associated information to the end of the array of words WordList. This operation is more difficult than it might seem at first glance. It requires converting the information in the three text edit records into the form required for storage in the fields of the record type WordRec. Specifically, the Word field in a WordRec is a string, but the text stored in a TERec is held in a packed array. That means that a conversion from packed array to string must be done. The syllable field must be converted from a one- or two-character string representing a number to an integer. The part of speech from a one-character code to a value of the enumerated type PartType.

The conversion from a packed array to which we have a handle to a string can be done by taking each character in the array and concatenating it to the end of the string. Remember, you cannot work with the TERec field HText directly because of Pascal's type checking. You must first assign that handle to one pointing to a packed array of Char.

```
TextData := Pointer(TE1^^.HText);
 PoemWord.Word := '';
  for I := 0 to TE1^^.TELength - 1 do
   begin
    S := TextData^^[I];
    PoemWord.Word := Concat(PoemWord.word, S);
   end;
```

Because the Concat function only works with strings and not with a string and a character, we must first assign the character we wish to add to a string of length and size one. Note that the first position in the packed array is position 0.

The conversion of the part-of-speech code is simply done with a Case statement.

```
TextData := Pointer(TE3^^.HText);
case TextData^^[0] of
 'n' :
 PoemWord.Part := noun;
 'v' :
 PoemWord.Part := verb;
 'a' :
 PoemWord.Part := adj;
otherwise
 SysBeep(5);
```

The third conversion requires us to take note of the difference between the ASCII code for a character and the value of that character as a digit. The ASCII code is the value of the digit plus the ASCII code of the character zero (0). An If statement is fine for converting a one- or two-digit numeric string.

```
if TE2^^.TELength = 2 then
 PoemWord.Syl := (Ord(TextData^^[0]) - Ord('0')) * 10 + Ord(TextData^^[1]) - Ord('0')
else
 PoemWord.Syl := Ord(TextData^^[0]) - Ord('0');
```

Here is the entire AddWord procedure. The only other function performed by AddWord is to place the record at the end of the array WordList, provided there is room. The procedure does only the minimal amount of input validity checking since the program is near the memory limit of Macintosh Pascal running on a 128K machine. The reader using another system may choose to add additional validity checks.

```
procedure AddWord;
 type
  Chars = packed array[0..9] of Char;
  Ptr = ^Chars;
  CharHandle = ^Ptr;
 var
  TextData : CharHandle;
  I : Integer;
  s : string[1];
   begin { place in WordFile}
    TextData := pointer(TE1^^.HText);
    PoemWord.Word := '';
    for I := 0 to TE1^^.TELength - 1 do
     begin
      S := TextData^^[I];
      PoemWord.Word := concat(PoemWord.word, S);
     end;
    TextData := Pointer(TE3^^.HText);
    case TextData^^[0] of
    'n' :
     PoemWord.Part := noun;
    'v' :
     PoemWord.Part := verb;
    'a' :
     PoemWord.Part := adj;
  otherwise
   SysBeep(5);
 end;
  TextData := Pointer(TE2^^.HText);
   if TE2^^.TELength = 2 then
   PoemWord.Syl := (Ord(TextData^^[0]) - Ord('0')) * 10 + Ord(TextData^^[1]) -
   Ord('0')
 else
  PoemWord.Syl := Ord(TextData^^[0]) - Ord('0');
   if NumWords+1 > MaxWords then
   begin
    WordList[NumWords + 1] := PoemWord;
    NumWords := NumWords + 1;
   end;
  end;
```

The next procedure is the one tied to the Add a New Word option in the menu. The ClearWord procedure simply clears the old text out of the text edit fields by selecting it and deleting it. Remember, the SelStart and SelEnd parameters of TESetSelect require long integers.

```
procedure ClearWord;
 begin
  InLineP(TESetSelect, 0 + 0, 100 + 0, TE1);
  InLineP(TEDelete, TE1);
  InLineP(TESetSelect, 0 + 0, 100 + 0, TE2);
  InLineP(TEDelete, TE2);
  InLineP(TESetSelect, 0 + 0, 100 + 0, TE3);
  InLineP(TEDelete, TE3);
  InLineP(TEActivate, TE1)
 end;
```

The poems are written by the WritePoem procedure, which is almost a direct translation from the original Haiku program. The only difference is that the procedure picks words from the array rather than the file.

```
procedure WritePoem;
 var
  Pick, Size : Integer;
 begin
  Previous.Part := Any;
  SylCount := 0;
 repeat
  Pick := Random mod NumWords + 1;
  if WordList[Pick].Part in Grammars[Previous.Part] then
   if SylCount + WordList[Pick].Syl < = 17 then
```

```
      begin
       Writeln(WordList[Pick].Word);
       Previous := WordList[Pick];
       SylCount := SylCount + WordList[Pick].Syl
      end;
     until SylCount = 17;
    end;
```

To complete the program, the last two procedures GetWords and Save-Words, retrieve the words from the file at the start of the program and store them back afterward. They can be seen in this complete program listing.

```
program MacHaiku;
 const
  TENew = $A9D2;
  TEIdle = $A9DA;
  TEActivate = $A9D8;
  TEDeactivate = $A9D9;
  TEDispose = $A9CD;
  TEKey = $A9DC;
  TEClick = $A9D4;
  TESetSelect = $A9D1;
  TEDelete = $A9D7;
  NewWindow = $A913;
  CloseWindow = $A92D;
  FindWindow = $A92C;
  SetPort = $A873;
  SetCursor = $A851;
  GetCursor = $A9B9;
  GetMenuBar = $A93B;
  ClearMenuBar = $A934;
  FreeMem = $A01C;
  NewMenu = $A931;
  AppendMenu = $A933;
  InsertMenu = $A935;
  DrawMenuBar = $A937;
  SetMenuBar = $A93C;
  DisposeMenu = $A932;
  MenuSelect = $A93D;
  HiLiteMenu = $A938;
  NewControl = $A954;
  FindControl = $A96C;
  HiLiteControl = $A95D;
  TestControl = $A966;
  GetCtlValue = $A960;
  SetCtlValue = $A963;
  TrackControl = $A968;
  SetCtlMax = $A965;
  MouseDown = 1;
  KeyDown = 3;
  InMenuBar = 1;
  MaxWords = 100;
 type
  PartType = (verb, noun, adj, any);
  PartSet = set of PartType;
  WordRec = record
   Word : string[10];
   Part : PartType;
   Syl : Integer
  end;
 Ptr = ^LongInt;
 Handle = ^Ptr;
 WindowRecord = array[1..78] of Integer;
 WindowPtr = ^WindowRecord;
 TERec = record
  DestRect, ViewRect, SelRect : Rect;
  LineHeight, FirstBL : Integer;
  SelPoint : LongInt;
  SelStart, SelEnd, Active : Integer;
  WorkBreak, ClikLoop, ClickTime : LongInt;
  ClickLoc : Integer;
  CaretTime : LongInt;
  CaretState, Just, TElength : Integer;
  HText : Handle;
  RecalBack, RecalLines, ClikStuff, CrOnly : Integer;
  TxFont, TxFace, TxMode, TxSize : Integer;
  InPort, HighHook : Ptr;
  CaretHook : Ptr;
  NLines : Integer;
  LineStarts : array[0..32000] of Integer;
 end;
TEPtr = ^TERec;
TEHandle = ^TEPtr;
```

```
var
 OldMenuBar, OurMenu : Handle;
 WindowStor1 : WindowRecord;
 EntryWindow, PoemWindow, WhWindow : WindowPtr;
 TempRect, EntryWRect, PoemWRect, ClearRect : Rect;
 ViewRect1, ViewRect2, ViewRect3 : Rect;
 TE1, TE2, TE3, CurTE : TEHandle;
 Event : EventRecord;
 Mouse : Point;
 WhControl, IBeam : Handle;
 ClearBut : Handle;
 InControl, WhichPart, ClickedRegion, SYlCount : Integer;
 Ch : Char;
 Done : Boolean;
 PoemWord, Previous : WordRec;
 WordFile : file of WordRec;
 InSet : set of Char;
 Grammars : array[PartType] of PartSet;
 WordList : array[1.. MaxWords] of WordRec;
 NumWords : Integer;

procedure InitMenu;
begin
 OldMenuBar : = Pointer(LInLineF(GetMenuBar));
 InLineP(ClearMenuBar);
 OurMenu : = Pointer(LInLineF(NewMenu, 100, 'Options'));
 InLineP(AppendMenu, OurMenu, 'Add a New Word;Write a Poem;(----;Quit');
 InLineP(InsertMenu, OurMenu, 0);
 InLineP(DrawMenuBar);
end; {INitMenu}
procedure InitWindows;
begin
 EntryWindow : = @WindowStor1;
 SetRect(EntryWRect, 20, 45, 250, 300);
  if LInLineF(NewWindow, EntryWindow, EntryWRect, 'Enter a Word', True, 0, Pointer(-1),
False, 0 + 0) = ord(EntryWindow) then
  ;
end; {InitWindows}
```

```
 procedure InitControl;
 begin
  InLineP(SetPort, EntryWindow);
  SetRect(ClearRect, 120, 210, 210, 240);
  MoveTo(20, 40);
  DrawString('Word');
  SetRect(ViewRect1, 60, 28, 170, 45);
   ClearBut := Pointer(LInLineF(NewControl, EntryWindow, ClearRect,
 'Accept', True, 0, 0, 0, 0, 0 + 0));
  MoveTo(20, 70);
  DrawString('Syllables');
  SetRect(ViewRect2, 83, 58, 110, 74);
  MoveTo(20, 100);
  DrawString('Part of Speech');
  SetRect(ViewRect3, 115, 88, 155, 110);
  TempRect := ViewRect1;
  InsetRect(TempRect, -2, -2);
  FrameRect(TempRect);
  TempRect := ViewRect2;
  InsetRect(TempRect, -2, -2);
  FrameRect(TempRect);
  TempRect := ViewRect3;
  InsetRect(TempRect, -2, -2);
  FrameRect(TempRect);
  TE1 := Pointer(LInlineF(TENew, ViewRect1, ViewRect1));
  TE2 := Pointer(LInlineF(TENew, ViewRect2, ViewRect2));
  TE3 := Pointer(LInlineF(TENew, ViewRect3, ViewRect3));
end;
procedure AddWord;
 type
  Chars = packed array[0..9] of Char;
  Ptr = ^Chars;
  CharHandle = ^Ptr;
 var
  TextData : CharHandle;
  I : Integer;
  Temp : string[5];
  s : string[1];
```

```
begin { place in WordFile}
 TextData := pointer(TE1^^.HText);
 PoemWord.Word := ';
for I := 0 to TE1^^.TELength - 1 do
 begin
  S := TextData^^[I];
  PoemWord.Word := concat(PoemWord.word, S);
 end;
TextData := Pointer(TE3^^.HText);
case TextData^^[0] of
 'n' :
 PoemWord.Part := noun;
 'v' :
 PoemWord.Part := verb;
 'a' :
 PoemWord.Part := adj;
otherwise
 SysBeep(5);
end;
 TextData := pointer(TE2^^.HText);
if TE2^^.TELength = 2 then
 PoemWord.Syl := (Ord(TextData^^[0]) - Ord('0')) * 10 +
 Ord(TextData^^[1]) - Ord('0')
else
 PoemWord.Syl := Ord(TextData^^[0]) - Ord('0');
 if NumWords+1 > MaxWords then
  begin
   WordList[NumWords + 1] := PoemWord;
   NumWords := NumWords + 1;
  end;
 end;
 procedure WritePoem;
  var
   Pick, Size : Integer;
 begin
   Previous.Part := Any;
   SylCount := 0;
```

```
repeat
 Pick := Random mod NumWords + 1;
 if WordList[Pick].Part in Grammars[Previous.Part] then
 if SylCount + WordList[Pick].Syl < = 17 then
   begin
    Writeln(WordList[Pick].Word);
    Previous := WordList[Pick]
    SylCount := SylCount + WordList[Pick].Syl
   end;
 until SylCount = 17;
end;
until SylCount = 17
end;
procedure InitGrammars;
begin
 Grammars[Nouns := [adj, verb];
 Grammars[adj, noun];
 Grammars[verb] := [adj, noun];
 Grammars[any] := ]adj, verb, noun]
end;
procedure ClearWord;
begin
 InLineP(TESetSelect, 0 +0, 100 + 0, TE 1);
 InLineP(TEDelete, TE1);
 InLineP(TESetSelect, 0 + 0, 100 + 0, TE2);
 InLineP(TEDelete, TE2);
 InLineP(TESetSelect, 0 + 0, 100 + 0, TE3);
 InLineP(TEDelete, TE3);
 InLineP(TEActivate, TE1)
end;
procedure GetWords;
begin
 Open(WordFile, 'Words.Data');
 NumWords := 1;
 while not (eof(WordFile)) do
  begin
   WordList[NumWords] := WordFile ^ ;
   NumWords := NumWords + 1;
   Get(WordFile);
  end;
Close(WordFile)
end
```

```
procedure SaveWords;
 var
 | : Integer;
begin
 ReWrite(WordFile, 'Words.Data');
 Open(WordFile);
 for I: = 1 to NumWords do
  begin
   WordFile ^ := WordList[1]:
    Put(WordFile)
  end
end;
procedure HandleMenu;
 const
  Add = 1;
  Write = 2;
  Quit = 4;
 var
  MenuChoice : LongInt;
begin
 MenuChoice := LInLineF(MenuSelect, Event.Where);
 caseLoWord(MenuChoice) of
  Add:
   ClearWord;
  Write:
   WritePoem;
  Quit :
   Done := True;
otherwise
 ;
end; {case}
InLineP(HiLiteMenu, 0);
end;
procedure HandleKey;
var
 MaxLen : Integer;
begin
 if CurTE = TE1 then
  MaxLen := 10
 elseif CurTE = TE2 then
  MaxLen := 2
 elseifCurTE = TE3 then
  MaxLen := 1;
 Event.Message := BitAnd(Event.Message, 255);
 Ch := Chr(Event Message);
 if (CurTE^^.TELength < MaxLength) or (Ch = Chr(8)) then
  InLineP(TeKey, Ch, CurTE)
 else
  SysBeep(5);
end;
 begin
  HideAll;
  GetWords;
  NumWords := 0;
  ShowText;
  InitWindows;
  FlushEvents(15, 0);
  InitMenu;
  InitControl;
  InitGrammars;
  IBeam := Pointer(LInLineF(GetCursor, 1));
  CurTE := TE1;
  InLineP(TEActivate, CurTE);
  Done := False;
  repeat
   repeat
    GetMouse(Mouse.h, Mouse.v);
     if PtInRect(Mouse, ViewRect1) or PtInRect(Mouse, ViewRect2) or PtInRect(Mouse, View-
Rect3) then
     InLineP(SetCursor, IBeam^)
   else
    InitCursor;
   InLineP(TEIdle, CurTe);
  until GetNextEvent(15, Event);
  GlobalToLocal(Event.Where);
   case Event.What of
     1:
    begin
     ClickedRegion := WInLineF(FindWindow, Event.Where, @WhWindow);
     if ClickedRegion = InMenuBar then
      HandleMenu;
     if WInLineF(FindControl, Event.Where, EntryWindow, @WhControl) > 0
     then
      begin
       InControl := (WInLineF(TrackControl, whControl, Event.Where, nil));
       if InControl > 0 then
         AddWord;
      end
      begin
       InLineP(TeDeactivate, CurTE);
      if PtInRect(Event.Where, ViewRect 1then)
       CurTE := TE1;
      if PtInRect(Event.Where, ViewRect2) then
       CurTE := TE2;
      if PtInRect(Event.Where, ViewRect3) then
       CurTE := TE3;
       InLineP(TEActivate, CurTE);
       InLineP(TEClick, Event.Where, False, CurTE);
     end
        end;
       3 :    {Key Down} HandleKey;
      end; {Case}
     until Done;
     InLineP(CloseWindow, EntryWindow);
     InLineP(DisposeMenu, OurMenu);
     InLineP(SetMenuBar, OldMenuBar);
     InLineP(DrawMenuBar);
     SaveWords;
    end
```

Some Final Notes on Using InLines

We have now seen how the use of InLines allow us to create true Macintosh applications in Macintosh Pascal. Of course, since the program must first pass through Macintosh Pascal, some problems can and do occur.

First, program execution speed is slower than if the same program had been written in a compiled language such as C or assembly language. This is due to the fact that Macintosh Pascal must decode every statement each time it is executed (the bane of interpreters). Even when the statement is an InLine which, in turn, branches to a Toolbox ROM routine, time is needed to recognize the InLine keywords. Fortunately, the QuickDraw operates so fast that very little speed degradation can be noticed.

Secondly, the maximum program size is limited since Macintosh Pascal occupies a significant portion of memory by itself. This situation is aggrevated when using features such as windows that need large amounts of memory to store information about them. Remember, Macintosh Pascal itself allocates five window records for its Program, Text, Drawing, Instant, and Observe windows. When using InLines, lack of memory space will cause the program to bomb, displaying everyone's favorite System Error alert box. The error code for this type of error will appear as ID = 2. A general System Error debugging hint when programming with InLines is to try to reduce the amount of memory required by the program and run it again to see if it still doesn't work. A simple way of doing this is to call the HideAll procedure to remove from memory the program text, freeing that space. Some of the space shortage is ameliorated with the use of the Version 2.0 Application Shell, which contains only the interpreter and omits all of the user/programmer interface, including the ability to edit a program. This frees a large amount of the memory required by the complete Macintosh Pascal system. (For more on the Application Shell see Appendix A.)

Finally, it is important to keep in mind that you are sharing memory with Macintosh Pascal, and a small programming mistake may cause a system error and destroy Macintosh Pascal's ability to interpret your program. An example of this type of mistake is the sloppy use of a pointer or handle, which overwrites part of the interpreter or its supporting window, control, or menu records.

CHAPTER

7

Advanced QuickDraw

The depth and breadth of QuickDraw goes far beyond the simple shape drawing demonstrated in Chapter 5. QuickDraw's more advanced features include complex drawing shapes that are dynamically allocated in the heap and precise control over the drawing environment. This chapter covers the majority of the advanced features of QuickDraw.

GrafPorts

In Chapter 5, GrafPorts were described as a self-contained drawing environment. Also mentioned was the fact that the current GrafPort in Macintosh Pascal is the Drawing window. Now that the Window Manager and the ability to create a window has been introduced, it seems appropriate to expand the discussion of GrafPorts.

A GrafPort is a record used to define how drawing will take place in an area of the screen. The GrafPort record is dynamically allocated and accessed via a pointer. Many GrafPorts can be open at once, though usually a single GrafPort is associated with a window. When you define a window, a GrafPort is automatically created for that window as part of its window record. The record type of a GrafPort is:

```
type
 GrafPtr = ^GrafPort;
 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;
  bkColor : LongInt;
  colorBit : Integer;
  patStretch : Integer;
  picSave : QDHandle;
  rgnSave : QDHandle;
  polySave : QDHandle;
  grafProcs : QDProcsPtr;
end;
```

Most of the fields in this record will be unfamiliar to you at this point. They are used, to describe the characteristics of the drawing environment, the pen, the font being used, and so forth. The exact meaning of each is as follows.

The device field is the number of the output device the GrafPort will be using. The default value of 0 is the screen and is almost always the value used.

The portsBit field is the BitMap that points to the BitImage where all drawing will be done. This is set by the WindowManager but is by default the entire screen (0, 0, 512, 342).

The portRect field is a rectangle that describes where, inside the BitImage, drawing will take place. This is set by the Window Manager.

The visRgn and clipRgn fields are manipulated by the Window Manager and describe how much of the window is used for drawing and where the drawing will be clipped.

The bkPat and fillPat fields hold patterns used by certain QuickDraw routines drawing in the GrafPort. BkPat is the background pattern used to erase an area of the screen. It is white by default. The fillPat is the pattern used by the QuickDraw fill routines. It is black by default. QuickDraw contains routines to manipulate these fields.

The pnLoc, pnSize, pnMode, pnPat, and PnVis fields describe the characteristics of the pen. Routines that manipulate these fields are described extensively in this chapter.

The txFont, txFace, txMode, and txSize fields hold information about the font to draw characters. Routines that manipulate these fields are described extensively in this chapter.

The fgColor, bkColor, and colorBit field hold information regarding drawing in color. Routines that manipulate these fields are described in this chapter.

The patStretch field is used during output to a printer. A program should not alter its value.

The picSave, rgnSave, and polySave fields are used to help define the QuickDraw complex drawing shapes pictures, polygons, and regions. They are covered extensively in this chapter. A program is not concerned with the values of these fields.

Finally, the grafProcs field may be used to point to a special data structured used to customize QuickDraw drawing operations. Detailed information about this feature can be found in Apple's QuickDraw Programming Guide, a section of *Inside Macintosh*.

More on Drawing Text

In addition to selecting the font or textface of characters being drawn on the screen, QuickDraw offers wide control over the characteristics of text (Figure 7.1), as you might have suspected from using programs such as MacWrite and MacPaint. The drawing characteristics are held in fields of the GrafPort and are manipulated by QuickDraw routines.

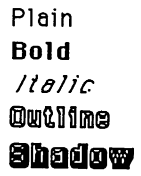


Figure 7.1 Text characteristics

To describe the style of text drawn, QuickDraw has an enumerated data type style defined as:

```
type
  StyleItem = (Bold, Italic, Underline, Outline, Shadow, Condense, Expand);
```

Since text can be drawn with combinations of the style attributes, a set of StyleItem has also been predefined by QuickDraw.

```
type
  Style = set of StyleItem;
```

The use of a set makes it possible to describe varying numbers of text characteristics to QuickDraw. This is done with the TextFace procedure.

```
procedure TextFace (Face : Style);
```

The TextFace procedure is used to indicate the text characteristics of text drawn in future calls to DrawString and DrawChar. Since Style is defined as a set, a set must be passed as a parameter.

```
TextFace([Bold]);                           {Bold}
TextFace([Italic, Underline]);              {Italic, Underline}
TextFace([Outline, Shadow, Condensed]);     {Outline, Shadow, Condensed}
TextFace([]);                               {Normal}
```

The size of the font being drawn can also be altered, demonstrating one of the more unique features of QuickDraw. The TextSize procedure is used for this purpose.

```
procedure TextSize(Size : Integer);
```

The TextSize procedure sets the size of the characters to be drawn by future calls to DrawChar or DrawString. The size is expressed in typographer's *points* which are each 1/72 of an inch and are not equivalent to the points used to describe the intersection of two lines on the coordinate plane. For instance, a 6-point character will be 6/72 of an inch high.

Any size can be specified but the results will vary depending on the contents of the system file on the disk being used. This is because the description of a font is maintained in the system resource file or, more rarely, in an application resource file. Separate definitions are needed for each different size font. Since font definitions take up a lot of space (approximately 5K for a 12-point font), it is unusual to find all the possible sizes of a font in the resource file. The designers of QuickDraw were quick to realize this situation and include in QuickDraw the ability to simulate nonresident font sizes by stretching or shrinking existing sizes of the same font. The scheme works as follows. When the TextSize procedure is called, QuickDraw calls on the Toolbox Font Manager to look for the definition of the font in that exact size. If the exact size isn't available, it looks for a size that is close and then scales it to the desired size. The procedural steps are as follows:

- It looks first for a font that's twice the size to scale it down.
- If no font twice the size exists, it looks for a font one half the size and scales it up.
- If no font one half the size exists, it looks for a larger size of the font and scales down the next larger size if one is there.
- If no larger size font is available, it looks for a smaller size and scales up the next smallest size if one is there.

The farther down this list the Font Manager must traverse, the worse the characters look on the screen. For example, if a 12-point font is requested and doesn't exist, the simulated font will look better if it is based on shrinking a 24-point font rather than expanding a 10-point font. You may have noticed this effect while using application programs.

If a size of zero is specified, the font manager will choose an available size closest to the system font size, which is 12. The default setting of TextSize is 0.

If a request font is not available in any size, the application font is used instead and is scaled to the requested size. If the application font is available in any size, the system font is then used, scaled to the appropriate size.

Determining Text Widths

In addition to the previously discussed StringWidth function, QuickDraw also includes a function that determines the width of an individual character.

```
function CharWidth(Ch : Char);
```

The CharWidth function returns the width of a single character in points (screen points, not typographer points). By using CharWidth you can see the effect that style and font variations have on the size of a particular character.

Pen Characteristics

In Chapter 5, the pen was introduced as the tool used by QuickDraw to draw shapes and text. The pen has four adjustable characteristics: its location, size, drawing mode, and drawing pattern.

Pen Location

The pen's location is a point in the coordinate grid where QuickDraw will start to draw the next shape, character, or line. The pen hangs down and to the right of this point. It's location is altered by some drawing routines and not by others. Drawing text moves the pen to the right of the last character, and drawing a line moves the pen to the last point on the line drawn. In addition, the Move and MoveTo procedures also alter the pen's location. Interestingly, the shape drawing routines, such as FrameRect and PaintOval, do not alter the pen location at all.

The pen's shape is rectangular with a default size of a 1-by-1 point square. The size of the pen can be changed with the PenSize procedure.

```
procedure PenSize(Width, Length : Integer);
```

The PenSize procedure sets the width and height of the pen to the number of pixels indicated. The minimum size of (0,0) causes no drawing to be performed; the maximum pen size of (32767, 32767) would draw the entire screen at once. Changing the PenSize allows you to alter the width of lines and outlines drawn. The following call to PenSize:

```
PenSize(8,10);
```

has the effect shown in Figure 7.2 on the default pen size.

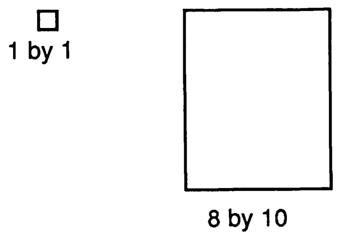


Figure 7.2 The change in PenSize

Pen Pattern

The pen pattern is the ink with which the pen writes. The standard pen patterns are the predefined QuickDraw values of Black, White, Gray, ltGray, and dkGray. The default pattern is black. Other patterns can also be defined and used. This feature is demonstrated later in this chapter with a program that lets you create your own patterns. We have already seen how the PenPat procedure can be used to alter the default pen pattern.

Pen Mode

The pen mode determines how the pen pattern is to affect what's already on the screen. It gives the programmer the option of several drawing modes rather than just painting the pen pattern over whatever is already on the screen. The pen mode is actually a Boolean operation with which QuickDraw compares the bit to be drawn with the bit already on the screen in order to determine what to draw. The eight Boolean operations are called:

patCopy	not PatCopy
PatOr	notPatOr
patXor	notPatXor
patBic	notPatBic

The truth tables for each of the pen patterns are as follows:

patCopy

Source pixel	Destination pixel	Result pixel
black	black	black
black	white	black
white	black	white
white	white	white

patOr

Source pixel	Destination pixel	Result pixel
black	black	black
black	white	black
white	black	black
white	white	white

patXor

Source pixel	Destination pixel	Result pixel
black	black	white
black	white	white
white	black	black
white	white	white

patBic

Source pixel	Destination pixel	Result pixel
black	black	white
black	white	black
white	black	black
white	white	white

The Not transfer modes reverse the result pixel in the normal transfer mode from black to white or white to black. For example:

notPatCopy

Source pixel	Destination pixel	Result pixel
black	black	white
black	white	white
white	black	black
white	white	black

The effect of the pen mode on a four-pixel screen area is shown in Figure 7.3.

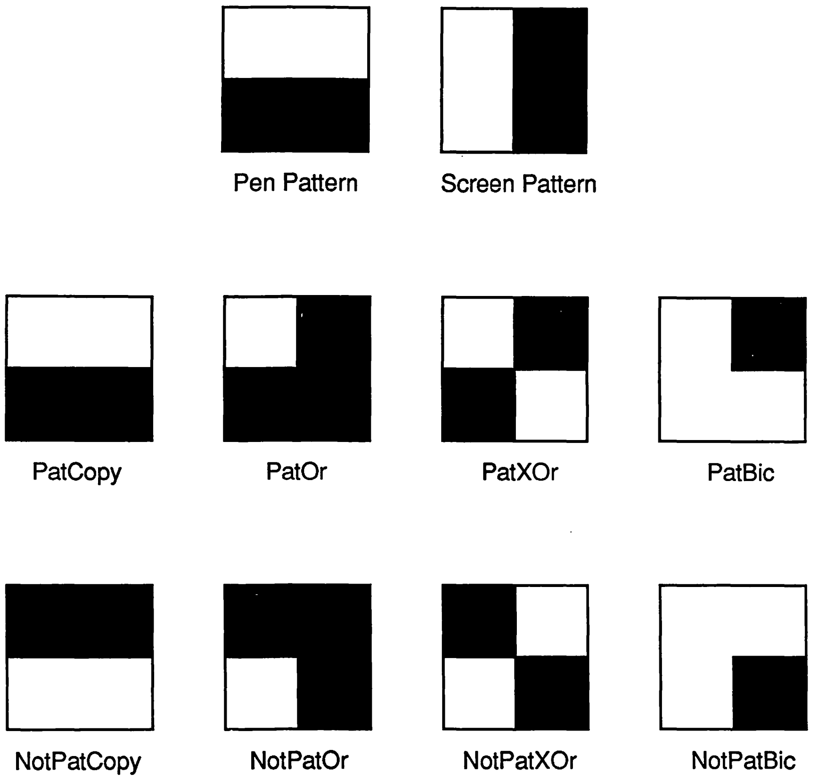


Figure 7.3 The pen modes

The pen mode can be changed with the PenMode procedure.

```
procedure PenMode(mode : Integer);
```

The PenMode procedure sets the transfer mode with which the pen pattern is transferred when lines or shapes (not characters) are drawn. The mode can be expressed as either an integer or one of the predefined Quick-Draw constants.

```
const
 patCopy     8;
 patOr       9;
 patXor      10;
 patBic      11;
 notPatCopy  12;
 notPatOr    13;
 notPatXor   14;
 notPatBic   15;
```

QuickDraw contains a set of additional procedures for manipulating pen characteristics.

```
procedure PenNormal;
```

The PenNormal procedure resets the pen to the initial pen settings so that:

PenSize	(1,1)
PenMode	patCopy
PenPat	Black

The pen location is not altered by a call to PenNormal.

```
procedure GetPenState(var pnState : PenState);
```

The GetPenState procedure stores the current pen location, size, pattern, and mode. QuickDraw conveniently provides a predefined record type called PenState to store the pen information.

The type PenState is defined as:

```
PenState = record
 pnLoc : Point;
 pnSize : Point
 pnMode : Integer;
 pnPat : Pattern
end;
```

This routine is useful when calling subroutines that alter the pen characteristics. The reverse operation of GetPenState is SetPenState.

```
procedure SetPenState (pnState : PenState);
```

The procedure SetPenState sets the pen location, size, mode, and pattern to the information held in the specified record of type PenState.

The remaining Pen routines follow.

```
procedure GetPen(var Pt : Point);
```

The GetPen procedure returns, as a point, the current location of the pen.

```
procedure HidePen;
```

The HidePen procedure causes the pen not to draw on the screen even when drawing procedures are called. HidePen is automatically called by certain complex drawing routines such as OpenRgn or OpenPoly.

```
procedure ShowPen;
```

The ShowPen procedure is the opposite of HidePen, making the pen visible.

Line Drawing Routines

QuickDraw contains two routines for drawing straight lines with the pen that are analogous to the two pen-movement routines Move and MoveTo.

```
procedure LineTo(h, v : Integer);
```

The LineTo procedure draws a line from the current pen location to the new location (h,v) expressed in local coordinates. The width of the line is the pen size and the pattern drawn is the pen pattern.

```
procedure Line(dh, dv : Integer);
```

The Line procedure draws a line from the current pen location to a point that is a distance dh points away horizontally and dv points away vertically. After a call to the Line procedure, the new pen location is (h + dh, v + dv) where (h, v) was the pen location before the call to Line.

Complex Drawing Shapes

In addition to the relatively simple drawing shapes such as rectangles and ovals, QuickDraw has the ability to define and draw quite complex shapes such as regions. A region is an arbitrary set of spatially coherent points on which complex yet rapid calculations can be performed. It is structures such as regions that set QuickDraw apart from all other graphic packages. A region is defined as a combination of lines, shapes such as rectangles, and other regions. The shape of a region (Figure 7.4) can consist of one area or many areas and solid areas or areas with holes in them. A region can best be thought of as an outline that divides the bit map into pixels within the region and pixels outside the region. The Region procedure and functions belong to the QuickDraw2 library and, for the first time, a **uses** statement will have to be included in programs.

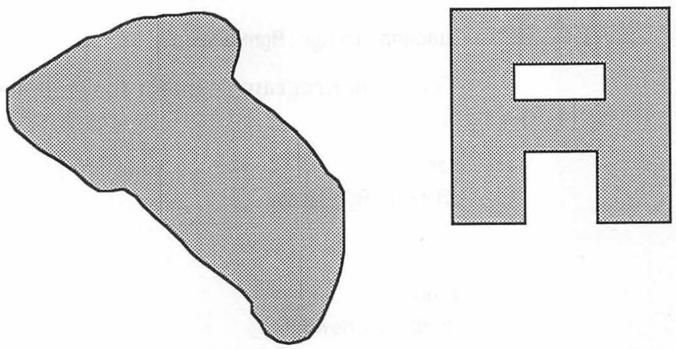


Figure 7.4 Examples of two regions

The data structure of a region is a variable length record with two fixed fields at its beginning.

```
uses
 QuickDraw2;
type
 Region = record
  RgnSize : Integer;
  RgnBox : Rect;
  {Optional region definition data}
end;
```

The RgnSize field contains the size in bytes of the entire region variable. The RgnBox is a rectangle that completely encloses the entire region.

Since regions are of variable size, they are stored dynamically in the heap and moved around by the operating system's memory management software as their size changes. This dynamic nature requires that access to a region be done with a handle. It should also be noted that in a program tight for memory, creation or expansion of a region could cause a program to exceed the amount of memory available. No manipulation is ever done directly on the region record; therefore, the programmer does not need to worry about the contents of the region record.

A region is created by a QuickDraw function which allocates space for the region and returns a handle to it. The handle is of the QuickDraw type RgnHandle where:

```
type
 RgnPtr = ^Region;
 RgnHandle = ^RgnPtr;
```

Once again, regions are manipulated only by use of their handle and never directly.

The function NewRgn allocates a new region, initializes the region record to the null region (RgnBox field set to 0,0,0,0), and returns a handle to the region.

```
function NewRgn : RgnHandle;
```

The small program segment that follows demonstrates the creation of a region.

```
var
 BarBell : RgnHandle;
  .
  .
begin
 Barbell := NewRgn;
```

First, a handle to the region is declared in the program's **var** section. This action is just like declaring any other variable. Then the NewRgn function is called to create the region, initialize it to the null region, and return a handle to it in the region handle Barbell. Notice that a Region record is never declared in the program but is created dynamically by the call to NewRgn.

Once a region is created and a handle to it returned, the components of the region are placed in the region record with the OpenRgn procedure.

```
procedure OpenRgn;
```

The procedure OpenRgn starts to save the definition of a region. Notice that the procedure does not have a parameter specifying which region is being defined. That is because OpenRgn allocates temporary space for the region being created, which is later assigned to a specific region. When a region is open, all calls to Line, LineTo, and the procedures that frame shapes define the outline of the region. OpenRgn call the procedure HidePen so that no actual drawing takes place while the region is open. Since the pen hangs to the right and below the pen location, even the smallest pen will affect bits that lie outside the region defined.

The region outline separates the bit map into those bits within the region and those bits outside the region.

```
BarBell := NewRgn
OpenRegion;
SetRect(Temp, 20, 20, 30, 50);
FrameOval(Temp);
SetRect(Temp, 30, 30, 80, 40);
FrameRect(Temp);
SetRect(Temp, 80, 20, 90, 50);
FrameOval(Temp);
CloseRgn(BarBell);
```

The program segment above is a complete region definition. After the calls to NewRgn and OpenRgn, the region is defined by the drawing statements. Of course, no drawing is done at this time. The region definition is completed with the CloseRgn procedure.

```
procedure CloseRgn(dstRgn : RgnHandle);
```

The CloseRgn procedure stops the collection of drawing routines into the temporary region created by OpenRgn and saves the defined region into the region indicated by the dstRgn parameter. Even though no drawing is done by CloseRgn, the procedure does a call to ShowPen to restore the pen's drawing capability.

Drawing a Region

Once a region is defined it can be drawn on the screen with one of the region drawing routines.

```
procedure FrameRgn(rgn : RgnHandle);
```

The FrameRgn procedure draws a hollow outline just inside the specified region using the current pen pattern, mode, and size. The outline is as wide as the pen width and as tall as the pen height, but under no circumstances will the outline go outside the region boundary. FrameRgn has no effect on the pen location.

Since this is an outlining routine, it can be called while a region is open. If it is the outline of the region being framed it is mathematically added to the region that is open.

To draw the region already defined, a call to the FrameRgn procedure is used. The result is shown in Figure 7.5.

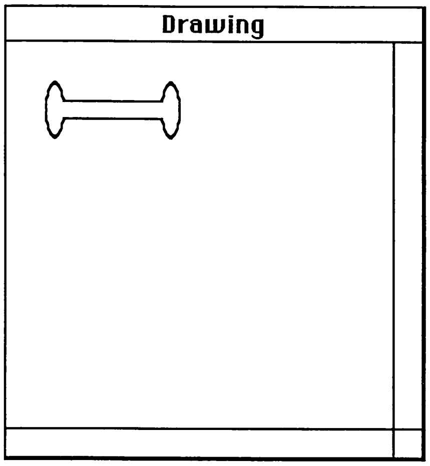


Figure 7.5 The result of FrameRgn (BarBell)

```
FrameRgn(BarBell);
```

The rest of the region drawing routines should appear familiar to you.

```
procedure PaintRgn(rgn : RgnHandle);
```

The procedure PaintRgn paints the specified region with the current pen pattern according to the current pen transfer mode. The pen location is not changed by PaintRgn.

procedure EraseRgn(rgn : RgnHandle);

The procedure EraseRgn erases the specified region by painting it with the background pattern. The current pen mode or pattern is ignored and the pen location is not changed.

procedure InvertRgn(rgn : RgnHandle);

The procedure InvetRgn flips the pixels enclosed in the specified region from black to white or from white to black. The current pen mode and pattern are ignored and the pen location is not changed.

procedure FillRgn(rgn : RgnHandle; pat : Pattern);

The FillRgn procedure fills the specified region with the given pattern in patCopy mode. The current pen mode and pattern are ignored and the pen location is not changed.

Disposing a Region

Once a region is no longer needed it should be disposed of so the operating system can reclaim the section of memory occupied by the region. The DisposeRgn procedure is used for this purpose.

procedure DisposeRgn (rgn : RgnHandle);

The DisposeRgn procedure deallocates the space used to store the region record and returns the memory to the free memory pool. Once DisposeRgn is called, no other attempts to manipulate that region should be made or you risk working with the dreaded dangling pointers.

Calculations with Regions

QuickDraw contains several routines that perform sophisticated mathematical manipulation of regions.

The OffsetRgn procedure is used to move a region across the coordinate plan a distance of dh points horizontally and dv points vertically.

procedure OffsetRgn(rgn : RgnHandle; dh, dv : Integer);

A positive value for dh and dv moves the region to the right and down. A negative value for either moves the region in the opposite horizontal or vertical direction. This procedure doesn't affect the screen unless the region is redrawn after the call to OffsetRgn. This procedure is particularly efficient since the components are stored internally, relative to the rectangle surrounding the region, and thus the offset need only be applied to the rectangle.

The ptinRgn function is analogous to the ptinRect function. It determines whether a specific point falls within a certain region, returning True if so or False if not.

```
function ptinRgn( pt : Point; rgn : RgnHandle) : Boolean;
```

The following program illustrates a simple video-type game using regions. The program creates and displays a region that looks like a "bull's-eye" target. This region has a large hole in it. The target is animated across the screen by erasing it, offsetting it, and then repainting it in the new location. The object of the game is to position the cursor in either of the two painted bands of the bull's eye. This is detected by continual calls to GetMouse and then checking to see if the cursor position is in the region with ptinRgn. If so, the region is inverted and a SysBeep beeps the Macintosh's speaker.

```
program BullsEye;
 uses
  QuickDraw2;
 var
  Eye : RgnHandle;
  K : Integer;
  Temp : Rect;
  Pt : Point;
 begin
  Eye : = NewRgn;
  OpenRgn;
  SetRect(Temp, 40, 40, 80, 80);
  FrameOval(Temp);
  SetRect(Temp, 50, 50, 70, 70);
  FrameOval(Temp);
  SetRect(Temp, 55, 55, 65, 65);
  FrameOval(Temp);
  CloseRgn(Eye);
  PaintRgn(Eye);
  for K : = 1 to 30 do
   begin
    EraseRgn(Eye);
    OffSetRgn(Eye, 3, 3);
    PaintRgn(Eye);
    GetMouse(pt.h, pt.v);
    if ptinRgn(pt, Eye) then
     begin
      InvertRgn(Eye);
      SysBeep(8)
     end
   end
 end.
```

procedure CopyRgn(rgn1, rgn2 : RgnHandle);

The CopyRgn procedure copies the mathematical structure of Rgn1 into Rgn2. Since an actual copy is made, any subsequent changes to Rgn1 have no effect on Rgn2. CopyRgn does not create the destination region; this must be previously done with NewRgn.

procedure SetEmptyRgn(rgn : RgnHandle);

The SetEmptyRgn procedure wipes out the contents of the specified region and sets it to the empty region (0,0,0,0).

procedure SetRectRgn(rgn : RgnHandle; Left, Top, Bottom, Right);

The SetRectRgn procedure takes similar action to SetEmptyRgn except that after the contents are wiped out, the rectangle surrounding the region is set to the region specified by Left, Top, Bottom and Right.

procedure RectRgn(rgn : RgnHandle; R : Rect);

The RectRgn procedure does the same thing as SetRectRgn except that the new rectangle is specified by a variable of type Rect rather than four boundary coordinates.

procedure InsetRgn (rgn : rgnHandle; dh, dv : Integer);

The InsetRgn procedure shrinks or expands the specified region. All the points on the region boundary are moved inward a distance dv points vertically and dh points horizontally. If dh or dv are negative the movement in that direction is outward instead of inward. InsetRgn performed on a rectangular region acts just like InsetRect.

procedure SectRgn(rgn1, rgn2, dstRgn : RgnHandle);

The SectRgn procedure calculates the intersection of Rgn1 and Rgn2 and places it in a third region dstRgn. This procedure does not create the destination region; that must be done with NewRgn. As an interesting note to the sophisticated programmer, the analogous rectangle routine SectRect requires that the destination rectangle be a variable parameter, but this is not the case for SectRgn. This is because in SectRgn, the destination is not the region itself but a pointer to the region; no value is placed into the parameter but into what is pointed to by the handle.

The following program outlines two rectangular regions and then paints their intersection (Figure 7.6).

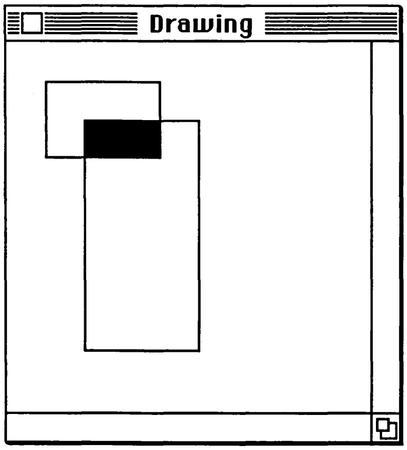


Figure 7.6 The intersection of Rgn 1 and Rgn 2

If the intersection does not exist or one of the regions is empty, the result is the empty region.

```
program InsectRgn;
 uses
  QuickDraw2;
 var
  Rgn1, Rgn2, dstRgn : RgnHandle;
  Temp : Rect;
 begin
  Rgn1 := NewRgn;
  Rgn2 := NewRgn;
  dstRgn := NewRgn;
  OpenRgn;
  SetRect(Temp, 20, 20, 80, 60);
  FrameRect(Temp);
  CloseRgn(Rgn1);
  FrameRgn(Rgn1);
  OpenRgn;
  SetRect(Temp, 40, 40, 100, 160);
  FrameRect(Temp);
  CloseRgn(Rgn2);
  FrameRgn(Rgn2);
  SectRgn(Rgn1, Rgn2, dstRgn);
  PaintRgn(dstRgn)
end.
```

procedure UnionRgn(rgn1, rgn2, dstRgn : RgnHandle);

The UnionRect procedure calculates the union of Rgn1 and Rgn2 and places it in the third region dstRgn. This procedure does not create the third region; this must be done by a call to NewRgn before using UnionRgn.

If both regions are empty, the result is also the empty region.

procedure DiffRgn(rgn1, rgn2, dstRgn : RgnHandle);

The DiffRect procedure subtracts region Rgn2 from Rgn1 and places the result in the third region dstRgn. This procedure does not create the third region; this must be done by a call to NewRgn before using DiffRgn.

If Rgn2 is empty, the result is the empty region.

procedure XorRgn(rgn1, rgn2, dstRgn : RgnHandle);

The XonRgn procedure calculates the difference between the union and intersection of Rgn1 and Rgn2, and places it in the third region dstRgn. This procedure does not create the third region; this must be done by a call to NewRgn before using UnionRgn.

If the regions are the same, that is coincident, the result is the empty region.

function RectInRgn(r : Rect; rgn : rgnHandle) : Boolean;

The RectInRgn function checks whether the given rectangle intersects the specified region, returning True if so and False otherwise.

function EqualRgn(rgn1, rgn2 : RgnHandle) : Boolean;

The EqualRgn function checks to see if the two regions are equal, returning True if they are and false otherwise. Two regions are equal if they have the same size, shape, and location. Any two empty regions are equal.

function EmptyRgn(rgn: RgnHandle) : Boolean;

The EmptyRgn function returns True if a region is empty and False otherwise.

Using Regions

The following program called HockeyShot uses regions to display a hockey goal and backboard (Figure 7.7). The object of the game is to position the puck into the goal. The player controls the puck (an oval) by moving the mouse. This determines the puck's vertical position. The horizontal position of the puck as it moves toward the goal is controlled by the program and sped up as the program progresses. The backboard, which is painted black, is defined as a rectangular region minus the goal, which is a smaller rectangular region and displayed in white.

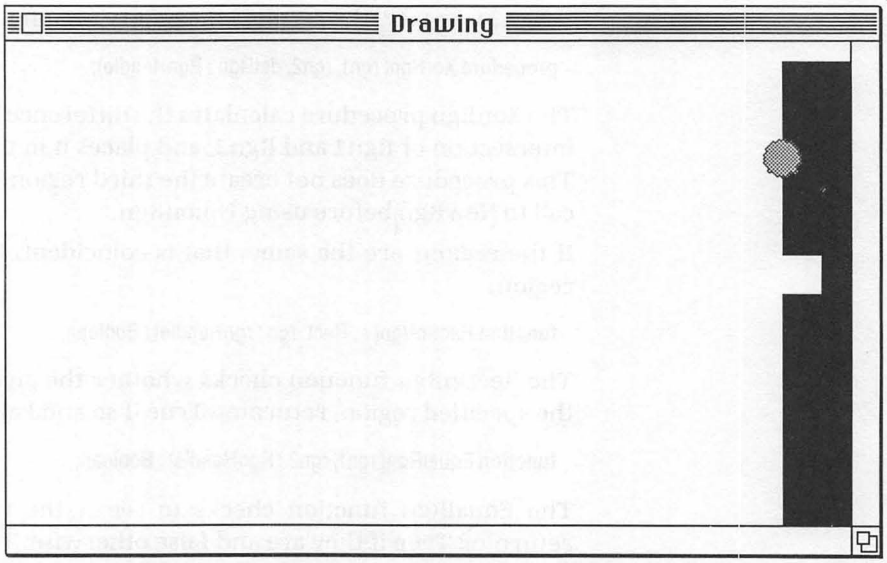


Figure 7.7 The HockeyShot program

The vertical position of the puck is determined by two consecutive calls to GetMouse. The two vertical coordinates are compared, and then the puck is moved in the direction of the mouse move.

```
program HockeyShot;
 uses
  QuickDraw2;
```

```
var
 Board, Goal : rgnHandle;
 Temp, Puck : Rect;
 OldY, dx, dy, X, Y : Integer;
 Pt : Point;
 Hit, Miss : Boolean;
 Ct : Integer;
begin
 Ct := 0;
 Hit := False;
 Miss := False;
 Board := NewRgn;
 Goal := NewRgn;
 OpenRgn; {Set up board}
 SetRect(Temp, 400, 10, 510, 250);
 FrameRect(Temp);
 CloseRgn(Board);
 OpenRgn; {Set up goal}
 SetRect(Temp, 400, 110, 420, 130);
 FrameRect(Temp);
 CloseRgn(Hole);
 DiffRgn(Board, Goal, Board);
 PaintRgn(Board);
 SetRect(Puck, 10, 10, 30, 30);
 FillOval(Puck, Gray);
 GetMouse(X, OldY); {Original mouse position}
 dx := 3;
 repeat
  Ct := Ct + 1;
  EraseOval(Puck);
  GetMouse(X, Y);
  Pt.h := X;
  Pt.v := Y;
  if RectinRgn(Puck, Goal) then
   begin
    SysBeep(30);
    Hit := True;
    SysBeep(5);
   end
 else if RectinRgn(Puck, Board) then
  begin
   Miss := True;
   SysBeep(2)
  end;
```

```
if Y > OldY then
 dy := 2;
if Y = oldY then
 dy := 0;
if Y > OldY then
 dY := -2;
if Ct mod 15 = 0 then
 dx := dx + 1; {Speed up}
 OldY := Y;
 OffSetRect(Puck, dx, dy);
 FillOval(Puck, Gray);
 until Hit or Miss;
end.
```

Polygons

The second complex shape supported by QuickDraw is the Polygon. A polygon is a closed shape consisting of straight lines drawn with the Move and MoveTo procedure (Figure 7.8). In a sense, it is a subset of a region and is defined in a similar fashion, but it has fewer procedures capable of manipulating it.

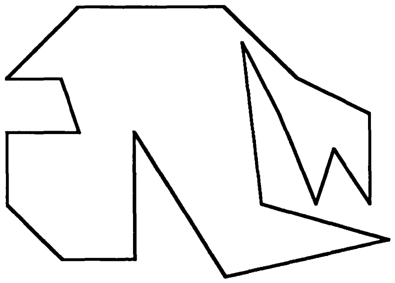


Figure 7.8 A Polygon

The polygon data type is part of the QuickDraw2 library and is defined as a record with two fixed and one variable field:

```
Polygon = record
 polySize : Integer;
 polyBBox : Rect;
 polyPoints : array [0 .. 0] of Point
end;
```

The polySize field contains the size, in bytes, of the polygon variable. The polyBBox field contains a rectangle that encloses the entire polygon. The polyPoint array is a variable-length array (expanded as necessary) containing all the end points that make up the polygon.

Since the polygon record type is a dynamic data type with varying memory needs during execution, it is accessed through a handle like a region.

```
type
 PolyPtr = ^Polygon;
 PolyHandle = ^PolyPtr;
```

To create a polygon, a call must be made to the QuickDraw routine which allocates space for the polygon record in memory, returns a handle to the record, and starts to record the points on the perimeter of the polygon. This routine is called OpenPoly.

```
function OpenPoly : PolyHandle;
```

The OpenPoly function takes no parameters and returns a handle to a new polygon. The function also tells QuickDraw to start saving the polygon definition as specified by calls to the line drawing procedures Line and LineTo. OpenPoly calls the procedure HidePen so that while the polygon is open, no drawing is performed unless you call ShowPen from inside the polygon definition and then balance it with a call to HidePen afterward.

Once OpenPoly is called, the polygon is then described with line drawing routines. For example, to define a rectangular polygon, the following code could be used.

```
PaulsPoly := OpenPoly;
 MoveTo(10, 10);
 LineTo(100, 10);
 LineTo(100, 30);
 LineTo(10, 30);
 LineTo(10, 10);
ClosePoly;
```

As you can see, the last statement in the polygon definition was terminated by the ClosePoly procedure.

```
procedure ClosePoly;
```

The ClosePoly procedure terminates the recording of the polygon definition and causes QuickDraw to calculate the polygon size and enclosing rectangle. ClosePoly also calls ShowPen to balance the HidePen originally called by OpenPoly.

Once a polygon is defined, it can then be displayed in the Drawing window with one of the Polygon drawing routines.

```
procedure FramePoly (Poly : PolyHandle);
```

FramePoly draws the outline of the polygon pointed to by the specified polygon handle. This is done by playing back the routines used to define the polygon with the pen's current pattern, mode, and size. Remember, since the pen hangs below and to the right of the pen location, the polygon's outline will extend beyond the right and bottom edges of the polygon's enclosing rectangle or boundary.

The other four polygon drawing routines should be very familiar to you by now. All of these routines draw within the polygon's boundary.

```
procedure PaintPoly( Poly : PolyHandle);
procedure ErasePoly( Poly : PolyHandle);
procedure InvertPoly( Poly : PolyHandle);
procedure FillPoly( Poly : PolyHandle; pat : Pattern);
```

When a polygon is no longer needed, it should be disposed of to reclaim the memory that it occupied. This is done with the KillPoly procedure.

```
procedure KillPoly (Poly : PolyHandle);
```

The KillPoly procedure deallocates the polygon whose handle is specified. This should only be done once the polygon is no longer needed.

The following program allows freehand drawing of a polygon in the Drawing Window. It operates by opening the polygon, waiting for the first mouse down event, using that point as the start of the polygon, and performing a MoveTo to that point. It then waits for a series of mouse down events and performs a LineTo to each mouse down location. A stop box is used to terminate the drawing (no line is drawn to the stop box), or a code to detect a double mouse click could be used. Since no drawing is normally done inside the polygon definition, ShowPen is called to display the lines being drawn. Once the polygon is completely drawn, it is then redrawn with fillPoly. Of course, HidePen is called before this.

```
program Poly1Moves;
 uses
  QuickDraw2;
 var
  Shape : Polyhandle;
  Stop : Rect;
  E : Eventrecord;
 begin
  SetRect(Stop, 10, 10, 30, 30);
  frameRect(Stop);
  repeat
   until GetNextEvent(2, E);
   GlobalToLocal(E.Where); {startpoint}
   Shape := OpenPoly;
   ShowPen;
   MoveTo(E.Where.h, E.Where.v);
   repeat
    repeat
    until GetNextEvent(2, E);
    GlobalToLocal(E.Where);
    if not ptinRect(E.Where, Stop) then
      LineTo(E.Where.h, E.Where.v);
   until ptinRect(E.Where, Stop);
   HidePen;
   ClosePoly;
   fillPoly(Shape, gray);
   KillPoly(Shape)
 end.
```

A second, more sophisticated, but not necessarily better version of the same program is presented below. This version uses an array to record the points forming the polygon and then plays them back inside the polygon definition. From this program, it is easy to get an idea of how QuickDraw replays the polygon definition when a polygon drawing routine is called.

```
program DrawPolyII;
 uses
  QuickDraw2;
 var
  Shape : Polyhandle;
  List : array[1..10] of point;
  Pt : Point;
  NumPts, Ct, X, Y : Integer;
  Stop : Rect;
  E : EventRecord;
begin
 SetRect(stop, 10, 10, 30, 30);
 frameRect(stop);
 repeat
 until GetNextEvent(2, E);
 GlobalToLocal(E.Where); {startpoint}
 List[1] := E.Where;
 MoveTo(E.Where.h, E.Where.v);
 Ct := 2;
 repeat
```

```
repeat
until GetNextEvent(2, E);
GlobalToLocal(E.Where);
if not ptinRect(E.Where, Stop) then
 begin
  List[Ct] := E.Where;
  LineTo(E.Where.h, E.Where.v);
  Ct := Ct + 1
 end
until ptinRect(E.Where, Stop);
NumPts := Ct;
Ct := 2;
Shape := OpenPoly;
MoveTo(List[1].h, List[1].v);
repeat
 LineTo(List[Ct].h, List[Ct].v);
 Ct := Ct + 1;
until Ct > NumPts;
ClosePoly;
fillPoly(Shape, gray);
KillPoly(Shape)
end.
```

Manipulating Polygons

The only polygon manipulation routine available is OffSetPoly.

```
procedure (Poly : PolyHandle; dh, dv : Integer);
```

The OffSetPoly procedure moves the polygon a distance of dh points horizontally and dv points vertically on the coordinate plane. If dh and dv are positive, the movement is to the right and down. Negative values for either of these parameters move the polygon in the appropriate opposite direction. No change of the polygon on the screen occurs, but the change will be reflected when the polygon is redrawn.

The following program adds some animation to the Poly1 program by moving the created polygon across the screen.

```
program Poly1;
 uses
  QuickDraw2;
 var
  Shape : Polyhandle;
  Stop : Rect;
  E : Eventrecord;
  K : Integer;
```

```
begin
 SetRect(Stop, 10, 10, 30, 30);
 frameRect(Stop);
 repeat
 until GetNextEvent(2, E);
 GlobalToLocal(E.Where); {startpoint}
 Shape := OpenPoly;
 ShowPen;
 MoveTo(E.Where.h, E.Where.v);
 repeat
  repeat
  until GetNextEvent(2, E);
  GlobalToLocal(E.Where);
  if not ptinRect(E.Where, Stop) then
   LineTo(E.Where.h, E.Where.v);
 until ptinRect(E.Where, Stop);
 HidePen;
 ClosePoly;
 fillPoly(Shape, gray);
 for K := 1 to 30 do
  begin
   ErasePoly(Shape);
   OffSetPoly(Shape, 10,10);
   fillPoly(Shape, gray);
  end;
 KillPoly(Shape)
end.
```

Defining Custom Patterns

We have already seen the five built-in QuickDraw patterns: White, Black, Gray, ltGray, and dkGray. However, programmers are free to define their own patterns to be used for purposes such as the pen pattern and in routines like FillRect.

The pattern data type is defined by QuickDraw as:

```
type
 Pattern : packed array[0 .. 7] of 0 .. 255;
```

This is essentially an 8-element array with each element holding a value from 0 to 255. The definition represents the basic structure of a pattern which is an 8 pixel by 8 pixel grid (Figure 7.9). Since a pixel needs one bit to be represented and the maximum value of 8 bits is decimal 255, the array structure works just fine.

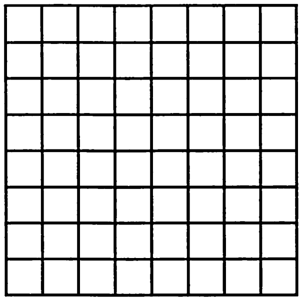


Figure 7.9 8-by-8 pixel grid

If you see some similarity between this grid and the GridEdit program, it is not just a coincidence. The PatternMaker program that follows is based upon our old friend GridEdit from Chapter 5. Some major extensions must be added to GridEdit to adapt it to this task. Needed is the ability to determine a hexadecimal value based on which pixels are black and the ability to place those values into a pattern data structure.

Let's remove the latter of these obstacles first. We could assign each element in the array the appropriate hexadecimal value, but the program would be easier to adapt for other uses if we use the StuffHex procedure.

```
procedure StuffHex(Pointer : QDPtr; S : string);
```

The StuffHex procedure pokes the hexadecimal values held as characters in the string S into the structure pointed to by Pointer. Pointer is a QDPtr which is essentially a pointer to any structure. StuffHex performs no type checking at all and it is easy to "mess up" a program with it. Use it carefully!

Converting the black and white pixels in the grid to hexadecimal values is a relatively simple task. But first, a quick review of hexadecimal. The hexadecimal number system, base 16, is often used as a shorthand notation for binary because there is a straightforward relationship between the two number systems. Table 7.1 summarizes the corresponding decimal, binary and hexadecimal values.

Since 4 binary digits go into forming a single hexadecimal digit, each row of the grid is represented by 2 hexadecimal digits. Each pixel in a hexadecimal digit represents a different power of two. For instance, in the first row, the pixels represent the following power of two.

$2^3\ 2^2\ 2^1\ 2^0\ 2^3\ 2^2\ 2^1\ 2^0$

Table 7.1

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

The array, which maintains the value of the pixel as a Boolean value, will be scanned, four positions at a time, calling a procedure named Convert to add the current pixel value to a hexadecimal value being determined.

```
procedure Convert (Pt : Boolean);
   Base : Integer;
   var Sum : Integer);
begin
 if Pt then
  case Base of
  3 :
   Sum := Sum + 8;
  2 :
   Sum := Sum + 4;
  1 :
   Sum := Sum + 2;
  0 :
   Sum := Sum + 1;
 end; case
end;
```

Once four pixels have been converted in a hexadecimal number, this number must be converted to a character value which will be concatenated into the string sent to the StuffHex procedure. It is easy to convert a single decimal digit into a character by adding to it the ORD of the character '0'.

```
ORD(digit) + ORD('0');
```

However, to convert a single hexadecimal value, the digits greater than nine must be accounted for. This can be done by adding to the character 'A' the difference between the digit and 10. The function HexString performs this function.

```
function HexString (N : Integer) : Char;
 begin
  if N < 10 then
   HexString := Chr(ord(N) + Ord('0'))
  else
   begin
    N := N - 10;
    HexString := Chr(Ord('A') + N);
   end
 end;
```

Once you create a pattern with the grid, you click inside the Stop rectangle and the program scans the grid to create the hexadecimal string to be stuffed.

```
for Row := 1 to GridSize do
 begin
  Sum := 0;
  Base := 3;
  for Col := 1 to GridSize do
   begin
    Convert(Mark[Col, Row], Base, Sum);
    Base := Base - 1;
    if Base < 0 then
     begin
      StuffStr := ConCat(StuffStr, HexString(Sum));
      Base := 3;
      Sum := 0
    end
  end
end;
```

The hexadecimal string is then stuffed into a variable of type Pattern, and the pattern is displayed by filling a rectangle with it.

```
StuffHex(@NewPat, StuffStr);
FillRect(PatRect, newPat)
```

Notice that the pointer operator @ is used in front of the variable of type Pattern because StuffHex works with a pointer rather than the variable itself.

There are several other differences between GridEdit and PatternMaker, most notably the way mouse clicks inside the grid are handled. In GridEdit, the program scanned the two-dimensional array of rectangle to see where and in which rectangle the click occurred. PatternMaker uses a more efficient approach which calculates the rectangle selected by the coordinates of the cursor at the time of the click. This approach turns out to be faster when rectangles with higher coordinates are selected. This task has also been moved to a procedure for clarity.

```
procedure FindClick;
begin
 Row := Pt.v div 10 - 1;
 Col := Pt.h div 10 - 1;
 if (Row <= GridSize) or (Col <= GridSize) then
  begin
   Mark[Col, Row] := not (Mark[Col, Row]);
   if Mark[Col, Row] then
    FillRect(Grid[Col, Row], black)
    else
     begin
      FillRect(Grid[Col, Row], white);
      FrameRect(Grid[Col, Row])
     end
    end
  end;
```

The other major change is that event handling is used rather than relying on the Button and GetMouse procedures. You will find working with event handling is a more efficient and surprisingly easier programming technique. Here is the entire program together.

```
program PatternMaker;
 const
  GridSize = 8;
```

```
var
 Grid : array[1..GridSize, 1..GridSize] of Rect;
 Mark : array[1..GridSize, 1..GridSize] of Boolean;
 R : Rect;
 Sum, Base : Integer;
 Row, Col : Integer;
 Pt : Point;
 BigRect, PatRect, StopRect : Rect;
 StuffStr : string;
 NewPat : Pattern;
 E : EventRecord;
{----------------------------------------}
 function HexString (N : Integer) : Char;
 begin
  if N < 10 then
   hexString :=chr(ord(N) + ord('0'))
  else
   begin
    N := N - 10;
    HexString := Chr(Ord('A') + N);
    end
  end;
 {------------------------------------------}
  procedure Convert (Pt : Boolean;
     Base : Integer;
     var Sum : Integer);
  begin
   if Pt then
    case Base of
     3 :
      Sum := Sum + 8;
     2 :
      Sum := Sum + 4;
     1 :
      Sum := Sum + 2;
     0 :
      Sum := Sum + 1;
    end; {case}
   end;
  {------------------------------------------}
 procedure InitGrid;
 begin
  for Col := 1 to GridSize do
   for Row := 1 to GridSize do
    begin
     Mark[Row, Col] := False;
     setRect(Grid[Row, Col], 10 + Row * 10, 10 + Col * 10, 20 + Row * 10, 20
      + Col * 10);
     frameRect(Grid[Row, Col]);
    end;
  SetRect(BigRect, 10, 10, 20 + GridSize * 10, 20 + GridSize * 10);
 end;
{-------------------------------------------}
 procedure FindClick;
 begin
  Row := Pt.v div 10 - 1;
  Col := Pt.h div 10 - 1;
  if (row < = GridSize) or (Col < = GridSize) then
   begin
    Mark[Col, Row] := not (Mark[Col, Row]);
    if Mark[Col, Row] then
     FillRect(Grid[Col, Row], black)
    else
     begin
      FillRect(Grid[Col, Row], white);
      FrameRect(Grid[Col, Row]);
   end
  end
 end;
{-------------------------------------------}
begin
 StuffStr := '';
 SetRect(StopRect, 150, 150, 180, 180);
 SetRect(PatRect, 10, 150, 40, 180);
 FrameRect(StopRect);
 InitGrid;
 repeat
  if GetNextEvent(2, E) then
   begin
    GlobalToLocal(E.Where);
    Pt := E.Where;
    if ptinRect(Pt, BigRect) then
     FindClick;
   end;
```

```
until ptInRect(Pt, StopRect);
for Row := 1 to GridSize do
 begin
  Sum := 0;
  Base := 3;
  for Col := 1 to GridSize do
   begin
    Convert(Mark[Col, Row], Base, Sum);
    Base := Base - 1;
    if Base < 0 then
     begin
      StuffStr := ConCat(StuffStr, HexString(Sum));
      Base := 3;
      Sum := 0
     end
   end
 end;
StuffHex(@NewPat, StuffStr);
FillRect(PatRect, newPat)
end.
```

You will find that the PatternMaker program is easily adaptable for other similar applications such as creating bit maps and cursors.

Cursors

When working with the Macintosh one of the first features a user notices is the cursor, that little black arrow that moves with the mouse. The shape of the cursor is implemented via a QuickDraw data type and several routines that give the programmer control of the cursor's appearance and whether its displayed or not. QuickDraw's low-level, interrupt-driven routines link the cursor with the mouse location. The programmer need take no action for this to happen nor is there any way to disconnect the cursor from the mouse; they are inseparable.

The Cursor data type is declared by QuickDraw as:

```
type
  Cursor = record of
   Data : array[0..15] of Integer;
   Mask : array[0..15] of Integer;
   HotSpot : Point
end;
```

The shape of the cursor is defined by a 256-bit image arranged in a 16-by-16 bit square (Figure 7.10).

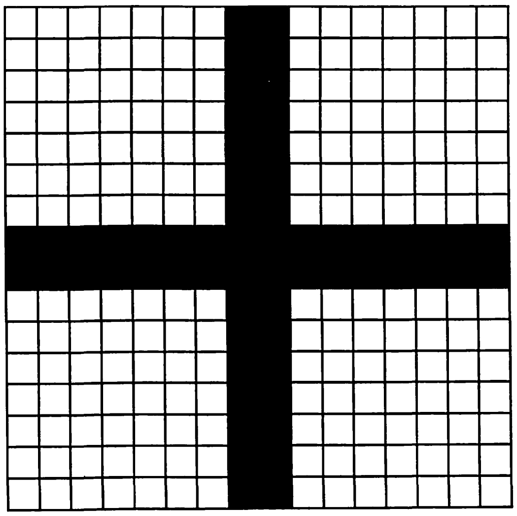


Figure 7.10 A cursor

The Data field of a cursor record holds the 256 bits defining a cursor in 16 consecutive words of memory, each 2 bytes long, accessed as a 16-position array. The Mask field is used to determine how the cursor is displayed on the screen. The Mask field is also a grid of 256 bits in a 16-by-16 array, but the value of the bits in the mask field are used to determine how the corresponding pixels in the Data field are shown on the screen. When the mask bit's value is 1, the corresponding bit the Data field will display is indicated by that bit (0 for white, 1 for black). When the mask bit's value is 0 the corresponding position in the Data array will be displayed transparent; that is, the bit will be displayed the same as the pixel that lies underneath it.

The HotSpot is the point where the cursor image will be aligned on the screen. It is expressed as a QuickDraw point.

To provide some support for programmers, QuickDraw includes a predefined cursor which is known as the north-northwest arrow. This is the cursor you are most familiar with from many applications programs and the Finder. Several other cursor shapes are included as a resource in the system resource file. QuickDraw has five procedures for handling the cursor.

```
procedure InitCursor;
```

The InitCursor procedure sets the current cursor to the predefined north-northwest arrow.

```
procedure HideCursor;
```

The HideCursor procedure removes the cursor from the screen, restores the bits underneath it, and decrements a counter, called the cursor level, which keeps track of calls to HideCursor and its complementary procedure ShowCursor. The cursor level is set to 0 by InitCursor. Every call to HideCursor should be balanced by a subsequent call to ShowCursor.

```
procedure ShowCursor;
```

ShowCursor increments the cursor level by one and if the cursor level becomes 0, displays it on the screen. A call to ShowCursor should balance every previous call to HideCursor.

```
procedure ObscureCursor;
```

The ObscureCursor procedure plays a form of hide-and-seek with the cursor, hiding it until the next time the mouse is moved. Unlike HideCursor, no call to ShowCursor is needed to balance ObscureCursor.

```
procedure SetCursor (myCursor : Cursor);
```

The SetCursor procedure sets the current cursor held in the cursor variable myCursor. If the cursor is currently hidden, its new appearance will be displayed when it's uncovered.

The GridEdit program used to define custom patterns can be easily adapted to create cursors by expanding the size of the grid and using StuffHex to place a value in both the Data and Mask field of the cursor record. The program will also need to define the HotSpot of the cursor.

Pictures

Pictures are the last and most versatile of QD's complex drawing shapes. Like Regions and Polygons, Pictures record calls to drawing routines for future playback, but unlike Regions and Polygons, there are no restrictions on what can define a Picture. For instance, text drawing routines can be included as part of Picture but not a Region or Polygon. Pictures can also be automatically scaled, stretched or shrunk into any size.

Like its counterparts, the Picture data type is a dynamic record type accessed via a handle.

```
type
 Picture = record
  picSize : Integer;
  picFrame : Rect;
  {picture definition data}
 end;
```

The Picture record type has three fields. The picSize holds the overall size of the picture record. The picFrame is a rectangle that surrounds the Picture, and unlike the structures of Regions and Polygons, is determined by the programmer and not by QuickDraw. The picFrame plays an important role in determining the scaling of a Picture. The last part of the record is the Picture definition information, which is the record of all the QuickDraw calls that make up the particular Picture and is analogous to the storing of the points of a polygon. In order to save space the definition data is stored in a coded form. Of course, since the number of drawing elements on a picture can vary dramatically, this field is allocated dynamically. A quick note on Computer Science: notice how in all of the complex drawing shapes the dynamic field is the last field of the record. This is done so the offset to this field is always known. In the case of a Picture record, it is always known that the definition data will start at the 11th byte of the record since the other two fields will always occupy 10 bytes (2 for the integer and 8 for the rectangle, which consists of four integers). The end of the Picture record can always be determined by the picSize field, which is updated by QuickDraw every time a change is made to the Picture definition.

The information in a Picture record is always maintained by QuickDraw and there is no need for a programmer to directly access it. All access to a Picture is done through a PicHandle which is defined as:

```
type
 PicPtr = ^Picture;
 PicHandle = ^PicPtr;
```

All access to a Picture is done through its handle. Since a Picture is a dynamic structure, a QuickDraw routine is needed to allocate an area in memory for the Picture and return a handle to it. This is performed by the OpenPicture function.

```
function OpenPicture (picFrame : Rect) : PicHandle;
```

The OpenPicture function is the first step in defining a picture. The function takes several actions including the allocation of an area of memory to hold the Picture record, returning a handle to that record. It also commences recording the calls to QuickDraw drawing routines in the Picture record, routines calling HidePen so that no drawing takes place on the screen while the Picture is open and storing the value of picFrame in the Picture's record.

Once a Picture is open, all drawing routines called will become part of the Picture definition but will not be directly drawn on the screen unless a call to ShowPen is made (if it is, it must be later balanced with an explicit call to HidePen). The picFrame parameter indicates to QuickDraw the rectangle that encloses the picture. This rectangle will later be used by QuickDraw to determine the ratio used to shrink or expand the picture if that becomes necessary.

Only one Picture should be open at a time in a program. The complementary QuickDraw routine to OpenPicture is ClosePicture.

```
procedure ClosePicture;
```

The ClosePicture procedure tells QuickDraw to stop saving Picture definition information. Only one call to ClosePicture should be made for each OpenPicture since they are balancing routines. ClosePicture also calls ShowPen.

Once a picture is fully defined it can be drawn on the screen with only one drawing routine DrawPicture.

```
procedure DrawPicture( myPicture : PicHandle; dstRect : Rect);
```

The DrawPicture procedure plays back the specified Picture definition, drawing it on the screen. The drawing takes place inside the given destination rectangle which specifies both where on the screen to place the Picture and what size it should be. If the destination rectangle is the same size as the Picture frame used when the Picture was opened, then the Picture is drawn on the screen the same size as it was defined. If the destination rectangle is larger or smaller than the frame rectangle, then QuickDraw calls its low-level drawing routines to shrink or expand the picture to fit into the new size.

The following program PictureRings defines a simple Picture consisting of three circle-shaped ovals and a text string. It is drawn on the screen in its original Picture frame, then in a destination rectangle slightly larger, and then in a third destination rectangle much larger and more oblong. Figure 7.11 shows the Drawing window output for the program. Notice the effect changing the size of the destination rectangle has on the shape of the ovals and the size of the text.

```
program PictureRings;
 uses
  QuickDraw2;
 var
  Pic : PicHandle;
  Ov, R : Rect;
```

```
begin
  SetRect(R, 10, 10, 80, 100);
  Pic := OpenPicture(R);
  SetRect(Ov, 40, 40, 60, 60);
  FrameOval(Ov);
  SetRect (Ov, 50, 50, 70, 70);
  FrameOval(Ov);
  SetRect (Ov, 60, 40, 80, 60);
  FrameOval(Ov);
  MoveTo(40, 90);
  DrawString('Our Picture');
  ClosePicture;
  DrawPicture(Pic, R);
  SetRect (R, 100, 100, 180, 200);
  DrawPicture(Pic, R);
  SetRect (R, 100, 100, 280, 300);
  DrawPicture(Pic, R);
  KillPicture(Pic)
end.
```

The program ends with a call to the KillPicture procedure.

```
procedure KillPicture (myPicture : PicHandle);
```

The KillPicture procedure frees up the memory space that was occupied by the Picture record pointed to by the specified Picture handle. Use this routine only when you are completely finished with a Picture and have no more use for it, but use it.

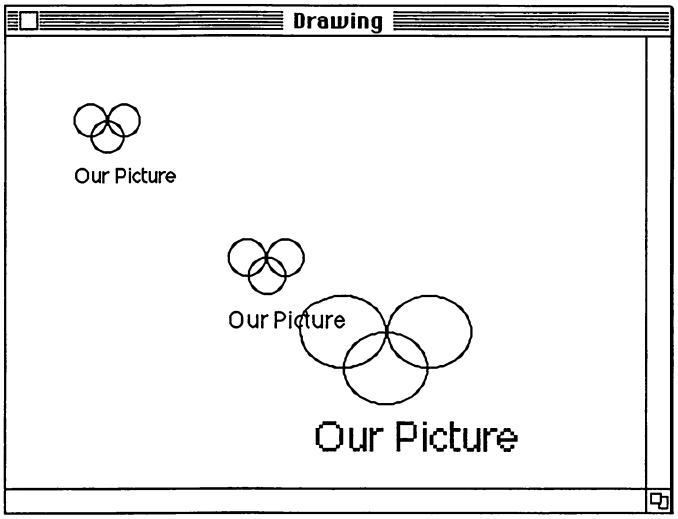


Figure 7.11 The output of program PictureRings

Calculations with Points

QuickDraw has a number of additional procedures and functions handy in many applications. A set of routines similar to the rectangle manipulation routines exist for points as well.

```
procedure AddPoint ( srcPoint : Point; var dstPoint : Point);
```

The AddPoint procedure adds the coordinates of srcPoint to those of dstPoint and returns the result in destPoint. For example, consider the following points.

```
srcPoint.x := 10;
srcPoint.y := 10;
dstPoint.x := 30;
stPoint.y := 40;
```

After the following call to AddPoint:

```
AddPoint(srcPoint, dstPoint);
```

The contents of dstPoint will be

```
dstPoint.x is 40
dstPoint.4 is 50
```

```
procedure SubPt(srcPoint : Point; var dstPoint : Point);
```

The SubPt procedure subtracts the coordinates of srcPoint from dst-Point and returns the result in dstPoint.

```
function EqualPt(PointA, PointB) : Boolean;
```

The EqualPoint function compares the two points and returns True if they are equal (have the same coordinates) and False otherwise.

```
procedure SetPt(var Pt : Point; H, V : Integer);
```

The SetPt procedure assigns the specified coordinates to the specified point. The following call to SetPoint:

```
SetPoint(APoint, 30, 46);
```

is equivalent to the following two assignment statements:

```
APoint.H := 30;
APoint.V := 46;
```

```
procedure LocalToGlobal(var Pt : Point);
```

The LocalToGlobal procedure is the opposite of the GlobalToLocal procedure. LocalToGlobal takes a point in the local coordinates of the current GrafPort and returns the corresponding global coordinates of that point. The original coordinate is lost since it is overwritten by the new value.

```
function GetPixel (H, V) : Boolean;
```

The GetPixel function takes the specified coordinates as a point in local coordinates and returns the condition of the pixel that hangs down and to the right of it. If that pixel is black, the function returns True and if the pixel is white, False.

Drawing in Color

This topic may seem strange since the Macintosh comes only with a black and white display (although a very good one). The people at Apple had the foresight to include color capability in QuickDraw to support future expansions of the Macintosh and peripherals. All colors are displayed on the black and white screen as black.

```
procedure ForeColor (color : LongInt);
```

The ForeColor procedure sets the foreground color of the GrafPort to the given color. The following colors (an artistic selection) are predefined as constant by QuickDraw.

```
const
  blackColor = 33;
  whiteColor = 30;
  redColor = 205;
  greenColor = 341;
  blueColor = 409;
  cyanColor = 273;
  magentaColor = 137;
  yellowColor = 69;
```

The default background color is blackColor.

```
procedure BackColor (color : LongInt);
```

The BackColor procedure is used to set the GrafPort's background color to the given color. The same predefined colors listed for the ForeGround procedure can also be used here.

Additional Routines

The Random function returns an integer, randomly and uniformly pseudorandom in the range from − 32768 through 32767.

```
function Random : Integer;
```

The sequence of values returned depends upon the QuickDraw variable randSeed which Macintosh Pascal initializes to 1. You can restart the sequence by resetting randSeed to 1 in your program.

It is common to need a random number from a smaller range than − 32768 through 32767. This can be done by taking the **mod** of the number returned. For instance, to produce a number from 1 to 48, a random number can be found like this:

```
Ran := Abs(Random mod 48) + 1;
```

The **mod** 48 of the value returned by Random produces a number from 0 to 47 (the use of the Abs function assures that it is positive); adding 1 shifts the range from 1 to 48.

The following program uses the above technique to select six random numbers to play the New York State Lotto 48 game. Lotto 48 is a state-run lottery game in which the player selects 6 numbers out of 48 and hopes (wishes, prays) that they match the 6 numbers picked in the weekly drawing. During the week this was written, the Lotto 48 jackpot exceeded 30 million dollars. Using Macintosh Pascal to select your numbers is as good a method as any.

```
program Lotto48;
 var
  List : array[1..6] of Integer;
  Ct, I, guess : Integer;
  Flag : Boolean;
```

```
begin
 Ct := 1;
 Flag := false;
 repeat
  Guess := Random mod 48 + 1;
  for I := 1 to 6 do
   begin
    if Guess = List[I] then
    Flag := True
   end;
 if Flag = False then
  begin
   List[Ct] := Guess;
   Ct := Ct + 1;
  end;
  Flag := False;
  until Ct > 6;
  for I := 1 to 6 do
  Writeln(List[I]);
   Writeln('Good Luck!!!')
end.
```

The program uses an array to store the numbers already selected to make sure that it is not a duplicate of one already selected. The array is checked after each value is returned; if the value is not in the list, it is then added.

```
procedure BackPat ( pat : Pattern);
```

The BackPat procedure sets the background pattern used in the GrafPort to be for any QuickDraw "erase" procedure. Any pattern can be used.

```
procedure SetOrigin (h, v : Integer);
```

SetOrigin changes the origin point of the local coordinate system of the Drawing window. The h and v parameters set the coordinates.

CHAPTER

8

A Complete Application: The Logger

This chapter presents a complete application using many of the concepts and routines developed in the other chapters of this book. The application called The Logger maintains a computerized record log of computer usage including date, time, purposes, and prints—both a detailed and summary report. It might be used to track computer usage to meet the IRS standards for home computer use. The program uses events, simulated radio and pushbuttons, text files, and random file processing.

In describing a complex program it can be useful to show both the input and output of the program and then describe the processing that fits in between. The input of the logger is a simulated window displayed inside the Drawing window (Figure 8.1). It shows six radio buttons, three pushbuttons, and a text box.

Each individual log record contains two pieces of information about how the computer is being used. One is a category selected from one of the six radio buttons displayed. The categories can be easily customized for a particular purpose but initially there are four business categories along with a personal and investment category. These categories will be the basis for a summary report showing the percentage of use per category. The second part of the description is a text string describing the task that can be entered by the user. These two of the three forms of input are used to form the data that is stored in a file.

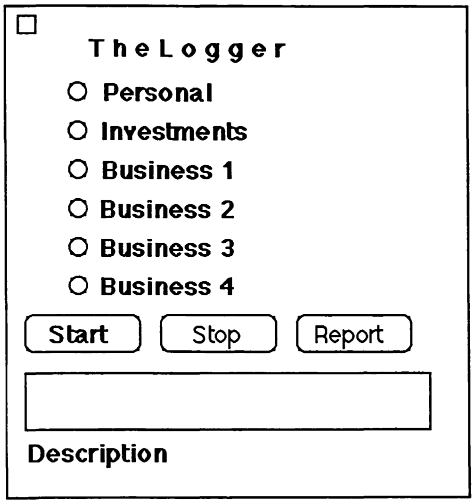


Figure 8.1 The Logger window

The third form of input is the pushbuttons that control the action of the program. Initially, the Start and Report buttons are active. Pushing the Start button causes the program to store the current information in the file along with the time and date. After Start is pressed, both Start and Report become inactive. The Report button causes the program to send to the printer the detail and summary report pictured below. A number of calculations have to be made to produce the report, and they will be described later. The Stop button is only made active after the Start button has been pressed. Stop causes the program to complete the record that was previously started by finding it at the end of the file and logging the current time.

The output of the program is pictured below in Figure 8.2.

The report is in two sections: a detail section and a usage summary. The detail section lists in columnar form all the records that have been stored by the user and the elapsed time of use. The summary section lists the total time of usage of each category and a percentage of the total time used.

The processing done can be thought of as being divided up into several parts: the handling of the mouse input and event handling, the file handling, and the report production. Of these, the mouse input will seem very familiar to you since it is based on the work done in the event handling and QuickDraw chapters. The other two parts will be discussed in greater detail.

```
                              The Logger

 Date       Start  Stop    Category      Description           Time
 3/29/86    4:14P  4:15P   Business1     Payroll               00:01
 3/29/86    4:15P  4:16P   Investments   Stock tracking        00:01
 3/29/86    4:17P  4:21P   Business3     Inventory             00:04
 3/29/86    4:22P  4:29P   Business3     Scheduling            00:07
 3/29/86    4:29P  4:31P   Personal      Invitations           00:02

Summary of usage by category
Personal       00:02    9.5%
Investments    00:02    9.5%
Business1      00:02    9.5%
Business2      00:00    0.0%
Business3      00:15   71.4%
Business4      00:00    0.0%
```

Figure 8.2 The Logger report

The basis of the entire program is the file containing the records. The file is of type LogRec defined as:

```
LogRec = record
  Date : string[8];
  StartTime, StopTime : Integer;
  Cat : CatType;
  Descript : string[25];
end;
```

The fields hold the date in a string formed by the program from the clock information, the start and stop times (obviously not posted at the same time), the description entered, and the category read from the radio buttons. The start and stop time uses only the hour and minute with the seconds discarded. In the interest of saving storage space, the hours and minutes are encoded and stored together in one integer. This is done by multiplying the hour by 100 and then adding the minutes. This is later reversed by dividing by 100 to get the hours and taking the mod of 100 to get the minutes (handy thing, this base 10 system!). Notice that the elapsed time is not stored in the record but is calculated later on by the report. This saves a few bytes in each record. The enumerated type CatType is defined as

```
CatType = (P, Iv, B1, B2, B3, B4);
```

It is more efficient to store this information as an enumerated type than a string since an enumerated value never takes more than one byte to store.

The file processing is rather simple since once a record is complete, we have no need to work with it again except to produce the report. This file is opened with the Open procedure as one of the first actions taken by the program. Using Open will allow us random access to the file which will speed processing time. When a new record is added to the file it is always placed at the end of the file. This is done by moving the file to the end of file and then placing the record. This is all done in the StartRoutine procedure.

```
procedure StartRoutine;
 begin
  GetTime(Time); {read the clock}
  with Time do
   begin
    Log.Date := Concat(StrToint(Month), '/', StrToInt(Day));
    Log.Date := Concat(Log.Date, '/', Omit(StrToInt(Year), 1, 2));
    Log.StartTime := Hour * 100 + Minute;
   end;
  Log.Descript := Te;
  Seek(LogFile, MaxInt);
  LogFile^ := Log;
  Put(LogFile);
 end; {Start Routine}
```

This procedure assumes a variable called Time of type DateTimeRec and LogFile, a file of LogRec. Of interest in the procedure is the use of Seek to find the end of file by passing a record number of MaxInt.

The complementry procedure to StartRoutine is StopRoutine, which is called after the Stop button is pressed. This procedure has to find the last record in the file, which was placed in the file without a StopTime by the StartRoutine.

```
procedure StopRoutine;
 begin
  GetTime(Time);
  Seek(LogFile, MaxInt);
  K := FilePos(LogFile);
  K := K - 1;
  Seek(Logfile, K);
  with Time do
   LogFile^.StopTime := Hour * 100 + Minute;
  Put(LogFile);
 end;
```

The StopRoutine must find the last record in the file. We saw before how to find the end of file with Seek, but Seek takes us one past the last record. To find the last record we seek the end of file, use the FilePos function to find its record number, and then seek the record that is before it.

The production of the report is more complex than the file processing. It requires printing a heading, reading the file record by record, calculating the elapsed time, printing the record, totaling the elapsed times , calculating the percentage of usage, and printing the summary. This is all handled in the PrintRoutine procedure which is called after the Report button is pressed. Let's take the tasks one at a time.

Printing the heading is not as trivial as it might seem. Since the heading uses both the bold and underlined printing modes of the ImageWriter native typeface, it requires sending control codes to the printer. This is done through a Write statement. The control codes for these printer functions are as follows:

```
Bold Face On - Chr(27), Chr(33)
Bold Face Off - Chr(27), Chr(34)
UnderlineOn - Chr(27), Chr(88)
Underline Off - Chr(27), Chr(89)
```

The heading is printed with:

```
Writeln(Out, Chr(27), Chr(33), '' : 30, 'The Logger', Chr(27), Chr(34));
Writeln(Out);
Write(Out, Chr(27), Chr(88), 'Date' : 6, '' : 4);
Write(Out, ' Start' : 6, 'Stop' : 6, '' : 2, 'Category' : 10);
Writeln(Out, '' : 4, 'Description' : 12, '' : 13, 'Time' : 5, Chr(27), Chr(89));
```

Next the file is read one record at a time, the information formatted and sent to the printer. The first thing printed is the Date field, which is simple since it was formatted before it was placed in the file. Next is the Start and Stop times. These times must be converted from the 24 hour clock to AM and PM and then properly formatted. The HrsAndMins procedure, which is contained inside of PrintRoutine, is used for this. One of the things it must do is place a leading zero in front of a one-digit minute value. HrsandMins sends the information to the printer through the Text file named Out.

```
procedure HrsandMins;
 begin
  if H = 0 then
   Write(Out, '00:');
  if H > 12 then
   Write(Out, (H - 12) : 2, ':')
  else if H > = 10 then
   Write(Out, H : 2, ':')
  else
   Write(Out, '0', H : 1);
```

```
    if M >= 10 then
     Write(Out, M : 2)
    else
     Write(Out, '0', M : 1);
    if H > 12 then
     Write(Out, 'P')
    else
     Write(Out, 'A');
  end;
```

Notice that no Writeln statements are used since all the information is printed on one line. Next to be printed is the category and description. Since the category is stored in a code, it must be converted to a string in a Case statement. The Description field is printed without modification, and then enough spaces are printed to occupy 25 columns, including the description. The elapsed time is calculated by converting the Start and Stop time into seconds and subtracting one from the other. This is later converted back into hours and minutes and printed with a version of the HrsAndMins routine that does not print the AM or PM adjective.

Here is this section of PrintRoutine.

```
H := StartTime div 100;
M := StartTime mod 100;
HrsandMins;
Write(Out, '' : 1);
ETime := 60 * H + M;
H := StopTime div 100;
M := StopTime mod 100;
HrsAndMins;
ETime := (60 * H + M) - ETime;
case Cat of
 P :
  S := 'Personal ';
 Iv :
  S := 'Investments';
 B1 :
  S := 'Business1 ';
 B2 :
  S := 'Business2 ';
 B3 :
  S := 'Business3 ';
 B4 :
  S := 'Business4 ';
end;
```

```
      I := 25 - Length(Descript); {figure padding}
      Write(Out, S, '' : 2, Descript, '' : I);
      {Calculate elapsed time}
      H := ETime div 60;
      M := ETime mod 60;
      NoAMorPM; {print it}
      Writeln(Out);
      Usage[Ord(Cat)] := Usage[Ord(Cat)] + ETime; {total times}
     end;
    end;
```

The final part of this routine prints the summary report. As each record is read from the file that the calculated elapsed time is added to a six-position array called Usage, that holds the running totals of the elapsed time for each category. The summary heading is printed, the category names printed along with the total time for that category, and the percentage of use is determined and printed.

```
    {Calculate elapsed time}
      H := ETime div 60;
      M := ETime mod 60;
      NoAMorPM;
      Writeln(Out);
      Usage[Ord(Cat)] := Usage[Ord(Cat)] + ETime;
      end;
     end;
    Writeln(Out);
    Writeln(Out, Chr(27), Chr(88), 'Summary of usage by category', Chr(27), Chr(89));
    for I := 0 to 5 do
     TotUsage := TotUsage + Usage[I];
    for I := 0 to 5 do
     begin
     case I of
     0:
      S := 'Personal ';
     1:
      S := 'Investments';
     2:
      S := 'Business1 ';
     3:
      S := 'Business2 ';
     4:
      S := 'Business3 ';
     5:
      S := 'Business4 ';
     end; {case}
```

```
      Write(Out, S, ' ');
      H := Usage[I] div 60;
      M := Usage[I] mod 60;
      NoAMorPM;
      Writeln(Out, ' ' : 2, (Usage[I] / TotUsage * 100) : 5 : 1, '%')
      end
    end;
```

Here is the entire PrintRoutine together.

```
procedure PrintRoutine;
  var
   TotUsage, I : Integer;
  procedure HrsandMins;
  begin
   if H = 0 then
    Write(Out, '00:');
   if H > 12 then
    Write(Out, (H - 12) : 2, ':')
   else if H > = 10 then
    Write(Out, H : 2, ':')
   else
    Write(Out, '0', H : 1);
   if M > = 10 then
    Write(Out, M : 2)
   else
    Write(Out, '0', M : 1);
   if H > 12 then
    Write(Out, 'P')
   else
    Write(Out, 'A');
  end;
{----------------------------------------}
  procedure NoAMorPM;
  begin
   if H = 0 then
    Write(Out, '00:')
   else if H > = 10 then
    Write(Out, H : 2, ':')
   else
    Write(Out, '0', H : 1, ':');
   if M > = 10 then
    Write(Out, M : 2)
   else
    Write(Out, '0', M : 1);
  end;
```

```
begin {Print Routine}
 K := 0;
 Writeln(Out, Chr(27), Chr(33), ' ' : 30, 'The Logger', Chr(27), Chr(34));
 Writeln(Out);
 Write(Out, Chr(27), Chr(88), 'Date' : 6, ' ' : 4);
 Write(Out, ' Start' : 6, 'Stop' : 6, ' ' : 2, 'Category' : 10);
 Writeln(Out, ' ' : 4, 'Description' : 12, ' ' : 13, 'Time' : 5, Chr(27), Chr(89));
 while not eof(LogFile) do
  begin
   Seek(LogFile, K);
   K := K + 1;
  if not eof(LogFile) then
  with LogFile^ do
   begin
   Write(Out, Date : 8, ' ' : 2);
   H := StartTime div 100;
   M := StartTime mod 100;
   Write(H, M);
   HrsandMins;
   Write(Out, ' ' : 1);
   ETime := 60 * H + M;
   H := StopTime div 100;
   M := StopTime mod 100;
   HrsAndMins;
   Write(Out, ' ' : 3);
   ETime := (60 * H + M) - ETime;
 case Cat of
  P :
   S := 'Personal ';
  Iv :
   S := 'Investments';
  B1 :
   S := 'Business1 ';
  B2 :
   S := 'Business2 ';
  B3 :
   S := 'Business3 ';
  B4 :
   S := 'Business4 ';
  end;
```

```
   I := 25 - Length(Descript);
   Write(Out, S, ' ' : 2, Descript, ' ' : I);
 {Calculate elapsed time}
   H := ETime div 60;
   M := ETime mod 60;
   NoAMorPM;
   Writeln(Out);
   Usage[Ord(Cat)] := Usage[Ord(Cat)] + ETime;
   end;
  end;
 Writeln(Out);
 Writeln(Out, Chr(27), Chr(88), 'Summary of usage by category', Chr(27), Chr(89));
 for I := 0 to 5 do
  TotUsage := TotUsage + Usage[I];
 for I := 0 to 5 do
  begin
   case I of
   0:
    S := 'Personal ';
   1:
    S := 'Investments';
   2:
    S := 'Business1 ';
   3:
    S := 'Business2 ';
   4:
    S := 'Business3 ';
   5:
    S := 'Business4 ';
   end; {case}
  Write(Out, S, ' ');
  H := Usage[I] div 60;
  M := Usage[I] mod 60;
  NoAMorPM;
  Writeln(Out, ' ' : 2, (Usage[I] / Totusage * 100) : 5 : 1, '%')
 end
end;
```

Notice that the procedures AMorPM and NoAMorPM are declared inside of PrintRoutine and are local to it. The remainder of the program is the main section, which initializes the window and handles the mouse and keyboard input in a manner explained in earlier chapters. When the program starts, it first checks the data file to see if the last record contains a value in the StopTime field. Remember that when a StartTime is placed in the file, the StopTime is set to −1. If StopTime of the last record in the file is −1, then the program is in the context of a record having been opened but not closed, and the program should start by displaying the Stop button as active rather than the Start button.

When a keyboard event occurs, the key pressed is determined from the Message field of the event record. By anding it with 255, the ASCII code is found for the key. That code is then appended to the string TE which will be assigned to the Decript field of the LogRec. The character is also displayed on the screen in the Description box.

Here is the entire main section of the program.

```
begin
 HideAll;
 {Open files}
 Open(LogFile, 'Log.Data');
 Rewrite(Out, 'Printer:');
 Seek(LogFile, MaxInt); {Read last record}
 K := FilePos(LogFile);
 K := K - 1;
 Seek(Logfile, K);
 If LogFile^.StopTime = -1 then {StopTime ?}
  Status := Go
 else
  Status := No;
 ShowDrawing;
 Flag := False;
 {Display window}
 SetRect(Large, 10, 10, 250, 265);
 SetRect(Close, 17, 15, 27, 25);
 FrameRect(Large);
 FrameRect(Close);
 InitCursor;
 MoveTo(65, 40);
 TextFont(0); {Chicago, Chicago}
 DrawString('T h e L o g g e r');
 MoveTo(55, 60);
 DrawString('Personal');
 MoveTo(55, 80);
 DrawString('Investments');
 MoveTo(55, 100);
```

```
DrawString('Business1');
MoveTo(55, 120);
DrawString('Business2');
Moveto(55, 140);
DrawString('Business3');
Moveto(55, 160);
DrawString('Business4 ');
for K := 1 to 6 do {Do radio buttons}
 begin
  SetRect(C[K], 43, 50 + (K - 1) * 20, 53, 60 + (K - 1) * 20);
  FrameOval(C[K]);
 end;
Clicked := 1; {Turn on first button}
DoDot(Clicked);
SetRect(Start, 20, 170, 80, 190);
SetRect(Stop, 90, 170, 150, 190);
SetRect(Print, 160, 170, 220, 190);
if Status = No then {Draw buttons}
 begin
  DrawStart(0);
  DrawReport(0);
  DrawStop(1);
 end
else
 begin
  DrawStart(1);
  DrawReport(1);
  DrawStop(0)
 end;
SetRect(Descrip, 20, 200, 230, 230); {Text box}
FrameRect(Descrip);
MoveTo(20, 240);
TextFont(0);
DrawString('Description');
MoveTo(22, 220);
repeat
 If GetNextEvent(14, Event) then {process events}
  case Event.What of
   1 : {Mouse down event}
  begin
  GlobalToLocal(Event.Where);
  Mouse := Event.Where;
  {Handle push buttons}
```

```
if (PtInRect(Mouse, Start)) and (Status = No) then
 begin
  InvertRoundRect(Start, 10, 10);
  repeat
  until GetNextEvent(14, Event);
  InvertRoundRect(Start, 10, 10);
  DrawStart(1);
  DrawReport(1);
  DrawStop(0);
  Status := Go;
  StartRoutine;
end;{Start}
if (PtinRect(Mouse, Stop)) and (Status = Go) then
begin
 InvertRoundRect(Stop, 10, 10);
 repeat
 until GetNextEvent(6, Event);
 InvertRoundRect(Stop, 10, 10);
 Status := No;
 DrawStop(1);
 DrawStart(0);
 DrawReport(0);
 EraseRect(Descrip);
 FrameRect(Descrip);
 Te := '';
 MoveTo(22, 220);
 StopRoutine
end; {Stop}
 if PtinRect(Mouse, Print) and (Status = No) then
  begin
   InvertRoundRect(Print, 10, 10);
   repeat
   until GetNextEvent(6, Event);
   InvertRoundRect(Print, 10, 10);
   {Call Print routine}
   PrintRoutine
   end;{Print}
   {Handle radio buttons}
   if Status = No then
   for K := 1 to 6 do
   if PtInRect(Mouse, C[K]) then
   begin
    DoDot(Clicked);
    DoDot(K);
    Clicked := K;
   end;
   Log.Cat := P; {Store category}
   for K := 1 to Clicked - 1 do
   Log.Cat := Succ(Log.Cat);
   if PtInRect(Mouse, Close) then
   begin
   Flag := True;
   HideAll
   end
   end;
   2 : {Untrapped Mouse Up}
   ;
   3 : {Keyboard event}
   begin
   T := Chr(BitAnd(Event.Message, 255));
   if Ord(t) < > 8 then
   begin
   DrawChar(T);
   Te := Concat(Te, T)
   end
   else
   DrawChar(chr(8))
   end
   end;{case}
 until flag;
end.
```

Finally, here is the entire Logger program in one place.

```
program TheLogger;
 type
  State = (Go, No);
  CatType = (P, Iv, B1, B2, B3, B4);
  LogRec = record
    Date : string[8];
    StartTime, StopTime : Integer;
    Cat : CatType;
    Descript : string[25];
   end;
 var
  Time : DateTimeRec;
  Status : State;
  Log : LogRec;
  LogFile : file of LogRec;
  Descrip, Start, Stop, Print, Large, Close : Rect;
  C : array[1..6] of Rect;
  Flag : Boolean;
  Mouse : Point;
  Event : EventRecord;
  Clicked, I, K : Integer;
  H, M, ETime : Integer;
  T : Char;
  S, Te : string;
  Out : Text;
  Usage : array[0..5] of integer;
{----------------------}
 function IntToStr (Int : Integer) : string;
  var
    Str : string;
    R : Integer;
 begin
  Str := '';
  while Int > 0 do
    begin
     R := Int mod 10;
     Str := Concat(CHR(R + 48), str);
     Int := Int div 10;
    end;
  IntToStr := Str
 end;
{----------------------}
```

```
 procedure StartRoutine;
 begin
  GetTime(Time);
  with Time do
   begin
    Log.Date := Concat(IntToStr(Month), '/', IntToStr(Day));
    Log.Date := Concat(Log.Date, '/', Omit(IntToStr(Year), 1, 2));
    Log.StartTime := Hour * 100 + Minute;
   end;
  Log.Descript := Te;
  Log.StopTime := -1;
  Seek(LogFile, MaxInt);
  LogFile^ := Log;
  Put(LogFile);
 end; {Start Routine}
{---------------------------}
 procedure StopRoutine;
 begin
  GetTime(Time);
  Seek(LogFile, MaxInt);
  K := FilePos(LogFile);
  K := K - 1;
  Seek(Logfile, K);
  with Time do
   LogFile^.StopTime := Hour * 100 + Minute;
  Put(LogFile);
 end;
{--------------------------------------------}
 procedure PrintRoutine;
  var
   TotUsage, I : Integer;
  procedure HrsandMins;
  begin
   if H = 0 then
    Write(Out, '00:');
   if H > 12 then
    Write(Out, (H - 12) : 2, ':')
   else if H >= 10 then
    Write(Out, H : 2, ':')
   else
    Write(Out, '0', H : 1);
   if M >= 10 then
    Write(Out, M : 2)
   else
    Write(Out, '0', M : 1);
   if H > 12 then
    Write(Out, 'P')
   else
    Write(Out, 'A');
  end;
{------------------------------------}
  procedure NoAMorPM;
  begin
   if H = 0 then
    Write(Out, '00:')
   else if H >= 10 then
    Write(Out, H : 2, ':')
   else
    Write(Out, '0', H : 1, ':');
   if M >= 10 then
    Write(Out, M : 2)
   else
    Write(Out, '0', M : 1);
  end;
 begin
  K := 0;
  Writeln(Out, Chr(27), Chr(33), ' ' : 30, 'The Logger', Chr(27), Chr(34));
  Writeln(Out);
  Write(Out, Chr(27), Chr(88), 'Date' : 6, ' ' : 4);
  Write(Out, ' Start' : 6, 'Stop' : 6, ' ' : 2, 'Category' : 10);
  Writeln(Out, ' ' : 4, 'Description' : 12, ' ' : 13, 'Time' : 5, Chr(27), Chr(89));
  while not eof(LogFile) do
   begin
    Seek(LogFile, K);
    K := K + 1;
 if not eof(LogFile) then
 with LogFile^ do
 begin
  Write(Out, Date : 8, ' ' : 2);
  H := StartTime div 100;
  M := StartTime mod 100;
  HrsandMins;
  Write(Out, ' ' : 1);
  ETime := 60 * H + M;
  H := StopTime div 100;
  M := StopTime mod 100;
  HrsAndMins;
  Write(Out, ' ' : 3);
  ETime := (60 * H + M) - ETime;
  case Cat of
  P :
   S := 'Personal ';
  Iv :
   S := 'Investments';
  B1 :
   S := 'Business1 ';
  B2 :
   S := 'Business2 ';
  B3 :
   S := 'Business3 ';
  B4 :
   S := 'Business4 ';
  end;
  I := 25 - Length(Descript);
  Write(Out, S, ' ' : 2, Descript, ' ' : I);
 {Calculate elapsed time}
  H := ETime div 60;
  M := ETime mod 60;
  NoAMorPM;
  Writeln(Out);
  Usage[Ord(Cat)] := Usage[Ord(Cat)] + ETime;
 end
end;
```

```
  Writeln(Out);
  Writeln(Out, Chr(27), Chr(88), 'Summary of usage by category', Chr(27), Chr(89));
  for I := 0 to 5 do
   TotUsage := TotUsage + Usage[I];
  for I := 0 to 5 do
   begin
    case I of
    0:
    S := 'Personal ';
    1:
    S := 'Investments';
    2:
    S := 'Business1 ';
    3:
    S := 'Business2 ';
    4:
    S := 'Business3 ';
    5:
    S := 'Business4 ';
    end; {case}
    Write(Out, S, ' ');
    H := Usage[I] div 60;
    M := Usage[I] mod 60;
    NoAMorPM;
    Writeln(Out, ' ' : 2, (Usage[I] / Totusage * 100) : 5 : 1, '%')
   end
 end;
{---------------------------------------}
 procedure DrawStart (Fnt : Integer);
 begin
  TextFont(Fnt);
  EraseRoundRect(Start, 10, 10);
  FrameRoundRect(Start, 10, 10);
  MoveTo(30, 185);
  DrawString('Start')
 end;
{---------------------------------------}
```

```
 procedure DrawStop (Fnt : Integer);
 begin
  TextFont(Fnt);
  EraseRoundRect(Stop, 10, 10);
  FrameRoundRect(Stop, 10, 10);
  MoveTo(100, 185);
  DrawString('Stop')
 end;
{-------------------------------------------}
 procedure DrawReport (Fnt : Integer);
 begin
  TextFont(Fnt);
  EraseRoundRect(Print, 10, 10);
  FrameRoundRect(Print, 10, 10);
  Moveto(165, 185);
  DrawString('Report');
 end;
{-------------------------------------------}
 procedure DoDot (K : Integer);
 begin
  InsetRect(C[K], 2, 2);
  InvertOval(C[K]);
  InsetRect(C[K], -2, -2)
 end;
{-------------------------------------------}
begin
 HideAll;
 {Open files}
 Open(LogFile, 'Log.Data');
 Rewrite(Out, 'Printer:');
 Seek(LogFile, MaxInt); {Read last record}
 K := FilePos(LogFile);
 K := K-1;
 Seek(Logfile, K);
```

```
if LogFile^.StopTime = -1 then {StopTime ?}
 Status := Go
else
 Status := No;
ShowDrawing;
Flag := False;
{Display window}
SetRect(Large, 10, 10, 250, 265);
SetRect(Close, 17, 15, 27, 25);
FrameRect(Large);
FrameRect(Close);
InitCursor;
MoveTo(65, 40);
TextFont(0); {Chicago, Chicago}
DrawString('T h e L o g g e r');
MoveTo(55, 60);
DrawString('Personal');
MoveTo(55, 80);
DrawString('Investments');
MoveTo(55, 100);
DrawString('Business1');
MoveTo(55, 120);
DrawString('Business2');
Moveto(55, 140);
DrawString('Business3');
Moveto(55, 160);
DrawString('Business4 ');
for K := 1 to 6 do {Do radio buttons}
 begin
  SetRect(C[K], 43, 50 + (K-1) * 20, 53, 60 + (K-1) * 20);
  FrameOval(C[K]);
 end;
Clicked := 1; {Turn on first button}
DoDot(Clicked);
SetRect(Start, 20, 170, 80, 190);
SetRect(Stop, 90, 170, 150, 190);
SetRect(Print, 160, 170, 220, 190);
if Status = No then {Draw buttons}
 begin
  DrawStart(0);
  DrawReport(0);
  DrawStop(1);
 end
```

```
else
 begin
  DrawStart(1);
  DrawReport(1);
  DrawStop(0)
 end;
SetRect(Descrip, 20, 200, 230, 230); {Text box}
FrameRect(Descrip);
MoveTo(20, 240);
TextFont(0);
DrawString('Description');
MoveTo(22, 220);
repeat
 if GetNextEvent(14, Event) then
  case Event.What of
  1 : {Mouse down event}
  begin
   GlobalToLocal(Event.Where);
   Mouse := Event.Where;
   {Handle push buttons}
   if (PtInRect(Mouse, Start)) and (Status = No) then
   begin
    InvertRoundRect(Start, 10, 10);
    repeat
    until GetNextEvent(14, Event);
    InvertRoundRect(Start, 10, 10);
    DrawStart(1);
    DrawReport(1);
    DrawStop(0);
    Status := Go;
    StartRoutine;
   end;{Start}
 if (PtinRect(Mouse, Stop)) and (Status = Go) then
  begin
   InvertRoundRect(Stop, 10, 10);
  repeat
  until GetNextEvent(6, Event);
  InvertRoundRect(Stop, 10, 10);
  Status := No;
  DrawStop(1);
  DrawStart(0);
  DrawReport(0);
  EraseRect(Descrip);
  FrameRect(Descrip);
  Te := '';
  MoveTo(22, 220);
  StopRoutine
 end; {Stop}
 if PtinRect(Mouse, Print) and (Status = No) then
  begin
   InvertRoundRect(Print, 10, 10);
   repeat
   until GetNextEvent(6, Event);
   InvertRoundRect(Print, 10, 10);
   {Call Print routine}
   PrintRoutine
  end;{Print}
 {Handle radio buttons}
  if Status = No then
   for K := 1 to 6 do
    if PtInRect(Mouse, C[K]) then
     begin
      DoDot(Clicked);
      DoDot(K);
      Clicked := K;
     end;
  Log.Cat := P; {Store category}
  for K := 1 to Clicked - 1 do
   Log.Cat := Succ(Log.Cat);
  if PtInRect(Mouse, Close) then
   begin
    Flag := True;
    HideAll
   end
  end;
```

```
      2 : {Untrapped Mouse Up}
      ;
      3 : {Keyboard event}
      begin
      T := Chr(BitAnd(Event.Message, 255));
      if Ord(t) 8 then
        begin
          DrawChar(T);
          Te := Concat(Te, T)
        end
      else
        DrawChar(chr(8))
      end
    end;{case}
  until flag;
end.
```

CHAPTER

9

The Standard Apple Numerical Environment (SANE)

SANE is not a description of a Macintosh owner but rather the acronym for the Standard Apple Numerical Environment, SANE, which is based on the Institute of Electrical and Electronic Engineer's (IEEE) Standard 754 for Binary Floating-Point Arithmetic and is an attempt by Apple to bring standardized numerical operations to a range of products. The IEEE standard specifies data types and conversion and manipulation routines for the data types. SANE supports the IEEE standard along with adding additional data types and functions.

SANE is probably the most underrated feature of the Macintosh, giving the programmer accuracy surpassing the mainframe computers of the early 1960's. Because of the incredible precision of SANE, the Macintosh becomes equally suitable for applications ranging from general ledger systems to computer simulated decays of subatomic particles.

Like the Toolbox and QuickDraw, SANE is supported by Macintosh Pascal through both direct implementation and the use of a library. For instance, the Macintosh real data types are direct implementations of the SANE data types and require no effort by the programmer to use; but to use routines belonging to SANE, the SANE library must be first called with:

```
uses
SANE;
```

This chapter provides an overview of SANE and descriptions of the SANE data types and routines with the intention of making the Macintosh Pascal programmer aware of its capabilities and the consequences of its use. Because the complexities of SANE are more applicable to the numerical analyst than the programmer, a complete description of SANE error codes, data type conversion, and numerical environment controls has been omitted. Those interested in these SANE topics should consult the Standard Apple Numeric Environment manual part of several publications from Apple Computer.

Data Types

SANE is primarily interested in real data types since the accuracy of a real number is limited by its physical representation inside the computer, and it is real values that are of greatest use for scientific and mathematical applications. There can be some confusion between the use of the term real to mean a floating point value with a fractional component and Pascal's Real data type. In this discussion, the term real will be used to cover all real data types of which the Pascal Real type is only one.

When working with reals, errors are introduced in arithmetic operations from the inability of a computer to store irrational fractions (those with an infinite number of decimal points) in a finite number of bits. Certain fractions that are rational in base ten are rational when converted to binary and can be stored exactly in the computer. Other rational decimal fractions are irrational when converted to binary. For instance, the decimal fraction 0.5_{10} can be accurately represented in binary, by 0.1_2, but the decimal fraction 0.1_{10} is a repeating fraction in binary, represented as $0.00011001100..._2$. Thus the larger the capacity of the real data types being used, the higher the accuracy of the representation of the value. While the programmer may be familiar with only the Pascal Real data type, SANE and Macintosh Pascal implement four different real data types that have varying degrees of accuracy. In order to understand the differences between them, it is important to understand how any real number is stored in any computer.

Reals are stored by dividing a memory location into three separate fields that store the numbers sign, its fractional component (sometimes called the mantissa), and the power of two that the fractional component is raised to. Figure 9.1 shows the form of such a generic real.

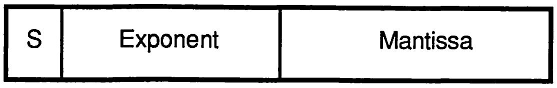


Figure 9.1 The generic real data types

The value stored is represented as

$\pm$mantissa * 2^{exponent}

As was previously mentioned, SANE supports four real data types of which three fall into the form of this generic real. What differentiates the three is the total number of bits used to represent the number. The SANE data types have been given different names by Macintosh Pascal, but either name will be recognized in your program. The data type you are probably most familiar with from Macintosh Pascal is the Real type, also called Single by SANE, which is a 32-bit type devoting 1 bit for the sign, 24 bits for the mantissa, and 7 bits for the exponent. The second real type is called Double by both Macintosh Pascal and SANE. Double (probably for double precision) is a 64-bit data type using 1 bit for the sign of the number, 53 bits for the number's mantissa part, and 10 bits for the exponent. The third type, called Extended by both Macintosh Pascal and SANE, is a whopping 80-bit data type using 1 bit for the sign, 64 bits for the mantissa and 15 bits for the exponent.

Figure 9.2 represents these three real types.

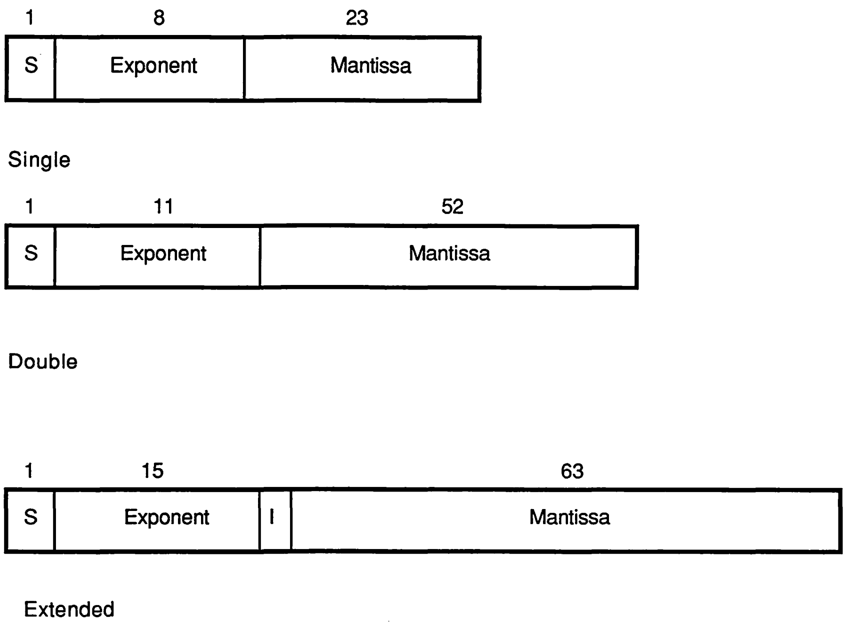


Figure 9.2 The single, double, and extended data types

A fourth real number type, called Comp by SANE and Computational by Macintosh Pascal, is organized differently. The Comp data type represents a real number as just an integral value without using a mantissa and exponent. This is intended for accounting-type applications in which dollars and cents values can be stored exactly without any worry of error introduced when converting to mantissa/exponent type format. Comp is implemented as a 64-bit type where 1 bit is used for the sign and the remaining 63 bits are used to store the integral value which can be as large as $2^{63} - 1$. The format of the Comp data type is shown in Figure 9.3.

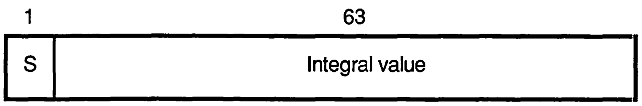


Figure 9.3 The Comp data type

When working with the Comp data type a decimal point has to be assumed by the program wherever it is appropriate. The decimal point can be displayed by dividing the value by its scaling factor in a Writeln Statement. For instance, if the value of the Comp variable named Interest represents dollars and cents, it can be printed with a decimal point between the second and third digits with the following Write statement.

```
Write(Interest/100);
```

The Macintosh Pascal manual speaks of a special form of the Write and Writeln statements designed to work with the Comp data type. This remains an unimplemented feature of the language and is best done with the above mentioned technique.

The precision of the four real number data types are summarized in Table 9.1.

The maximum and minimum negative values are the negative of the values shown.

Table 9.1

	Single	Double	Comp	Extended
Size(bits)	32	64	64	80
Exponent Range				
Minimum	− 126	− 1022	______	− 16383
Maximum	127	1023	______	16384
Mantissa Precision				
Bits	24	53	63	64
Decimal				
Digits	7-8	15-16	18-19	19-20
Approximate Decimal Range				
Minimum	1.5E − 45	5.0E − 324	0	1.9E − 4951
Maximum	3.4E + 38	1.7E + 308	9.2E + 18	1.1E + 4932

Arithmetic Operations

Three of these real data types—Single, Double and Comp—are known by SANE as application data types, but Extended is known as an arithmetic data type. This is because of the way SANE handles all real arithmetic operations. SANE performs all arithmetic operations on Single, Double, and Comp types by first converting the values to Extended values. There is no loss in precision introduced since any value of these three types can be fully represented as an Extended. The result of the arithmetic operation is then returned as an Extended value that is converted to the real type needed by the expression. Through this scheme, operations can be performed with mixes of all three data types with equal precision, and storage space can be saved by declaring variables of the smallest data type that can hold the value being used. The following example will help to clarify this operation.

```
program Example;
 uses SANE;
 var
  S : Single;
  D : Double;
  E : Extended;
   .
   .
 S := S + D * E;
```

The preceding program uses three variables, each of different real types. Notice that the SANE data type names are used, but Real could have been substituted for Single. The expression

```
S := S + D * E
```

will cause S and D to be converted to Extended values to be used in the expression. The expression will then compute an Extended value which will be converted to Single to be stored in the variable S. This presents no problem as long as the value computed by the expression can be represented by the Single data type. Should it be too large or small, an error will occur.

Selecting a Data Type

How should you select which of the real data types to use in your programs? The question is one of tradeoffs. If a value can be represented by an integer, it should since integer numbers are what any computer does best and most efficiently. If a real number is needed, the programmer must look at the precision and range needed to represent the values being worked with. It is important to keep in mind, however, that the more precise a data type, the larger the amount of storage that is needed. Speed is also a factor to be considered but one with surprising consequences. The following program was used to benchmark the execution time of the same program using the different real data types. The program does repeated evaluation of an expression having four different operators.

```
program Benchmark;
 var
  X, Y : Real;
  Start, Stop, K : Integer;
begin
 Start := TickCount;
 for K := 1 to 500 do
 X := 2 * Y / 18 + Y;
 Stop := TickCount;
 Writeln((Stop - Start))
end.
```

Execution was timed by recording the Macintosh's clock with the TickCount function, which holds 1/60th of a second intervals since system startup. The results show the best execution time when X and Y are declared to be Extended. This is apparently because no conversion has to be done, either prior to the operation or after, to perform the assignment statement. The results are:

Type	Tick Counts
Single	192
Double	193
Extended	159
Comp	191

The program using Extended variable was a full 20% faster than its nearest competitor. This would tend to lead to the conclusion that Extended should always be used. This might very well be true except when you are worried about storage space, and most programmers are. Here is the same chart with the inclusion of the storage requirements for just the two variables X and Y.

Type	Tick Counts	Storage Bytes
Single	192	8
Double	193	16
Extended	159	20
Comp	191	8

So Comp, while being about 21% faster to execute than Single, also occupies 250% more storage space. It turns out that no generalizations can be made about which data type to use; it must be considered and decided by the programmer on a case-by-case basis. Incidentally, the same program using Integers executed in 104 ticks and with LongInts, 108 ticks.

Mathematical Functions

The SANE library contains a number of mathematical functions which can be used to supplement the standard Pascal built-in functions. These functions help to rectify the deficiencies of standard Pascal in this area. Each of these functions will accept any of the real types, convert them to Extended for calculating the result, and then coerce the answer to the type needed in the expression.

Logarithm Functions

Sane provides three logarithm functions.

The Base 2 logarithm function:

```
function Log2(X : Extended) : Extended
```

This function computes the logarithm base 2 of X.

The natural logarithm function:

```
function Ln(X : Extended) : Extended
```

This function computes the logarithm base e of X.

The natural logarithm function of the argument plus 1

```
function Ln1(X : Extended) : Extended
```

This function computes the logarithm base e of the argument plus 1 or the equivalent of Ln(X + 1). The advantage of using this function instead of Ln(X + 1) is that it is more accurate when the argument is a small value, such as an interest rate.

Exponential Functions

Along with the Logarithm function, the SANE library includes a set of exponential functions.

The base 2 exponential function:

```
function Exp2(X : Extended) : Extended
```

This function computes the value of 2x.

The base e exponential function:

```
function Exp(X : Extended) : Extended
```

This function computes the value of e^x.

The base e exponential minus 1 function:

```
function Exp1(X : Extended) : Extended
```

This function computes the value of $e^x - 1$. When the value of X is very small, Exp1(X) is more accurate than the straightforward computation of Exp(X) – 1.

The Integer exponential function.

```
function XpwrI(X : Extended; I : Integer) : Extended
```

This function computes the value of X^I when X is a real value and I is an integer.

The general exponential function.

```
function XpwrY(X, Y : Extended) : Extended
```

This function is similar to XpwrI except that the power the value is raised to is a real value, not an integer.

Financial Functions

The SANE library contains two functions meant to save programmers effort while doing this type of computing.

The Annuity function:

```
function Annuity (R, N : Extended) : Extended
```

The Annuity function computes $\text{Annuity}(R,N) = \frac{1 - (1 + R)^{(-N)}}{R}$

R is the interest rate and N is the number of periods. The Annuity function is more accurate than the straightforward computation of the expression.

The compound interest function:

```
function Compound (R, N :Extended) : Extended
```

The Compound function computes $(1 + R)^N$ where R is the interest rate and N is the number of periods. When R is small this function returns a more accurate answer than the straightforward computation of the compound interest formula.

Random Number Generation

```
function Random(var X : Extended) : Extended
```

The SANE library has its own random number generation function in addition to the one contained in QuickDraw. A sequence of pseudorandom numbers can be generated in the range from 1 to 2^{31-1} by initializing the seed X to an integer value and then making repeated calls to the function, each call to the next random number in the function. If the seed value is a real value, the results are unspecified.

APPENDIX A

Macintosh Pascal Version 2.0

In 1986, THINK Technologies, the developers of Macintosh Pascal, released an updated version of the programming language called Macintosh Pascal Version 2.0. The new version corrects some of the bugs associated with Version 1.0 and adds a few new features. Although the release did not include the increased support for the Toolbox that many had hoped for, it unexpectedly included an application shell which makes creation of stand-alone Macintosh Pascal programs possible.

The Application Shell

The Version 2.0 disk contains a second program called the PShell which is essentially the Macintosh Pascal interpreter without the user interface. The PShell can be used to create and run stand-alone programs since the user has no access to the Program window and no ability to edit it. To use the PShell a version of your program must be saved with the new *As application* option in the Version 2.0 Save As command. Once a program is saved as an application, it cannot be edited any more so be sure that a second copy of the program exists that is not saved as an application. Once an application is created it can be run with the PShell by placing them both on the same disk and then Opening the applications icon. The icon for the PShell itself cannot be opened.

When an application starts to run, it is executed as though the Go option had been selected but none of the familiar surroundings of the Macintosh Pascal environment are displayed. This means that the user sees no menus or windows. Be sure to open any windows you wish displayed with ShowText, ShowDrawing, or the appropriate InLine call if you create your own window. When the program finishes, the PShell exits directly to the finder.

If you create your own menus with InLine calls to the menu manager, you do not have to worry about saving a handle to the original menu list, since none exists, nor do you have to worry about returning the Macintosh Pascal environment to its original state.

The PShell goes a long way to ward removing one of the largest complaints about Macintosh Pascal: you could not create a program for a user without having to explain the use of the programming environment. With the PShell, the user view of a Macintosh Pascal program is just like that of any other application that you start just with a double click. Even that inconvenience can be removed by setting the application as the disks start-up program.

A second new option exists in the Save As dialog box and that is *Save As Object*. A program saved as an object is stored in Macintosh Pascal's internal format for a program and is thus loaded from the disk faster. The only drawback to saving a program as an object is that it occupies more space in this format.

Changes to Files and Devices

Version 2.0 has two new device names (TEXTWINDOW and KEYBOARD:) that can be associated with Text files. They can be used to return the Input and Output files to their default settings. TEXTWINDOW can be used only with Output and KEYBOARD: only with Input. For example:

```
Reset(Input, KEYBOARD:);
```

will restore the keyboard as the device associated with the Input file.

The device PRINTER: has also been changed in Version 2.0. Instead of always being associated with the device connected to the printer port it is now directed to whichever of the two ports is selected by the user with the Choose Printer desk accessory.

Procedural Parameters

Version 2.0 now supports the passing of functions and procedures as parameters. This is an interesting programming technique that you may not be fully familiar with. Here is a short example.

```
program ParaExample;
 procedure A (procedure X);
 begin
  Writeln('in A');
  X
 end;
 procedure B;
 begin
  Writeln('in B');
 end;
 procedure C;
 begin
  Writeln ('in C');
 end;
 function C (procedure X) : Integer;
 begin
  X;
  D := 2
  end;
 begin
  A(B);
  A(C);
  Writeln(D(C));
  Writeln(D(B))
 end.
```

The parameter list of the procedure A and the function D is a procedure declaration for the procedure X. This is the formal parameter and you will notice that there is no procedure X in the program. The name of the procedure that will be invoked when a call to X is made is passed as the actual parameter. For instance, the first line in the main program is the statement A(B): a call to the procedure A with the formal parameter of B. Inside procedure A the procedure call X will call procedure B since that was passed to the procedure from the main program. Here is the output of the program.

```
in A
in B
in A
in C
in C
2
in D
2
```

To further clarify the order of execution it might to helpful to run the program with the Step execution option.

Resources

Macintosh Pascal Version 2.0 supports the use of resources attached to a program file. This allows a programmer to use a resource editor or compiler to create resources such as icons, window definitions, and control definitions and place them in the resource fork of a program file. Resources are used to separate definitions of Toolbox features from the program for ease of programming and program maintenance. When a program is stored with the Save As option, the resources are also saved.

Memory Management

The HLock and HULock procedures are not directly supported by Macintosh Pascal Version 2.0 and do not have to be called from within an InLine call. An additional new function, not part of the Toolbox, has also been added.

```
function MHHi (Hdl : Handle) : Integer;
```

The MHHi procedure takes the data structure indirectly pointed to by the handle Hdl and moves it to the top of the application heap. The function returns 0 if the move was possible and was performed and −1 if it wasn't.

Other Changes

Program Size

The maximum size of a program that can be written in Macintosh Pascal has been expanded to approximately 750 lines on a 128K Macintosh and 2000 lines on a 512K Macintosh.

Error Handling

After an error, the context of a program is now saved, allowing the programmer to examine the values of variables. The Observe or Instant window can be used for this purpose. When using the Observe window the Enter key will force the evaluation of the expressions used. Version 2.0 is also less sensitive to errors occurring with the use of InLines. It returns to the edit/debugging environment in situations where Version 1.0 would have caused a total system error.

Relaxed Order of Declarations

Version 2.0 allows the programmer to include variable, type, label, and constant declarations in any order.

User Interface Changes

A few minor changes have been made to the user interface, allowing the programmer to use tabs to space comments, to select the font and font size of the program, and to select preferences such as the number of characters to hold in the Text window (the more stored, the less memory for a program). Preference will also let you direct the Output file to the printer, a file, or both. This ability removes the necessity to perform this action in your program.

Version 2.0 also saves the desktop setting, recording the positions, size, and order of the windows. They are restored to the last setting whenever the program is used again.

APPENDIX

B Toolbox Quick Reference

Menu Manager

Type	Name	Address	Returns
Function	GetMenuBar	$A93B	LongInt
Procedure	ClearMenuBar	$A934	
Procedure	SetMenuBar	$A93C	
Function	MenuSelect	$A93D	LongInt
Procedure	InsertMenu	$A935	
Procedure	DrawMenuBar	$A937	
Procedure	AppendMenu	$A933	
Function	NewMenu	$A934	LongInt

Window Manager

Type	Name	Address	Returns
Function	FrontWindow	$A924	LongInt
Procedure	GetWTitle	$A919	
Procedure	SelectWindow	$A91F	
Function	NewWindow	$A913	LongInt
Procedure	HideWindow	$A916	
Procedure	BringToFront	$A920	
Procedure	DragWindow	$A925	
Function	TrackGoAway	$A91E	Boolean
Function	FindWindow	$A92C	Integer
Procedure	ShowWindow	$A915	
Procedure	DisposeWindow	$A914	
Procedure	SetWRefCon	$A918	
Function	GetWRefCon	$A917	LongInt
Procedure	SetWTitle	$A91A	
Procedure	GetWTitle	$A919	

Controls

Type	Name	Address	Returns
Function	NewControl	$A9C6	LongInt
Function	FindControl	$A96C	Integer
Function	TrackControl	$A968	Integer
Procedure	SetCtrlValue	$A963	
Procedure	HiLiteControl	$A95D	
Procedure	DisposeControl	$A955	
Procedure	KillControls	$A956	
Procedure	HideControl	$A958	
Procedure	ShowControl	$A957	
Procedure	DrawControls	$A969	
Procedure	SizeControl	$A95C	
Function	GetCtlValue	$A960	Integer
Procedure	SetCtlMin	$A964	
Function	GetCtlMin	$A961	Integer
Procedure	SetCtlMax	$A965	
Function	GetCtlMax	$A962	Integer
Procedure	SetCRefCon	$A95B	
Function	GetCRefCon	$A95A	LongInt

Text Editing

Type	Name	Address	Returns
Function	NewControl	$A9C6	LongInt
Function	FindControl	$A96C	Integer
Function	TrackControl	$A968	Integer
Procedure	SetCtrlValue	$A963	
Function	TENew	$A9D2	TEHandle
Procedure	TEKey	$A9DC	
Procedure	TEClick	$A9D4	
Procedure	TEDispose	$A9CD	
Function	GetCursor	$A9B9	CursHandle
Function	GetCursor	$A9B9	CursHandle
Procedure	SetCursor	$A851	
Procedure	TEActivate	$A9D9	
Procedure	TEIdle	$A9DA	
Procedure	TECut	$A9D6	
Procedure	TECopy	$A9D5	
Procedure	TEPaste	$A9D8	
Procedure	TEDelete	$A9D7	
Procedure	TEDeactivate	$A9D9	
Procedure	TESetSelect	$A9D1	
Procedure	TESetJust	$A9DF	

APPENDIX

C

Decimal - Binary - Hexadecimal Chart

Decimal	Binary	Hexadecimal
0	0000 0000	0
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
4	0000 0100	4
5	0000 0101	5
6	0000 0110	6
7	0000 0111	7
8	0000 1000	8
9	0000 1001	9
10	0000 1010	A
11	0000 1011	B
12	0000 1100	C
13	0000 1101	D
14	0000 1110	E
15	0000 1111	F
16	0000 0000	10
17	0001 0001	11
18	0001 0010	12
19	0001 0011	13

Decimal	Binary	Hexadecimal
20	0001 0100	14
21	0001 0101	15
22	0001 0110	16
23	0001 0111	17
24	0001 1000	18
25	0001 1001	19
26	0001 1010	1A
27	0001 1011	1B
28	0001 1100	1C
29	0001 1101	1D
30	0001 1110	1E
31	0001 1111	1F
32	0010 0000	20

Index

AddPoint, 230
AND, logical, 66
Animation, 84–86
Annuity (SANE), 267
AppendMenu (Toolbox), 114, 117–121, 275
AppleTalk, network events, 61
Application events, 61
Application global space, 6
Arithmetic. *See* Standard Apple Numeric Environment (SANE)
Arrays
- creating and disposing, 10–11
- and files, 33
- pattern, 217
- in records, 53–54
- text, 165
  - conversion to string, 177

ASCII codes, cf. key scan codes, 64–65
At sign (@), pointer operations, 125, 145, 220–221

BackColor, 231
BackPat, 233
Backspace, 166
Binary number system, 219, 279–280
BInLineF (Toolbox access), 112
BitAnd, 66–67, 164
Bit masks, events, 65–70, 74–75
- mouse and shift key, 164
- TextEdit, 164

BitNot and **BitXOr**, 68
BitOr and **BitShift**, 67
BlockMove, 11
BringToFront (Toolbox), 124, 276

CharWidth, 195
Check boxes, 141
- coordinates/dimensions, 142
- new, example, 143
- *see also* Control(s)

CheckItem (Toolbox), 122
ClearMenuBar (Toolbox), 114–121, 275
Clipping
- GrafPorts, 193
- TextEdit, 155

Close, 38
ClosePicture, 228
ClosePoly, 213
CloseRgn, 203
CloseWindow (Toolbox), 140
Color, QuickDraw, 192, 231
Compiler cf. interpreter, 1–2, 4–5, 86
Compound (SANE), 267
Constants, Menu Manager, InLine access, 115
Control(s), 173, 235–236
- coordinates, 142, 146
- functions listed, 276
- HaikuWriter, complex version, 177
  - code, listed, 180–188
- handles, 145
- and timers, 147
- and windows, 147–149

Control Manager, InLine access, 140–153
- event handling and values, 143–151
- miscellaneous routines, 152–153
- pointer and title, 142
- routines, listed, 142
- types and characteristics, 141

Coordinates
- controls, 142, 144–145, 146
- local, setting, 233
- points, 230–231
- QuickDraw
  - conversion, 231
  - global cf. local, 99
  - setting, 233
  - pen, 196
- text, 155, 164
- window, 126

Cursor(s)
- pointers, 160
- QuickDraw, 224–226
- TextEdit, 159–161

Cursor (type), 224

Data structures. *See* File(s); Pointers; Records; Sets
DateTimeRec (record), 17
Decimal number system, 219, 279–280

Declarations, relaxed order, version 2.0, 273
Devices, version 2.0, 270
DiffRgn, 209
DisableItem (Toolbox), 121–122
Disk events, 61
DisposeControl (Toolbox), 142, 147–152, 276
DisposeHandle, 10
DisposeMenu (Toolbox), 120–121
DisposeRgn, 205
DisposeWindow (Toolbox), 124, 276
DragWindow (Toolbox), 124, 131–133, 276
DrawChar, 94
DrawControls (Toolbox), 142, 147–152, 276
DrawMenuBar (Toolbox), 114–121, 275
DrawPicture, 228
DrawString, 94

EmptyRgn, 209
EnableItem (Toolbox), 121–122
Enumerated types, 237
EOF, 43–47
EqualPt, 230
EqualRgn, 209
ErasePoly, 214
EraseRgn, 205
Error handling, version 2.0, 273
Event(s), 59–75
  bit masks, 69–70, 74–75
  driven programs, 70–72
    timed, 71
  importance, 59–60
  keyboard, 64–68
    bit masks, 65–68
  queue, 60
    flushing, 74–75
    reading, 68–72
    reading, nondestructive, 72–74
  records, 62–64
  text, 156
  types, 60–61
    codes, 61
  and windows, 126
EventAvail, 72
cf. **GetNextEvent,** 68
Exp, Expl and **Exp2** (SANE), 266
Exponential functions, 266–267

File(s), 33–57
  component type, 36
  as data type, 35–36
  definition, 33
  pointer, 38
  random, 41–57
    end-of-file, 43–47
    ISAM, 52–57
    opening, 41–42
    sample code (HaikuWriter), 47–52
    using, 42–43
  of records, 36
  records as arrays, 33
  sequential, 36–41
    closing, 38
    erasing, 40
    opening, 37–38
    reading, 39
    sample code, 40–41
  types of, 33, 35
  see also Records
FillOval, 82
FillPoly, 214
FillRgn, 205
FillRoundRect, 83
Financial functions, 267
FindControl (Toolbox), 142, 144–151, 276
Finder, 2
FindWindow (Toolbox), 124, 126–133, 276
Flicker, screen, 86
Floating point calculations. *See* Standard Apple Numeric Environment (SANE)
FlushEvents, 74–75
Fonts, 101–102, 155
  printer, control codes, 239
  resizing, 195
ForeColor, 231
FrameOval, 82
FramePoly, 214
FrameRect, 79
FrameRgn, 204
FrameRoundRect, 83
FrontWindow (Toolbox), 123–127, 276

Get, 39
  and **EOF,** 43–47
GetCRefCon (Toolbox), 142, 152, 276
GetCtlMax, GetCtlMin (Toolbox), 142, 152, 276
GetCtlVale (Toolbox), 142, 152–153, 276
GetCursor (Toolbox), 154, 159–163, 277
GetHandleSize, 11
GetMenuBar (Toolbox), 114–121, 275
GetNextEvent, 68–72
  cf. **EventAvail,** 73
  masks, 69
GetPen, 200
GetPixel, 231
GetTime, 18
GetWRefCon (Toolbox), 124, 140, 276
GetWTitle (Toolbox), 124–133, 140, 276

GlobalToLocal, 99, 144–145, 164
Goto, 91
GrafPorts, 191–193
  clipping, 193
  record type, 192

Handles, 8–9
  allocating, 9–10
  controls, 145
  disposing, 10–11
  menu, 118
  polygons, 213
  regions, 202
  text, 155, 165
  version 2.0, 272
Heap, 6–12
  allocating handles, 9–10
  copying/moving data, 11
  dangling pointers, 12
  disposing handles, 10–11
  fragmentation/compaction, 7–9
  handles, 8–9
  locking/unlocking, 12
  regions, storage, 202
  resizing, 11
  see also Memory
Hexadecimal number system, 219–220, 279–280
HideControl (Toolbox), 142, 152, 276
HideCursor, 226
HidePen, 200
HideWindow (Toolbox), 124, 128–133, 276
HiLiteControl (Toolbox), 142, 152, 276
HiLiteMenu (Toolbox), 120–121
HiWord, 120
HLock and **HULock,** 272

IEEE, 259
ImageWriter, control codes, 239
Indexed sequential access method (ISAM) files, 52–57
InitCursor, 225
InLineP (Toolbox access), 112
Input
  redirection, 270, 273
  verification
    sets, 23
    TextEdit, 165–166
InsertMenu (Toolbox), 114–121, 275
InsetOval and **InsetRect,** 106
InsetRgn, 207
Interest, compound, 267
Interpreter cf. compiler, 1–2, 4–5, 86
InvertCircle, 83
InvertOval, 82
InvertPoly, 214
InvertRgn, 205
InvertRoundRect, 83, 99
ISAM files, 52–57

Justification, 167–168

KEYBOARD: (device name), 270
Keyboard events, 60, 64–68
Key field (ISAM), 52–57
Key scan codes cf. ASCII codes, 64–65
KillControls (Toolbox), 142, 152, 276
KillPicture, 229
KillPoly, 214

Line, 201
Line drawing, 201
LineTo, 201
Linked lists, 28–31
  controls, 141
  window, 123
Linker, 1
LinLineF (Toolbox access), 112
Ln and **Ln1** (SANE), 266
Loader, 1
LocalToGlobal, 231
Locking heap, 12
Log2 (SANE), 266
Logarithmic functions, 266
Logger, The. *See* Sample code, The Logger
LoWord, 120, 175

Macintosh, overview, 2–4
Mathematics. *See* Standard Apple Numeric Environment (SANE)
Memory
  version 2.0, 272
  map, 5–6
  read-only (ROM), 2, 4
    Toolbox routines, addresses, 113
  Toolbox, InLine access, 189
  see also Heap
Menu(s)
  components, 113
  HaikuWriter, complex version, 177
    code, listed, 180–188
  handles, 118
Menu Manager, InLine access, 113–122
  checking items, 122
  constants, using, 115
  disabling items, 118, 121
  functions listed, 275
  restoring previous menu, 116
  routines
    illustrated, 114–122
    listed, 114

sample code, 118–129, 133–136
MenuSelect (Toolbox), 114–121, 275
MHHi, 272
mod, 232
Mouse
events, 60
and shift key, 164
Move, 94
MoveTo, 94

Network events, 61
New, 26
NewControl (Toolbox), 142, 276
NewHandle, 9–10
NewMenu (Toolbox), 114, 117–121, 275
NewRgn, 202
NewWindow (Toolbox), 124, 130–133, 137, 276
NOT, logical, 68
Null events, 60
Number systems, decimal/hexadecimal/ binary, 219, 220, 279–280
Numeric calculations. *See* Standard Apple Numeric Environment (SANE)

ObscureCursor, 226
OffSetPoly, 1216
OffsetRgn, 205
Open, 41–42
and **EOF,** 43–47
OpenPicture, 227
OpenPoly, 213
OpenRgn, 203
Operating system, 2, 3
Finder as, 2
memory, 5–6
OR, logical, 67
Output redirection, 270, 273
Ovals, 82–83

PaintCircle, 83
PaintOval, 82
PaintPoly, 214
PaintRgn, 204
PaintRoundRect, 83
Pascal, Macintosh
advantages, 1
application shell, 5, 269–270
error handling, 273
interpreter cf. compiler, 1–2, 4–5, 86
program size, 273
Toolbox calls, 4–5; *see also* Toolbox, InLine access
version 2.0, 5, 269–273
Pattern (type), 217
Patterns, QuickDraw, 193
background, 233
custom, 217–224
type, 217
PenMode, 199
PenNormal, 200
PenPat, 197
Pen, QuickDraw, 193, 196–200
coordinates, 196
text, 155
PenSize, 196
PenState (record), 200
Picture (type), 226
Pictures, QuickDraw, 193, 226–229
Pointers, 8, 13, 25–31
cursor, 160
dangling, 12
declaration, 25–26
and handles, 8; *see also* Handles
master, 8
nil, 27
operations on (@), 125, 145, 220–221
and records, 28–31
window, 123–125, 173
Points
QuickDraw (data type), 20–21, 230–231
typography, 194–195
Polygon (record), 212; *see also* QuickDraw
Printer control codes, ImageWriter, 239
Procedural parameters, version 2.0, 271–272
Program size, version 2.0, 273
PtInRect, 77, 79
Pushbuttons, 141, 235–236
coordinates/dimensions, 142
new, example, 143
simulated, 97–104
see also Control(s)
Put, 38
and **EOF,** 43–47

QuickDraw, 3–5, 77–110, 191–233
animation, 84–86
color, 193, 231
coordinates
conversion, 231
global cf. local, 99
setting, 233
cursors, 224–226
type, 224
flicker, avoiding, 86
freehand drawing, 87–93
GrafPorts, 191–193
clipping, 193
record type, 192

line drawing, 201
ovals, 82–83
patterns, 193
background, 233
custom, 217–224
type, 217
pen characteristics, 193, 196–200
coordinates, 196
pictures, 193, 226–229
type, 226
points, calculations with, 230–232
polygons, 193, 212–217
drawing, 214
handles, 213
manipulating, 216–217
opening and closing, 213–214
record type, 212
sample code, 214–216
principal operations, 77
pushbuttons, simulated, 97–104
QuickDraw 2 library and **uses**, 201–202, 212
radio buttons, simulated, 105–110
rectangles, 78–81
rounded, 83
regions, 193, 201–212
calculations with, 205–209
drawing, 204–205
handles, 202
opening and closing, 202–203
record type, 202
sample program, 210–212
text characteristics, 193–195
type, 194
text drawing, 94–97
variant records, 20
QuickDraw 2 library, 201–202, 212

Radio buttons, 141–144, 235
coordinates/dimensions, 142
simulated, 105–110
see also Control(s)
Random
QuickDraw, 232
SANE, 267
Random numbers, 232, 267
Range checking, 23
Rebooting, 116
Records, 13–22
arrays in, 53–54
declaration, 14–15
duplicating, 16
event, 62–64
files of, 36
pointers in, 28–31
predefined, 17
toolbox interface, 17–18, 20–21
variant, 18–22
with statement, 16–18
see also File(s)
Rectangles, QuickDraw, 78–81
rounded, 83
see also QuickDraw
RectInRgn, 209
RectRgn, 207
Regions. *See* QuickDraw
Reset, 37, 39, 40
and **EOF,** 43–47
Reset, system, 116
Resources, version 2.0, 272
Rewrite, 37, 39, 40
and **EOF,** 43–47
ROM, 2, 4
Toolbox routines, addresses, 113

Sample code
AnimaRect (animation), 85
BabysFirstWindow, 139
BullsEye (regions), 206
ButtonSim, 100–101
ButtonSim2, 103–104
ClickandShowandTell (windows), 127
ClickandTell (windows), 126
Convert (pattern), 219
DiskIndex (ISAM), 54–57
DrawPolyII, 215–216
EOF_Example1, 44–47
EventTwo, 71
Example (SANE), 263–264
FirstEvent, 70–71
GridEdit, 78–81
cf. PatternMaker, 218–221
HaikuWriter, complex version, 168–188
enhancements, 169
event handling, 172–173
listing, 180–188
memory considerations, 178
menus and controls, 177
packed arrays and type conversion, 177–178
user interface, 169, 171, 175–176
HaikuWriter, simple version (random files), 47–52
HexString, 220
HockeyShot (regions), 210–212
InsectRgn (regions), 208
LinkUp (linked list), 30
LootAtEvents, 73–74

Lotto48 (random numbers), 232
LowerToUpper, 25
MenuDemo, 118–119
MouseEvent, 72
OldExamplewithMoveAdded (moving windows), 131–132
PatternMaker, 221–224
PaulPaint (freehand drawing), 87–93
PictureRings, 228–229
Poly1, 216–217
Poly1Moves, 214–215
RadioButton, 107–110
RadioPush, 149–151
RadioRadio, 147–149
RectSeries (animation), 84
TextDemo, 157–159
TextDemo2, 161–164
ThreeCardMonte, 95–97
time of day, 239–240
WhatTime, 18
WindowsandMenus, 134–136
WindowsGalore, 128–129
WindowsWithTrackGoAway, 130
Sample code, The Logger, 235–258
controls, 235–236
event handling, 245
files
processing, 238
structure, 237
listing
complete, 249–258
main program, 245–248
print routines, 242–244
overview, 235–237
reports, 239–244
SANE. *See* Standard Apple Numeric Environment (SANE)
Scrap, 167
Screen
flicker, controlling, 86
memory, 5–6
SectRgn, 207
Seek, 42
and **EOF,** 43–47
SelectWindow (Toolbox), 124, 127–133, 276
SetCRefCon (Toolbox), 142, 152, 276
SetCtlMax and **SetCtlMin** (Toolbox), 142, 152, 276
SetCtrlValue (Toolbox), 142, 146–151, 276
SetCursor (Toolbox), 154, 160–163, 226, 277
SetEmptyRgn, 207
SetHandleSize, 11
SetMenuBar (Toolbox), 114–121, 275
SetOrigin, 233
SetPenState, 200
SetPort (Toolbox), 139
SetPt, 230
SetRect, 79
and controls, 143–144
SetRectRgn, 207
Sets, 13, 22–25
input verification, 23
operations, 23–25
SetWRefCon (Toolbox), 124, 140, 276
SetWTitle (Toolbox), 124, 127–133, 140, 276
Shift key and mouse click, 164
ShowControl (Toolbox), 142, 152, 276
ShowCursor, 226
ShowPen, 200
ShowWindow (Toolbox), 124, 140, 276
SizeControl, 276
SizeOf, 9–10
Sound buffer, memory, 5–6
Speed, execution
interpreter cf. compiler, 1–2, 4–5, 86
SANE, 264–265
Toolbox, InLine access, 188
Stacks, 6
Standard Apple Numeric Environment (SANE), 259–267
arithmetic, basic, 263–265
conversion, SANE-Pascal, 265
execution speed, 264–265
mathematical functions, 265–267
cf. Pascal types, 260–261
selecting, 264–265
types, 260–265
uses, 260
Strings, type conversion from packed array, 177; *see also* Text; TextEdit, InLine access
StringWidth (Toolbox), 142
StuffHex, 218
StyleItem (type), 194
SubPt, 230
SysBeep, 166

Tag field, 20
TEActivate (Toolbox), 154, 160–163, 277
TEClick (Toolbox), 154, 157–163, 277
TECopy (Toolbox), 154, 167, 277
TECut (Toolbox), 154, 167, 277
TEDeactivate (Toolbox), 154, 167, 277
TEDelete (Toolbox), 154, 167, 277
TEDispose (Toolbox), 154, 159, 277
TEIdle (Toolbox), 154, 161–163, 277
TEKey and **TENew** (Toolbox), 154, 156–163, 277

TEPaste (Toolbox), 154, 167, 277
TESetJust (Toolbox), 154, 168, 277
TESetSelect (Toolbox), 154, 167, 277
Text
  arrays, 165
  fonts, 101–102
  QuickDraw, 94–97, 193–195
    size, 194–195
    type, 194
  spacing, 109
  and windows, 155
TextEdit, InLine access, 153–168
  clipping, 155
  coordinates, 155, 164
  cursors, 159–161
  event handling, 156
  features, 153
  functions listed, 277
  handles, 155, 165
  input verification, 165–166
  justification, 167–168
  record type, 154–156
  routines
    listed, 154
    miscellaneous, 165–166
  sample programs, 157–159, 161–164
  scrap, 167
  shift key and bit masks, 164
  text entry, 165
  window, 155
TextFace and **TextSize,** 194
TextFont, 101–102
TESTWINDOW (device name), 270
Ticks, defined, 62
Time of day, 237, 239–240
Timers and controls, 147
Toolbox, 2–5
  memory, 5–6
  records as interface, 17–18, 20–21
  ROM addresses, 113
  variant records, 20
Toolbox, InLine access
  form, 112–113
  memory limits and system considerations, 189
  no type checking, 114
  cf. Pascal intrinsics, 111
  pitfalls, 112, 189
    version 2.0 improvements, 273
  speed, 188
TrackControl (Toolbox), 142, 145–151, 276
TrackGoAway (Toolbox), 124, 129–133, 276
Types
  enumerated, 237
  SANE, 260–265
    cf. Pascal, 260–261, 265

UnionRgn, 209
User interface
  overview, 2–4
  version 2.0, 273
User Interface Toolbox. *See* Toolbox
uses
  QuickDraw 2, 201–202, 212
  SANE, 260

Variant records, 18–22
Version 2.0, Macintosh Pascal, 5, 269–273

Window(s)
  coordinates, 126
  events, 61
  pointers, 124–125, 137, 138, 173
  text, 155
  titles, 137
  types, 138
Window Manager, InLine access, 122–133
  and controls, 147–149
  creating, 137
  dragging, 131–133
  elements, 122–123
  event handling, 126
  functions listed, 276
  highlighting and closing, 128–129
  programmer-defined, 137–140
  record type, 124
  routines, 123–133
    listed, 124
  sample program, 133–136
  and text, 155
  types, 123
  values, 125
WInLineF (Toolbox access), 112

XOR, logical, 68
XorRgn, 209
Xpwr (SANE), 267
XpwrI (SANE), 266

Advanced Macintosh™ Pascal

Advanced Macintosh Pascal begins where most Pascal books leave off by teaching programmers the advanced skills needed to produce efficient and useful applications. It discusses Macintosh memory management and examines records, sets, and pointers—the three advanced Pascal programming topics that are used extensively by the Macintosh User Interface Toolbox and QuickDraw graphics. Also examined are Pascal file types, sequential and random access files, and programs that utilize the Indexed Sequential Access Methods (ISAM) of handling data.

Learn how to use Macintosh QuickDraw, including advanced structures, routines, and programming techniques. With this programming text you will examine event programming, including the use of the various event types, event records, and bit masks.

You will also explore how to use InLine routines to program the menu manager, window manager, and control manager, as well as program dialog boxes, pulldown menus, radio buttons, and many other of the key features that distinguish the Macintosh interface.

Written in a highly readable style, *Advanced Macintosh Pascal* features numerous easy-to-follow program examples, including a complete application logger that records computer usage. It also presents an overview of Standard Apple Numerical Environment and covers the extended data types and routines needed for scientific and business programming. Programmers of Pascal will find this book an invaluable source for programming the Macintosh.

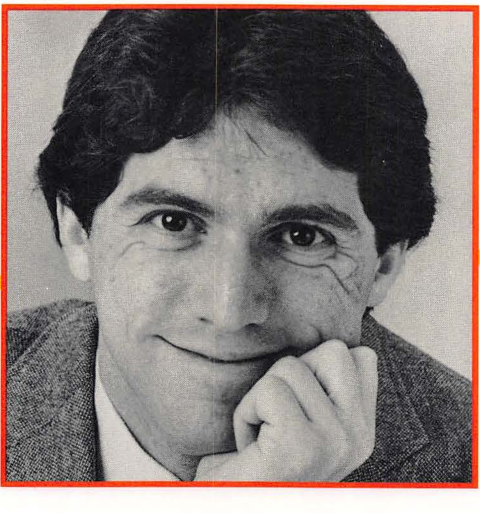


Photo: Ted Hardin

About the Author

Paul Goodman has taught Pascal programming at Queens College in New York, a member of the Apple College Consortium, for over six years. He has also co-authored an introductory Macintosh Pascal book and holds a Master of Science degree in Computer Science.

In addition to teaching, Paul is a consultant for major corporations and educational institutions on the use of personal computers and databases. He programs in ten languages, including Pascal, BASIC, Lisp, Prolog, and Assembly.

HAYDEN BOOKS
A Division of Howard W. Sams & Company
4300 West 62nd Street
Indianapolis, Indiana 46268 USA

$19.95/046570
ISBN: 0-672-46570-1

0 81262 46570 7