MACINTOSH ASSEMBLY LANGUAGE

AN INTRODUCTION

JAN L. HARRINGTON
MACINTOSH ASSEMBLY LANGUAGE:
AN INTRODUCTION
## Table of Contents

PREFACE ix

Chapter 1 INTRODUCTION 1
   Chapter Objectives 1
   Assembly Language: Why Bother? 1
   The Standard Macintosh User Interface 4

Chapter 2 NUMBERING SYSTEMS: MACINTOSH'S MICROPROCESSOR AND MEMORY 15
   Chapter Objectives 15
   Computer Numbering Systems and How Information is Represented in a Computer's Memory 15
   Macintosh's Microprocessor 25
   How Macintosh's RAM is Used 31
   Addressing RAM 33
   Questions and Problems 49

Chapter 3 USING THE MACINTOSH 68000 DEVELOPMENT SYSTEM 51
   Chapter Objectives 51
   Introduction 51
   Using the Editor 55
   The Assembler 58
   The Linker 63
   Running an Application 68
   Run-Time System Errors 68
   The Executive 72
   Debugging 74

Chapter 4 THE 68000 INSTRUCTION SET (PART 1) 80
   Chapter Objectives 80
   Creating an I/O Shell 80
   Assembler Directives 82
   Data Manipulation Instructions 86
   LOOPING 92
   Making Comparisons 94
   Testing the Condition Codes 95
   Questions and Problems 99

Chapter 5 THE 68000 INSTRUCTION SET (PART 2) 103
   Chapter Objectives 103
   Integer Arithmetic 103
Logical Operations 110
Subroutines 114
Putting the Instruction Set to Work —
  Sorting and Searching Arrays 116
Questions and Problems 130

Chapter 6
THE PASCAL CONNECTION TO THE TOOLBOX AND OPERATING SYSTEM ROUTINES 133
Chapter Objectives 133
Pascal Elementary Data Types 134
User-Defined Data Types 137
Pascal Data Structures 138
Procedure and Function Calls 141
An Overview of the Toolbox and Operating System Routines 144
Calling Toolbox and Operating System Routines —
  The Trap Mechanism 149
Using Toolbox and Operating System Routines —
  Modifying the Sort and Search 150
Questions and Problems 156

Chapter 7
SETTING UP THE DESKTOP: WINDOWS AND MENUS 159
Chapter Objectives 159
Creating Windows 159
Programming Technique — Making a Resource File
  Part of Program Code 175
Manipulating Windows 176
Creating Menus 183
Questions and Problems 191

Chapter 8
CONTROLLING PROGRAM ACTIONS:
  MONITORING EVENTS 193
Chapter Objectives 193
The System Event Mechanism 193
Retrieving Events 198
Handling Mouse Down Events 202
Handling Key Down Events 220
Handling Update Events 221
A Word About Activate Events 223
Pulling Things Together Thus Far — WindowPlay 223
Questions and Problems 232

Chapter 9
SCREEN AND KEYBOARD I/O:
  USING TEXTEDIT, ALERT AND DIALOG BOXES 235
Chapter Objectives 235
Entering, Displaying, and Editing Text 235
Controlling Program Actions with Alert and Dialog Boxes  261
Questions and Problems  272

Chapter 10  PRINTING  275
Chapter Objectives  275
Introduction  275
Accessing the Printing Manager  276
Data Structures for Printing  279
Programming Technique — Packing an Equates File  283
Establishing Print Records  284
The Sequence of Printing Manager Routines  285
Opening and Closing the Printing Manager  287
Collecting Information for the Print Record  287
Opening and Closing a Document  288
Printing a Single Page  289
Imaging and Printing Spool Files  300
Completing the Printing Task  301
Putting it All Together — BannerPrint  302
Questions and Problems  319

Chapter 11  FILE I/O  322
Chapter Objectives  322
Introduction  322
Data Structures for File Operations  323
Creating a Data File  330
Opening a File  332
Writing to Disk Files  333
Reading From Disk Files  336
Closing a File  339
Timing out for File I/O  339
Managing Disk Changes and Choosing File Names — the Standard Package  340
Questions and Problems  344

Chapter 12  ARITHMETIC I/O: FLOATING POINT ARITHMETIC  347
Chapter Objectives  347
Introduction  347
The Binary-Decimal Conversion Package  348
Floating Point Decimal-to-Binary Conversions  350
Programming Technique — Using Separately Assembled Subroutines  355
Formats Available Through FP68K's Conversion Routines  357
Executing a Binary-to-Decimal Conversion  359
Programming Technique — Using Macros  360
An Overview of the FP68K and ELEM68K Routines  362
It seems as though there is a flood of Macintosh software hitting the market these days, most of it written in Pascal on a Lisa (the Macintosh's now defunct big brother, most recently known as the Macintosh XL) or in assembly language on either a Lisa or a Macintosh. Some commercial software developers are also beginning to work in Forth and C. No language, however, gives a programmer more control over the Macintosh than assembly language.

People who learn to program the Macintosh in assembly language gain in four ways:

1. They learn the basic principles of microcomputer architecture;
2. They learn how computers store and manipulate numbers;
3. They learn the basics of using an assembly language instruction set;
4. They learn to focus on creating applications with a standard, user-friendly interface.

This book was written to teach the four areas listed above to both students in a classroom situation and an individual working alone. It is designed to take someone who knows Pascal and get that person functioning in an assembly language environment. It assumes no background in computer architecture or assembly language. By the same token, it is not a definitive work. One of its primary goals is to give a programmer the tools needed to understand documentation so that he or she can independently go beyond what this book covers. Therefore, while it presents material common to assembly language programming on all computers, this is also a practical book, aimed at doing assembly language programming on the Macintosh. It was designed to teach assembly language programming specifically on the Macintosh and is not intended to be a general 68000 assembly language text.

Programming the Macintosh in assembly language isn't an easy task. In fact, it can be complex and tedious since access to the Macintosh's internal ROM routines
is really designed for Pascal, and assembly language programmers must simulate the Pascal syntax. Nonetheless, working in assembly language does give a programmer computing power that no other language can deliver.

Learning to program in assembly language on the Macintosh presents two challenges: A person must not only master the microprocessor's instruction set, but also must be able to interact with the ToolBox and operating system routines that reside in Macintosh's ROM. All I/O is done through those routines. In fact, Macintosh assembly language programs are often little more than a series of calls to the ROM routines. The instruction set itself takes a back seat; it is used primarily to set up parameters before issuing a call.

Because assembly language on the Macintosh is a rather complex task, this book is not intended to be an exhaustive treatment of the subject, but it will:

1. Provide the technical background needed to function in assembly language
2. Introduce the commonly used instructions in the Macintosh's instruction set
3. Demonstrate how to use the ToolBox and operating system routines necessary to create basic assembly language applications.

This book does not deal specifically with producing Macintosh graphics. Creating spectacular graphics takes two kinds of knowledge: knowing how to use the ROM graphics routines and knowing how to sequence calls to those routines to draw the desired images. This book teaches the former, how to read the documentation that describes the graphics routines, and provides the skills needed to call the routines from assembly language. Sequencing calls to graphics routines to produce some particular picture, however, is beyond the scope of this book. The effective use of Macintosh graphics is an extensive subject that deserves a book all its own.

Resources for Learning

This book is based on Apple's Macintosh 68000 Development System (MDS), a package of software tools that supports the development of either stand-alone assembly language applications or assembly language routines that can be called by high-level language programs. While it is not the only such package available for the Macintosh, it is the most complete and the most convenient to use. If not available from your regular software supply house, it can be ordered directly from Apple:

Macintosh Technical Support
Apple Computer
MS 4-T
20525 Mariani Avenue
Cupertino, CA 95014

To obtain an exact price and details on ordering, call the customer service line at 408-973-2222 between 9:30 A.M. and 1:30 P.M. Pacific time.
Complete documentation for all Macintosh ROM routines can be found in Inside Macintosh, now available at computer retailers and bookstores or through direct order from Apple. Though the book you are reading right now is independent of Inside Macintosh, programmers inevitably will wish to go beyond what this book presents and it may be difficult to teach a course in Macintosh assembly language without at least one copy of that manual available for reference. This book teaches people how to interpret what they find in Inside Macintosh, how to decipher the Pascal syntax for the ToolBox and operating system calls and turn it into assembly language. It also focuses on understanding the sequence in which they should use the ROM routines.

There is one other reference that students should use in conjunction with this book and Inside Macintosh—The MC68000 Programmer’s Reference Manual (hereafter referred to as the PRM). The PRM, which is included with the Macintosh 68000 Development System, is a reference work detailing the instruction set of the Macintosh’s 68000 microprocessor.

Developing assembly language programs is much easier with the aid of a number of utility programs that Apple has written. These include programs that dump the contents of a disk file in hexadecimal, print a spooled print file, and aid in creating screen formats and alert and dialog boxes. For a while Apple was distributing these utilities with the Software Supplement to Inside Macintosh. Now that Inside Macintosh is available in bookstores, however, the utilities can be downloaded from a number of dial-up information systems, such as CompuServe, and from public bulletin boards. They are also available from most Macintosh users groups.

**Reader Background**

This book assumes that the reader has knowledge equivalent to a one-semester course in Pascal, though not necessarily on the Macintosh. It also assumes that the reader has some experience working with the Macintosh itself. In particular, he or she should have used a Macintosh word processor such as MacWrite. Though Chapter 1 discusses in detail the characteristics of the Macintosh user interface, the book assumes that people are familiar with mouse-driven applications that use pull-down menus and overlapping windows.

**Overview of the Book**

The introduction found in Chapter 1 lays a foundation for the Macintosh assembly language environment. It discusses the differences between assembly language and high-level languages and explains what is to be gained by working in assembly language. The chapter also examines the characteristics of the standard Macintosh user interface, emphasizing that all successful Macintosh software adheres to that interface.
Chapter 2 presents technical background information. This includes a look at the binary, octal and hexadecimal numbering systems, the architecture of the Macintosh's microprocessor (in particular, its registers), how the Macintosh uses its available RAM (including the stack), and addressing memory from assembly language.

Chapter 3 contains a short assembly language program to type in. This will provide practice in using the Macintosh 68000 Development System. Working through the exercise early in the course will make it easier for students to concentrate on programming without worrying about how to use the Editor, Assembler, and Linker.

Chapters 4 and 5 present an introduction to the assembly language instructions that form the backbone of a Macintosh assembly language program. This chapter has numerous blocks of sample code, each of which is to be inserted into a ToolBox "shell" that is created out of the program in Chapter 3.

Although this book deals with assembly language, it's a fact of life that access to ToolBox and operating system routines is based in Pascal. If people are going to be able to read the documentation of those routines in *Inside Macintosh*, they must understand Pascal data types, data structures, and procedures and functions. Therefore, Chapter 6 reviews the necessary Pascal concepts. It also describes the structure of the ToolBox and operating system routines and how an assembly language program gains access to them.

The remainder of the book deals with the ToolBox and operating system routines that are needed to create a Macintosh assembly language application. Chapter 7 discusses setting up the desktop (managing windows and pull-down menus). Chapter 8 discusses managing program operation by monitoring the keyboard and mouse. Chapter 9 handles entering and editing text. Printing from an application (tedious but not difficult) is presented in Chapter 10. File I/O (not as complicated as it looks) is discussed in Chapter 11.

Chapter 12 discusses floating point arithmetic. Even if an application does no significant amount of math, it will at least need to use the routines that convert a string of numeric characters into a binary number and a binary number to a string of characters for numeric I/O.

The Video Tape Index Program

The major application that is developed throughout most of the book is a video tape index. The program is a specialized database system that could be used in a home to catalog which program has been recorded on which video tape or in a video rental outlet for inventory control. The video tape index program first appears in Chapter 5 in the discussion of handling arrays in main memory and is used to explore sorting and searching techniques for such arrays. It is presented in bits and pieces throughout the book. Complete source code for the program can be found in Appendix A.
To most students the source code for the video tape index program may look a bit forbidding at first. It is long — about 3,000 lines of code — far longer than most of the Pascal programs that are written in programming classes. Nonetheless, it assembles to only about 12K and uses another 12K of space for data storage. It will therefore run on a 128K Mac.

Why use such a large example? Certainly the sample programs that come with the Macintosh 68000 Development System are much shorter. First, the very simplicity of those examples creates a problem. The features of a Macintosh application interact in many unexpected ways. While Apple provides a sample program that creates a window, the video tape index uses multiple, overlapping windows to demonstrate more extensive window management. One of Apple's sample programs demonstrates text editing, but only in one window. The video tape index uses multiple windows for text editing to explore a more complex, meaningful environment.

Secondly, meaningful Macintosh assembly language programs do become very large, generally occupying 25 to 400K. Apple's short examples really don't do any meaningful work. The video tape index program is a complete, useful application that can easily be customized to meet individual needs. It is also available, along with other sample programs, on disk from the publisher of this book.
INTRODUCTION

Chapter Objectives

1. To explore the advantages and disadvantages of programming in assembly language
2. To become familiar with the characteristics of the standard Macintosh user interface

Assembly Language: Why Bother?

Back in the early days (that means anything before 1964), people who wanted to learn about computers studied logical circuit design and then a mysterious language called "assembly." Once they had mastered assembly language programming, they moved on to high-level languages like FORTRAN.

It doesn't work that way any more. For most people today, their first exposure to programming is through a high-level (English-like) language, usually BASIC or Pascal. Pascal doesn't require much knowledge about the internal workings of your computer; not necessarily a bad thing, since knowing the internal organization of a microprocessor doesn't automatically make you a good programmer.

There are, though, some things that assembly language can do better than BASIC, Pascal and other high-level languages. Primarily, assembly language
programs run faster than high-level language programs. To understand why, you must first realize that there is only one language that a computer can run directly—machine language. Machine language consists of a series of binary codes (0's and 1's) which make perfect sense to a computer but very little sense to a human.

Assembly language was created to free programmers from having to program in machine language. Each command that the computer could understand (an instruction) was given a short mnemonic code consisting of two to five letters. Programmers could then use the mnemonics rather than the complex binary. Once the source code was finished, it had to be translated into machine language so the computer could run it. The translation was (and still is) accomplished by a program called an assembler. The resulting machine language version is called object code and can be run directly by the computer.

High-level languages also require translation to object code. Most versions of BASIC are interpreted. That means that the conversion to object code occurs line by line as the program is being run; no permanent machine language version of the program is ever created. If you have a FOR/NEXT loop that repeats 100 times, every statement in that loop will be translated to machine language 100 times. Interpreted BASIC programs are just about the slowest programs around.

Most other high-level languages are compiled. All of the translation to machine language occurs at one time. Just like an assembler, a compiler gives you a machine language version of your program. Object code derived from a compiler usually cannot be run alone, though. It needs to be linked to run-time libraries (a collection of standard programs that handle functions such as input and output). While compiled programs can run almost as fast as assembled programs, they tend to be bigger. This becomes a major concern when you are writing software for a machine with limited RAM such as the first edition Macintosh (with only 128K).

In addition to increasing the speed of program execution, assembly language gives you more control over your computer than high-level language. When you use an interpreted language, you have little opportunity to determine where your program or its variable tables are placed in main memory. Though some compilers do allow you to specify where large blocks of code should begin (e.g., your program's object code and run-time libraries), you are still extremely limited. With assembly language, you can access RAM locations directly and determine exactly what will be placed in each location. A well-written assembly language program is generally more efficient than an interpreted or compiled program; in other words, it makes better use of available main memory.

In order to gain the speed and efficiency of assembly language programs, you must in turn know something about the internal physical organization of your computer. You need to know not only how RAM is used, but you must also have some knowledge of the "architecture" of its microprocessor.

Assembly language has one major drawback, assuming that you don't consider having to acquire technical knowledge about your computer a drawback.
High-level languages are more or less portable between different computers. Consider, for example, all the different microprocessors and operating systems that run Microsoft BASIC. Languages such as Pascal and FORTRAN differ only minimally between computers. Assembly language, however, is specific to one particular microprocessor; the mnemonics are different for each one. Therefore, learning assembly language on one computer does not automatically prepare you to write assembly language programs on another. Each microprocessor has its own instruction set (the entire group of instructions that a microprocessor can understand).

Nonetheless without programming in assembly language it is very difficult to do serious program development on a Macintosh. With BASIC you are limited to very small, very slow programs. For example, after the Microsoft interpreter is loaded, you have only 14K left in the 128K machine for programs. Though this limitation has no relevance for the 512K Macintosh, a significant number of 128K machines have been purchased and much software is designed to run in that more restrictive environment.

Many Macintosh programs have been written in Pascal, but they were developed on a Lisa. Lisa Pascal for the Macintosh is very different from MacPascal. MacPascal is interpreted, like BASIC. That means that while it is an excellent tool for learning about Pascal, programs written in MacPascal will run nearly as slowly as interpreted BASIC programs.

There is another disadvantage to developing Macintosh applications completely in a high-level language which relates to the nature of Macintosh software. Successful Macintosh applications are designed around the standard Macintosh user interface (discussed in the second part of this chapter). To implement that interface, the Macintosh uses a set of over 500 prewritten routines. Some are in ROM (read only memory); others are on disk as part of the system files. The routines fall into two major groups: those that are part of the operating system and those that constitute the Toolbox. (For an overview of Macintosh's built-in routines, see Chapter 6.)

No language currently available gives you access to all of the Toolbox and operating system routines within the standard language environment. (Lisa Pascal can call all of the Macintosh's internal routines, but MacPascal cannot.) Some, like Microsoft BASIC 2.0, allow a programmer to build assembly language libraries that can be called from the high-level language program. Others, like MacPascal, have an interface to many of the routines which require assembly language knowledge to set up the calls. A programmer who wishes to exploit a Macintosh high-level language to its maximum must therefore have at least some knowledge of the Macintosh assembly language interface.

What it all boils down to is this — if you want to be able to tap the full power of a Macintosh, then you will find that being able to use assembly language is the most valuable tool available.
The Standard Macintosh User Interface

The Macintosh has made most of us redefine what it means when we say a program is easy to use. When we open a brand-new piece of Macintosh software, we expect to be able to run it by simply double-clicking on its icon from the Finder. We also expect to find that program actions are controlled by menus and that the mouse controls placement of the cursor. These are all characters of the standard Macintosh user interface. They have the effect of making Macintosh applications programs very easy to learn and use. By the same token, they increase the burden on the programmer.

Macintosh software packages that stray from the standard user interface have not fared well with reviewers or users. During the first six to nine months after Macintosh was released, many independent software developers rushed to market Macintosh versions of software that was running on other systems without completely adapting it to the Macintosh environment. Few of those early efforts are still being sold; most have been significantly upgraded to adhere to the Macintosh interface. The moral of the story is...if you're going to program the Macintosh, do it Apple's way when it comes to the user interface. In terms of that interface, creativity wins few prizes.

The Macintosh standard user interface is characterized by the following:

1. Use of the mouse as the primary input device to control menu selections, window manipulation, cursor placement and text selection
2. Pull-down menus, including the standard Apple, File, and Edit menus
3. Multiple, overlapping windows
4. Text editing with cut, copy, paste and clear functions
5. Control of program actions with alert and dialog boxes

Macintosh Cursors

The Macintosh’s mouse is “hard wired” to a moveable cursor that appears on the screen; it lays on top of everything else that is displayed. The cursor’s shape will vary with particular program actions. It may be:

1. An arrow (used when making menu selections, dragging windows, closing windows, sizing windows, scrolling windows contents, etc.)
2. A straight line (used to mark the place where text characters will be inserted)
3. An I-beam (used to aid in positioning the cursor in text documents)
4. A wrist watch (used to indicate long waits)

Other special cursors include the cross for sizing and positioning graphics objects, and an outlined cross used for making array selections. Applications may also design their own cursors.

**Menus**

Menus were certainly not invented by the Macintosh development team; they are used in a great deal of commercial software. Most users consider menu-driven software as easier to learn and easier to use than software that requires learning a set of commands. Menus on other computers, however, not only look different from Macintosh menus but accept input about menu selections in a very different way.

A typical non-Macintosh menu appears in Figure 1.1. A program using this menu will usually clear the screen, print the menu, and issue an input statement (e.g., a Pascal `read`). The user makes a selection by entering a number that corresponds to the appropriate menu option. Program execution is suspended until the menu selection is made; the user has no way to escape from making a menu choice, save perhaps resetting the computer.

![File Menu -
1. New
2. Open
3. Close
4. Save
5. Save As
6. Print
7. Quit

Enter option number:](image)

**Figure 1.1 A Standard Microcomputer Menu**

Macintosh menus also present the user with a list of options. Figure 1.2 presents the Macintosh version of the menu from Figure 1.1. The menu's title appears above the part of the screen where program actions take place; this is known as the *menu bar*. To see the menu options, the user positions the arrow cursor on the menu title, presses the mouse button, and drags the arrow down.
Options are *highlighted* (displayed in inverse video—white letters on a black background) as the arrow cursor is dragged. The user indicates a menu selection by releasing the mouse button when the cursor is positioned on the appropriate option.

Two things make the Macintosh menu selection process very different from standard menu selections. In the first place, the user can escape from the menu by either returning the arrow cursor to the menu title or by dragging the cursor off the bottom of the menu. Secondly, pulling down the menu doesn't require erasing what appears on the main portion of the screen, though part of the screen may be temporarily covered by the menu options. Selections from Macintosh menus can therefore be made while text and/or graphics are present on the Macintosh screen.

Most Macintosh applications will support three standard menus plus any additional menus the application requires. The leftmost menu in the menu bar has the silhouette of an apple with a bite out of it for a title. This "Apple" menu (see Figure 1.3) supports the Macintosh desk accessories and may also include an "about" feature that describes the software in which the menu appears. A desk accessory is a stand-alone program that can be run at any time without exiting the major application (e.g., MacWrite or MacPaint) being executed.
The second menu from the left is the File menu (see Figure 1.2). A standard file menu provides options for opening new and existing files, saving files, closing files, printing files, and exiting the program. The third standard menu, the Edit menu (Figure 1.4), implements editing operations: cut, paste, copy, and clear (delete). Note that clear is often not supported as a menu item (that is the case in Figure 1.4).
Users who are very familiar with a piece of software often prefer to issue menu selections from the keyboard. To make this possible, an application can associate a pair of keystrokes with any or all options in a menu. Known as *keyboard equivalents*, they appear to the right of the menu options as the cloverleaf symbol followed by a single key. Keyboard equivalents for the File and Edit menus are standard and should not be changed. Note that identifying which option has been selected from which menu, regardless of whether the selection is made by mouse or keyboard equivalent, is not automatic; it must be programmed into an application.

An application has complete control over which menus appear in the menu bar. The three standard menus should usually be present. Nonetheless, there are times when it makes no sense in terms of program function to allow selection from a particular menu. In that case, an application should disable that menu. Titles of windows that have been disabled appear dimmed; their titles are printed in light grey rather than black (Figure 1.5). If it makes sense to disable only specific options rather than an entire menu, the application should do so. Options that have been disabled appear dimmed, while the menu title is still printed in black (Figure 1.6).

Details on creating menus, manipulating the menu bar, and disabling and enabling menus can be found in Chapter 7. Information of identifying menu selections is part of Chapter 8.

![Figure 1.5 A Disabled Menu](image)
The New, Old, and Quit options in this File menu are
dimmed. That means that those particular options are
not available to the user. On the other hand, the Save As
option is available. Only the options that are not
appropriate at the time are dimmed.

Figure 1.6 A Macintosh Menu with Disabled Items

Windows

Windows are rectangles that appear on the Macintosh screen. They are used
to display text and graphics, to collect data essential to program function, and to
warn the user about the consequences of specific actions.

The Macintosh supports six different types of windows. Depending on its type,
a window may have one or more of the following features (see Figure 1.7):

1. A title displayed in a title bar
2. A drag region (the entire title bar except for the GoAway box)
3. A GoAway box (at the left of the title bar)
4. Controls (e.g., scroll bars, push buttons, radio button, check buttons)
5. A grow icon (located in the lower right corner of the window) - note that an
   icon is nothing more than a small picture that represents an object or a
   function within the computer.

A window that accepts user input, regardless of whether that input is text or
graphics, has the same title as the document file which contains the material on
disk. If the document has not yet been saved, the window title is "Untitled." Other
windows, such as the desk accessories, have titles that reflect their function. For
example, the note pad desk accessory's window has the title "Note Pad." Windows
that warn users (alerts) and windows that collect data (dialogs) have no titles.
The drag region consists of the entire title bar except the GoAway box. It allows the user to move the window around the screen. When a user positions the cursor in the drag region and presses the mouse button, an outline of the window will follow the arrow cursor as the user drags it around the screen. The final position of the window is determined by the location of the arrow cursor when the mouse button is released.

A GoAway box is the small rectangle that appears in the left-hand corner of the title bar. If the mouse button is clicked while the arrow cursor is within the GoAway box, the application should close the window. If the window contains a document that has been modified since it was last saved to disk, the application will ask the user whether or not the document should be saved before closing.

The term "controls" refers to a group of things that can appear in a window. They include scroll bars, push buttons, radio buttons, and check boxes. (The latter three are illustrated in Figure 1.8.) Scroll bars are used to change the portion of a large document that is visible at any time within a window. Scrolling is discussed in Chapters 7 and 8. The other types of controls appear primarily in dialog and alert boxes (see Chapter 9).

A grow icon appears in the lower right-hand corner of document windows. It allows a user to change the size of a window. When the user positions the arrow cursor in the grow icon and presses the mouse button, an outline of the window will follow the cursor as it is dragged about the screen. The final size of the window is determined by the position of the cursor when the mouse button is released. Sizing windows is discussed in Chapter 8.

Windows are not restricted to changing their size and position within a single plane. They can change positions relative to any other windows present on the screen. If we assume that windows are stacked on the screen like sheets of paper
might be piled on a desk, then we can say that windows can change their location in that pile. In Macintosh terminology, windows move from front to back. Like pieces of paper, Macintosh windows can overlap. Windows to the back may be obscured by those in front of them.

![Macintosh Controls](image)

**Figure 1.8** Macintosh Controls

The front-most window on the screen is the active window; an application can only work in an active window. Active windows are highlighted in some way, though the actual details of the highlighting depend on the type of window. For example, the highlighting in standard document windows like the one in Figure 1.7 includes horizontal lines in the title bar and a pattern in the scroll bars. When a standard document window is inactive, its title bar will contain only the title.

**Text Editing**

Throughout a Macintosh application, entry and modification of text is managed in a single, consistent manner. The place where new characters are added (indicated by a single, straight-line cursor) is known as the **insertion point**.

**Cut, paste, copy**, and **clear** — the editing operations — affect one or more contiguous characters in a block known as the **selection range**. The selection range is highlighted (see Figure 1.9) by displaying white characters on a black background.

A user selects text in two major ways:

1. By holding down the mouse button and dragging the cursor across the text. (In this case, if a selection goes beyond what is currently visible in the text
window, the text should scroll.) All text over which the cursor is dragged will be included in the selection range.

2. By clicking the mouse button while the shift button is down (known as shift-clicking). All text between the current position of the cursor and the place where the shift-click occurred will be selected.

![Figure 1.9 The Selection Range in a Text Document](image)

This is an example of how the Macintosh displays the current selection range. The characters in the selection range are displayed in inverse video.

The editing operations (cut, paste, and copy) affect what is known as the clipboard. The clipboard is a holding area for text and/or graphics images. It may be kept in main memory or may be saved to disk if it becomes very large. Executing a cut deletes the current selection range and places it on the clipboard; copy merely places the selection range on the clipboard without deleting it from the document. Paste takes whatever is on the clipboard and places it in the document just after the current selection range. Generally, the selection range for paste operations will simply be an insertion point. Note that the clipboard can only hold one thing at a time. While paste does not disturb the contents of the clipboard, each cut or copy replaces what was previously there.

Clear does not affect the clipboard. It merely deletes the current selection range. The backspace key has the same effect as clear.

The implementation of Macintosh text editing is discussed in detail in Chapter 9.
Alerts and Dialog Boxes

Alerts and dialog boxes are specialized windows. They are used by an application to either warn the user about the consequences of a particular action (an alert) or to collect information essential to program function (dialog boxes).

Alerts contain the text of a warning and one or more push buttons (see Figure 1.10). One button is selected as the default button; it is heavily outlined. Pressing either the Enter or Return key will have the same effect as positioning the arrow cursor over the button and clicking the mouse button. Most alert boxes have an OK button which simply closes the alert and continues with program action. Some also have a Cancel button which permits a user to escape from some action he or she may have inadvertently requested.

![Figure 1.10 An Alert](attachment:image)

Characteristics of an alert:
1. Contains text and push buttons
2. The default button is heavily outlined. It will be selected when the user presses Enter or Return
3. The alert is the active window (note that the text window in the background has been unhighlighted)

Dialog boxes come in two varieties: modal and modeless. A modal dialog box prevents the user from working anywhere but within the box. They are used to collect information that the application must have before it can continue. For example, a modal dialog box is used to collect that name of a file before saving it to disk for the first time. Modal dialog boxes display messages, have areas for entering text, and can contain push buttons, radio buttons, and check boxes (see Figure 1.11). They are removed completely from the screen when the user has finished with them.
Modeless dialog boxes are much more like other windows. They permit the user to work outside the dialog box while the dialog box is still on the screen. Modeless dialog boxes are most commonly used to implement Find and Search operations (see Figure 1.12).

Alerts and dialog boxes are discussed in Chapter 9.

**Figure 1.11 A Modal Dialog Box**

**Figure 1.12 A Modeless Dialog Box**
Chapter Objectives

1. To learn the three major numbering systems used to represent instructions, characters, and quantities in a computer

2. To understand the organization of the Macintosh's microprocessor and, in particular, its registers

3. To understand the purpose of a stack and how it works

4. To understand how the Macintosh's main memory is distributed between the operating system and an application program

5. To get an overview of the ways in which a Macintosh assembly language program specifies the location of data in main memory (addressing modes)

6. To understand the use of symbolic addresses

---

Computer Numbering Systems
and How Information is Represented in a Computer's Memory

When we talk about a computer's memory, we use either the hexadecimal (base 16) or octal (base 8) numbering systems. Both are used as a shorthand for
Binary Numbers

Base 2 (binary) is a natural for describing the internal state of a computer. Anything we want to put in a computer must be represented by groups of integrated circuits. Each one of those circuits can carry either a high voltage (by convention, assigned a value of 1) or a low voltage (assigned a value of 0). As it so happens, 0 and 1 are the digits that make up the binary numbering system.

As you probably remember from junior-high math, binary numbers work on a place-value system, just like the base 10 numbers we use every day. Instead of representing a power of 10, though, each binary place represents a power of 2.

Figure 2.1 shows you a sample binary number and the base 10 value of each place. There is one group of 128, one group of 64, one group of 32, one group of 8, and a single 1. In base 10, this number would be 233. To convert a binary number to a base 10 number, all you have to do is add up the base 10 place values of each binary place that has a 1 in it.

```
1 1 1 0 1 0 0 1  a binary number
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 Base Two place values
128 64 32 16 8 4 2 1 Base Ten equivalents
```

To covert Base Two (binary) to Base Ten (decimal):

Add up the decimal place values of each binary place that contains a one:

```
128 + 64 + 32 + 8 + 1 = 233
```

Figure 2.1 A Binary Number

Each binary place is called a binary digit, or bit. A bit can stand for one of two different things and therefore takes a value of either 0 or 1.

We certainly need to be able to have more than two values in the computer (there are 53 letters, 10 digits, and a number of punctuation marks and special characters), so we group a series of bits together. Eight bits are called a byte. A
byte can represent the binary equivalent of 0 through 255 (you get 255 when you put a 1 in each of the eight places). Those values are used as codes for whatever we want to store in the computer. The bits in a byte are numbered from 0 through 7, starting at the right.

Though there are a number of coding schemes, most microcomputers (including the Macintosh) use ASCII code to represent characters and instructions. (Numbers intended for mathematical operations are usually not coded, but stored as binary quantities.) ASCII stands for American Standard Code for Information Interchange.

When you studied Pascal, one important thing you had to know was the difference between storing a digit as a CHAR or as an INTEGER or a REAL. If you stored the digit in a CHAR variable, then you couldn't do arithmetic operations with it unless you first converted it to an INTEGER or a REAL. You are now in a position to understand why.

The binary ASCII codes for the digits 0 through 9 are 0110000 (48 in base 10) to 0111001 (57 in base 10). That's what will be stored in main memory when you assign a digit to a CHAR variable. The value of these codes bears no relation to the actual quantity the digits represent, and trying to use them in arithmetic operations would certainly produce ridiculous results. On the other hand, storing one of the quantities 0 through 9 in a numeric variable stores 0000 to 1001, the exact binary equivalent of the digit.

Storing digits as a CHAR requires one byte per digit. For example, "28" would be stored as 0011010 and 0111000. Numbers, though, can hold up to 255 in a single byte. 28 would be 00011100.

You've probably noticed that the ASCII codes for the digits are only 7 bits long. Standard ASCII is a 7-bit code. The eighth bit in the byte is usually not used.

The Macintosh, though, uses an extended ASCII code which lets you use a combination of the shift and option keys to generate characters which are not usually available from the keyboard. These additional characters are created by using bit seven (the eighth bit) to provide additional code combinations. Standard ASCII codes end at 01111111, but Macintosh codes go all the way through 10001001. You can see Macintosh's character codes in Table 2.1

Most assemblers, including the MDS Assembler, will accept binary numbers as part of the source code. To indicate that a quantity is binary, preface it with a percent sign (%). For example, the Assembler will recognize %1100011 as a binary number having the decimal value 99. Without the percent sign, the number will be interpreted as base 10 with the value of one hundred ten thousand and eleven.

The binary system is used in computers for one other major purpose besides specifying ASCII codes; it is used to count the bytes in the computer's memory. The number given to each byte is called its address. In the 128K Macintosh, there are 131,072 bytes of RAM (one kilobyte = 1024 bytes), so the binary equivalent of the maximum address is 1111111111111111. Such a number is too long for most people to handle easily. Therefore, we use hexadecimal as a shorthand.
### Table 2.1 Macintosh Extended ASCII Character Set

<table>
<thead>
<tr>
<th>What you Press</th>
<th>Binary Code</th>
<th>Hex Code</th>
<th>Meaning of the code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>00000000</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>Start of header</td>
<td>00000001</td>
<td>01</td>
<td>Start of header</td>
</tr>
<tr>
<td>Start of text</td>
<td>00000010</td>
<td>02</td>
<td>Start of text</td>
</tr>
<tr>
<td>Enter</td>
<td>00000011</td>
<td>03</td>
<td>Enter</td>
</tr>
<tr>
<td>End of tape</td>
<td>00000100</td>
<td>04</td>
<td>End of tape</td>
</tr>
<tr>
<td>Enquiry</td>
<td>00000101</td>
<td>05</td>
<td>Enquiry</td>
</tr>
<tr>
<td>Acknowledge</td>
<td>00000110</td>
<td>06</td>
<td>Acknowledge</td>
</tr>
<tr>
<td>Bell</td>
<td>00000111</td>
<td>07</td>
<td>Bell</td>
</tr>
<tr>
<td>Backspace</td>
<td>00001000</td>
<td>08</td>
<td>Backspace</td>
</tr>
<tr>
<td>Horizontal tab</td>
<td>00001001</td>
<td>09</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>Line feed</td>
<td>00001010</td>
<td>0A</td>
<td>Line feed</td>
</tr>
<tr>
<td>Vertical tab</td>
<td>00001011</td>
<td>0B</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>Form feed</td>
<td>00001100</td>
<td>0C</td>
<td>Form feed</td>
</tr>
<tr>
<td>Carriage return</td>
<td>00001101</td>
<td>0D</td>
<td>Carriage return</td>
</tr>
<tr>
<td>Shift out</td>
<td>00001110</td>
<td>0E</td>
<td>Shift out</td>
</tr>
<tr>
<td>Shift in</td>
<td>00001111</td>
<td>0F</td>
<td>Shift in</td>
</tr>
<tr>
<td>Data link escape</td>
<td>00010000</td>
<td>10</td>
<td>Data link escape</td>
</tr>
<tr>
<td>Open Apple</td>
<td>00010001</td>
<td>11</td>
<td>Open Apple</td>
</tr>
<tr>
<td>Check mark</td>
<td>00010010</td>
<td>12</td>
<td>Check mark</td>
</tr>
<tr>
<td>Filled diamond</td>
<td>00010011</td>
<td>13</td>
<td>Filled diamond</td>
</tr>
<tr>
<td>Filled circle</td>
<td>00010100</td>
<td>14</td>
<td>Filled circle</td>
</tr>
<tr>
<td>Closed Apple</td>
<td>00010101</td>
<td>15</td>
<td>Closed Apple</td>
</tr>
<tr>
<td>Synchronous idle</td>
<td>00010110</td>
<td>16</td>
<td>Synchronous idle</td>
</tr>
<tr>
<td>End transmission block</td>
<td>00010111</td>
<td>17</td>
<td>End transmission block</td>
</tr>
<tr>
<td>Cancel</td>
<td>00011000</td>
<td>18</td>
<td>Cancel</td>
</tr>
<tr>
<td>End of medium</td>
<td>00011001</td>
<td>19</td>
<td>End of medium</td>
</tr>
<tr>
<td>Substitute</td>
<td>00011010</td>
<td>1A</td>
<td>Substitute</td>
</tr>
<tr>
<td>Clear</td>
<td>00011101</td>
<td>1B</td>
<td>Clear</td>
</tr>
<tr>
<td>Move left</td>
<td>00011110</td>
<td>1C</td>
<td>Move left</td>
</tr>
<tr>
<td>Move right</td>
<td>00011111</td>
<td>1D</td>
<td>Move right</td>
</tr>
<tr>
<td>Move up</td>
<td>00011110</td>
<td>1E</td>
<td>Move up</td>
</tr>
<tr>
<td>Move down</td>
<td>00011111</td>
<td>1F</td>
<td>Move down</td>
</tr>
</tbody>
</table>

*Non-printing characters generally cannot be generated from the keyboard (exceptions are noted in the "What you Press" column).
†These keys appear on the Macintosh keypad.
### PRINTING CHARACTERS

<table>
<thead>
<tr>
<th>Press</th>
<th>See</th>
<th>Binary Code</th>
<th>Hex Code</th>
<th>Press</th>
<th>See</th>
<th>Binary Code</th>
<th>Hex Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space bar</td>
<td>A space</td>
<td>00100000</td>
<td>20</td>
<td>SHIFT-2</td>
<td>@</td>
<td>01000000</td>
<td>40</td>
</tr>
<tr>
<td>SHIFT-1</td>
<td>!</td>
<td>00100001</td>
<td>21</td>
<td>SHIFT-a</td>
<td>A</td>
<td>01000001</td>
<td>41</td>
</tr>
<tr>
<td>SHIFT-1</td>
<td>&quot;</td>
<td>00100010</td>
<td>22</td>
<td>SHIFT-b</td>
<td>B</td>
<td>01000010</td>
<td>42</td>
</tr>
<tr>
<td>SHIFT-3</td>
<td>#</td>
<td>00100011</td>
<td>23</td>
<td>SHIFT-c</td>
<td>C</td>
<td>01000011</td>
<td>43</td>
</tr>
<tr>
<td>SHIFT-4</td>
<td>$</td>
<td>00100100</td>
<td>24</td>
<td>SHIFT-d</td>
<td>D</td>
<td>01000100</td>
<td>44</td>
</tr>
<tr>
<td>SHIFT-5</td>
<td>%</td>
<td>00100101</td>
<td>25</td>
<td>SHIFT-e</td>
<td>E</td>
<td>01000101</td>
<td>45</td>
</tr>
<tr>
<td>SHIFT-7</td>
<td>&amp;</td>
<td>00100110</td>
<td>26</td>
<td>SHIFT-f</td>
<td>F</td>
<td>01000110</td>
<td>46</td>
</tr>
<tr>
<td>SHIFT-9</td>
<td>(</td>
<td>00101000</td>
<td>28</td>
<td>SHIFT-h</td>
<td>H</td>
<td>01001000</td>
<td>48</td>
</tr>
<tr>
<td>SHIFT-0</td>
<td>)</td>
<td>00101001</td>
<td>29</td>
<td>SHIFT-i</td>
<td>I</td>
<td>01001001</td>
<td>49</td>
</tr>
<tr>
<td>SHIFT-8</td>
<td>*</td>
<td>00101010</td>
<td>2A</td>
<td>SHIFT-j</td>
<td>J</td>
<td>01001010</td>
<td>4A</td>
</tr>
<tr>
<td>SHIFT-=</td>
<td>+</td>
<td>00101011</td>
<td>2B</td>
<td>SHIFT-k</td>
<td>K</td>
<td>01001011</td>
<td>4B</td>
</tr>
<tr>
<td>,</td>
<td>,</td>
<td>00101100</td>
<td>2C</td>
<td>SHIFT-l</td>
<td>L</td>
<td>01001100</td>
<td>4C</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>00101101</td>
<td>2D</td>
<td>SHIFT-m</td>
<td>M</td>
<td>01001101</td>
<td>4D</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>00101110</td>
<td>2E</td>
<td>SHIFT-n</td>
<td>N</td>
<td>01001110</td>
<td>4E</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>00101111</td>
<td>2F</td>
<td>SHIFT-o</td>
<td>O</td>
<td>01001111</td>
<td>4F</td>
</tr>
</tbody>
</table>

0·0 0 00110000 30  SHIFT-p  P 01010000 50
1 1 00110001 31  SHIFT-q  Q 01010001 51
2 2 00110010 32  SHIFT-r  R 01010010 52
3 3 00110011 33  SHIFT-s  S 01010011 53
4 4 00110100 34  SHIFT-t  T 01010100 54
5 5 00110101 35  SHIFT-u  U 01010101 55
6 6 00110110 36  SHIFT-v  V 01010110 56
7 7 00110111 37  SHIFT-w  W 01010111 57
8 8 00111000 38  SHIFT-x  X 01011000 58
9 9 00111001 39  SHIFT-y  Y 01011001 59
SHIFT-;  : 00111010 3A  SHIFT-z  Z 01011010 5A
;  ; 00111011 3B  [  ] 01011011 5B
SHIFT-<  < 00111100 3C  \  \  01011100 5C
= = 00111101 3D  ]  ] 01011101 5D
SHIFT->  > 00111110 3E  SHIFT-^  ^  01011110 5E
SHIFT-/  ? 00111111 3F  SHIFT-  _  01011111 5F
Table 2.1 (continued)

PRINTING CHARACTERS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>'</td>
<td>01100000</td>
<td>60</td>
<td>OPT-u/SHFT-a</td>
<td>Å†</td>
<td>10000000</td>
<td>80</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>01100001</td>
<td>61</td>
<td>SHFT-OPT-a</td>
<td>Å</td>
<td>10000001</td>
<td>81</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>01100010</td>
<td>62</td>
<td>SHFT-OPT-c</td>
<td>Č</td>
<td>10000010</td>
<td>82</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>01100011</td>
<td>63</td>
<td>OPT-e/SHFT-e</td>
<td>Ė</td>
<td>10000011</td>
<td>83</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>01100100</td>
<td>64</td>
<td>OPT-n/SHFT-n</td>
<td>Ň</td>
<td>10001000</td>
<td>84</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>01100101</td>
<td>65</td>
<td>OPT-u/SHFT-o</td>
<td>Ō</td>
<td>10001010</td>
<td>85</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>01100110</td>
<td>66</td>
<td>OPT-u/u</td>
<td>Ū</td>
<td>10001100</td>
<td>86</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>01100111</td>
<td>67</td>
<td>OPT-e/a</td>
<td>á</td>
<td>10001110</td>
<td>87</td>
</tr>
<tr>
<td>h</td>
<td>h</td>
<td>01101000</td>
<td>68</td>
<td>OPT-'/a</td>
<td>á</td>
<td>10001100</td>
<td>88</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
<td>01101001</td>
<td>69</td>
<td>OPT-'/a</td>
<td>á</td>
<td>10001101</td>
<td>89</td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>01101010</td>
<td>6A</td>
<td>OPT-u/a</td>
<td>å</td>
<td>10001100</td>
<td>8A</td>
</tr>
<tr>
<td>k</td>
<td>k</td>
<td>01101011</td>
<td>6B</td>
<td>OPT-n/a</td>
<td>å</td>
<td>10001111</td>
<td>8B</td>
</tr>
<tr>
<td>l</td>
<td>l</td>
<td>01101100</td>
<td>6C</td>
<td>OPT-a</td>
<td>å</td>
<td>10001100</td>
<td>8C</td>
</tr>
<tr>
<td>m</td>
<td>m</td>
<td>01101101</td>
<td>6D</td>
<td>OPT-c</td>
<td>ç</td>
<td>10001101</td>
<td>8D</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>01101110</td>
<td>6E</td>
<td>OPT-e/e</td>
<td>é</td>
<td>10001110</td>
<td>8E</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
<td>01101111</td>
<td>6F</td>
<td>OPT-'/e</td>
<td>é</td>
<td>10001111</td>
<td>8F</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>01110000</td>
<td>70</td>
<td>OPT-'/e</td>
<td>é</td>
<td>10010000</td>
<td>90</td>
</tr>
<tr>
<td>q</td>
<td>q</td>
<td>01110001</td>
<td>71</td>
<td>OPT-'/e</td>
<td>é</td>
<td>10010001</td>
<td>91</td>
</tr>
<tr>
<td>r</td>
<td>r</td>
<td>01110010</td>
<td>72</td>
<td>OPT-e/i</td>
<td>í</td>
<td>10010010</td>
<td>92</td>
</tr>
<tr>
<td>s</td>
<td>s</td>
<td>01110111</td>
<td>73</td>
<td>OPT-'/i</td>
<td>í</td>
<td>10010111</td>
<td>93</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>01110100</td>
<td>74</td>
<td>OPT-'/i</td>
<td>í</td>
<td>10010100</td>
<td>94</td>
</tr>
<tr>
<td>u</td>
<td>U</td>
<td>01110101</td>
<td>75</td>
<td>OPT-u/i</td>
<td>í</td>
<td>10010101</td>
<td>95</td>
</tr>
<tr>
<td>v</td>
<td>v</td>
<td>01110110</td>
<td>76</td>
<td>OPT-n/n</td>
<td>ň</td>
<td>10011110</td>
<td>96</td>
</tr>
<tr>
<td>w</td>
<td>w</td>
<td>01110111</td>
<td>77</td>
<td>OPT-e/o</td>
<td>ó</td>
<td>10011101</td>
<td>97</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>01111000</td>
<td>78</td>
<td>OPT-'/o</td>
<td>ó</td>
<td>10011100</td>
<td>98</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
<td>01111001</td>
<td>79</td>
<td>OPT-'/o</td>
<td>ó</td>
<td>10011101</td>
<td>99</td>
</tr>
<tr>
<td>z</td>
<td>z</td>
<td>01111010</td>
<td>7A</td>
<td>OPT-u/o</td>
<td>ó</td>
<td>10011110</td>
<td>9A</td>
</tr>
<tr>
<td>SHIFT-\</td>
<td>{</td>
<td>01111011</td>
<td>7B</td>
<td>OPT-n/o</td>
<td>ó</td>
<td>10011111</td>
<td>9B</td>
</tr>
<tr>
<td>SHIFT-\</td>
<td></td>
<td></td>
<td>01111100</td>
<td>7C</td>
<td>OPT-e/u</td>
<td>Ù</td>
<td>10011100</td>
</tr>
<tr>
<td>SHIFT-\</td>
<td>}</td>
<td>01111101</td>
<td>7D</td>
<td>OPT-'/u</td>
<td>Ù</td>
<td>10011101</td>
<td>9D</td>
</tr>
<tr>
<td>delete*</td>
<td>~</td>
<td>01111110</td>
<td>7E</td>
<td>OPT-'/u</td>
<td>Ù</td>
<td>10011110</td>
<td>9E</td>
</tr>
</tbody>
</table>

*A non-printing character
†Accented characters which are useful for foreign languages are generated by a two-key sequence. You must first press the OPTION key and the modifier (',i,u,n, or e) together; nothing will appear on the screen. Then press the key above which you wish the accent to appear.
## PRINTING CHARACTERS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT-t</td>
<td>µ</td>
<td>1011000000</td>
<td>A0</td>
<td>SHFT-OPT-/&gt;</td>
<td>µ</td>
<td>110000000</td>
<td>C0</td>
</tr>
<tr>
<td>SHFT-OPT-µ</td>
<td>101000001</td>
<td>A1</td>
<td>OPT-1</td>
<td>i</td>
<td>110000001</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>OPT-4</td>
<td>ø</td>
<td>101000100</td>
<td>A2</td>
<td>OPT-1</td>
<td>i</td>
<td>110000010</td>
<td>C2</td>
</tr>
<tr>
<td>OPT-3</td>
<td>£</td>
<td>10100011</td>
<td>A3</td>
<td>OPT-&lt;&gt;</td>
<td>√</td>
<td>11000011</td>
<td>C3</td>
</tr>
<tr>
<td>OPT-6</td>
<td>$</td>
<td>10100100</td>
<td>A4</td>
<td>OPT-4</td>
<td>¢</td>
<td>11000100</td>
<td>C4</td>
</tr>
<tr>
<td>OPT-8</td>
<td>*</td>
<td>10100101</td>
<td>A5</td>
<td>OPT-6</td>
<td>§</td>
<td>11000101</td>
<td>C5</td>
</tr>
<tr>
<td>OPT-7</td>
<td>¶</td>
<td>10100110</td>
<td>A6</td>
<td>OPT-7</td>
<td>£</td>
<td>11000110</td>
<td>C6</td>
</tr>
<tr>
<td>OPT-s</td>
<td>ß</td>
<td>10100111</td>
<td>A7</td>
<td>SHFT-OPT-/&gt;</td>
<td></td>
<td>11001111</td>
<td>C7</td>
</tr>
<tr>
<td>OPT-r</td>
<td>®</td>
<td>10101000</td>
<td>A8</td>
<td>OPT-\</td>
<td>«</td>
<td>11001000</td>
<td>C8</td>
</tr>
<tr>
<td>OPT-g</td>
<td>©</td>
<td>10101001</td>
<td>A9</td>
<td>OPT-;</td>
<td>...</td>
<td>11001001</td>
<td>C9</td>
</tr>
<tr>
<td>OPT-2</td>
<td>™</td>
<td>10101010</td>
<td>AA</td>
<td>(unused)</td>
<td></td>
<td>11001010</td>
<td>CA</td>
</tr>
<tr>
<td>OPT-e</td>
<td>·</td>
<td>10101101</td>
<td>AB</td>
<td>OPT-/&gt;/SHFT-a</td>
<td>A</td>
<td>11001011</td>
<td>CB</td>
</tr>
<tr>
<td>OPT-u</td>
<td>&quot;</td>
<td>10101100</td>
<td>AC</td>
<td>OPT-n/SHFT-a</td>
<td>À</td>
<td>11001100</td>
<td>CC</td>
</tr>
<tr>
<td>OPT-œ</td>
<td>×</td>
<td>10101101</td>
<td>AD</td>
<td>OPT-n/SHFT-o</td>
<td>Ô</td>
<td>11001101</td>
<td>CD</td>
</tr>
<tr>
<td>SHFT-OPT-œ</td>
<td>Æ</td>
<td>10101110</td>
<td>AE</td>
<td>SHFT-OPT-q</td>
<td>ß</td>
<td>11001110</td>
<td>CE</td>
</tr>
<tr>
<td>SHFT-OPT-ø</td>
<td>Ø</td>
<td>10101111</td>
<td>AF</td>
<td>OPT-q</td>
<td>ð</td>
<td>11001111</td>
<td>CF</td>
</tr>
<tr>
<td>OPT-5</td>
<td>₩</td>
<td>10110000</td>
<td>B0</td>
<td>OPT--</td>
<td></td>
<td>11010000</td>
<td>D0</td>
</tr>
<tr>
<td>SHFT-OPT-₪</td>
<td>±</td>
<td>10110001</td>
<td>B1</td>
<td>SHFT-OPT--</td>
<td></td>
<td>11010001</td>
<td>D1</td>
</tr>
<tr>
<td>OPT-ù</td>
<td>≤</td>
<td>10110010</td>
<td>B2</td>
<td>OPT-~</td>
<td>&quot;</td>
<td>11010010</td>
<td>D2</td>
</tr>
<tr>
<td>OPT-ν</td>
<td>≥</td>
<td>10110101</td>
<td>B3</td>
<td>SHFT-OPT-[</td>
<td>&quot;</td>
<td>11010101</td>
<td>D3</td>
</tr>
<tr>
<td>OPT-y</td>
<td>¥</td>
<td>10110100</td>
<td>B4</td>
<td>OPT-[</td>
<td>&quot;</td>
<td>11010100</td>
<td>D4</td>
</tr>
<tr>
<td>OPT-[</td>
<td>&quot;</td>
<td>10110101</td>
<td>B5</td>
<td>SHFT-OPT-]</td>
<td>&quot;</td>
<td>11010101</td>
<td>D5</td>
</tr>
<tr>
<td>OPT-d</td>
<td>Є</td>
<td>10110110</td>
<td>B6</td>
<td>OPT-/&gt;</td>
<td>+</td>
<td>11011110</td>
<td>D6</td>
</tr>
<tr>
<td>OPT-w</td>
<td>Σ</td>
<td>10110110</td>
<td>B7</td>
<td>SHFT-OPT-\</td>
<td>Ô</td>
<td>11011011</td>
<td>D7</td>
</tr>
<tr>
<td>SHFT-OPT-p</td>
<td>Π</td>
<td>10111000</td>
<td>B8</td>
<td>OPT-u/y</td>
<td>Ÿ</td>
<td>11011000</td>
<td>D8</td>
</tr>
<tr>
<td>OPT-p</td>
<td>π</td>
<td>10111001</td>
<td>B9</td>
<td>SHFT-OPT-‘</td>
<td>ß</td>
<td>11011001</td>
<td>D9</td>
</tr>
<tr>
<td>OPT-b</td>
<td>Ё</td>
<td>10111010</td>
<td>BA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPT-9</td>
<td>µ</td>
<td>10111011</td>
<td>BB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPT-0</td>
<td>ø</td>
<td>10111100</td>
<td>BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPT-z</td>
<td>Ω</td>
<td>10111101</td>
<td>BD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPT-’</td>
<td>≈</td>
<td>10111100</td>
<td>BE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPT-o</td>
<td>ø</td>
<td>10111111</td>
<td>BF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*The picture that appears on the screen varies with the type font in use.*
Hexadecimal Numbers

Base 16 (hexadecimal or simply "hex") presents a unique challenge to we human beings. This numbering system should be able to express the quantities 0 through 15 in a single place, but we only have ten digits available (0 through 9). Therefore, we use the letters A–F to represent 10 through 15 respectively. Figure 2.2 shows some hexadecimal place values. The sample number has a decimal (base 10) value of 77,631.

\[
\begin{array}{cccccc}
1 & 2 & F & 3 & F & \text{a hexadecimal number} \\
16^4 & 16^3 & 16^2 & 16^1 & 16^0 & \text{Base Sixteen place values} \\
65,536 & 4096 & 256 & 16 & 1 & \text{Base Ten equivalents}
\end{array}
\]

To convert Base Sixteen (Hexadecimal) to Base Ten (decimal):

Multiply each hexadecimal digit by its decimal equivalent and add:

\[
(65,536 \times 1) + (4096 \times 2) + (256 \times 15) + (16 \times 3) + 15 = 77,631
\]

**Figure 2.2 A Hexadecimal Number**

How hex can give us a shorthand for large binary numbers is probably not instantly obvious, but consider this: the maximum quantity that a four-digit binary number can represent is 15 (in binary, 1111), which, "by coincidence," is the maximum value of a single hex digit.

Converting a binary to hex number becomes very simple. First, divide the binary number into groups of 4 digits, working from the right. Then substitute the hexadecimal equivalent for each group of 4 binary digits. That's all there is to it.

As we saw above, the maximum RAM address in the 128K Macintosh is 1111111111111111 in binary. Figure 2.3 shows its conversion to hexadecimal. Now the maximum address appears as $1FFFF. The $ in front of the number alerts us (and the Assembler) that what follows is hexadecimal. The hex figure is certainly more manageable than that string of seventeen 1's. Though the MDS assembler will accept quantities and codes in binary, octal (Base 8), decimal, and hex, we generally specify addresses and character codes in hexadecimal and quantities in base 10.

Hexadecimal is also used as a shorthand for binary when representing ASCII codes. The digits have codes of $30$ through $39$; 0 has a code of $30$, 1 of $31$, 2 of $32$, and so on. The hexadecimal values of the codes seem much more logical than the base 10 codes of 48–57.
To do the conversion:

1. Divide the binary number into groups of four, starting from the right.
2. Substitute the corresponding hexadecimal digit for each group of four binary digits.

![Binary to Hexadecimal Conversion](image)

**Figure 2.3 Converting Binary to Hexadecimal**

By this point it has probably occurred to you that if the maximum RAM address is $1FFFF$, there is no way to specify such an address in one byte (the maximum hex value for one byte is $FF$); it will take three bytes. We also would like to be able to do arithmetic on numbers more than one byte in length (e.g., with values greater than 255, occupying more than eight binary places). The microprocessor used in the Macintosh conveniently allows us to work with words and longwords.

A word refers to two bytes (16 bits) and always begins on a byte with an even address. For example, a word could occupy the bytes at $33AA$ and $33AB$ but not the bytes at $33AB$ and $33AC$. We number the bits in a word 0–15, starting from the right. Bits 0–7 are referred to as the "low-order" byte; 8–15 are called the "high-order" byte.

A longword is 4 bytes (32 bits). Like a word, it must begin on a byte with an even address. The bits are numbered 0–31, starting at the right. Bits 0–15 are the low-order bits and 16–31 the high-order bits.

**Octal Numbers**

The octal numbering system (also known as base 8) has been around computers as long as hex, but it isn't used a great deal any more. Like hex, octal became popular as a shorthand for binary. It was useful when the largest bit grouping was a byte and when data codes were only 6 bits. Why resurrect octal here, then? Because the MOS assembler will accept octal numbers as well as binary, decimal, and hexadecimal numbers.
Since octal is base 8, it uses the digits 0 through 7. Each octal place therefore represents a power of 8 (just like binary places are powers of 2 and hex places are powers of 16). A sample octal number can be found in Figure 2.4. Its decimal value is 36,545.

To convert Base Eight (octal) to Base Ten (decimal):

Multiply each octal digit by its decimal equivalent and add:

\[(32,768 \times 8) + (4096 \times 0) + (512 \times 7) + (64 \times 3) + (8 \times 0) + 1 = 36,545\]

Figure 2.4 An Octal Number

Converting binary to octal is very much like converting binary to hex. While it takes four binary places to represent the full range of hex digits (0–F), it takes only three binary places to get the octal digits (111 base 2 = 7 base 8). Therefore, to do the conversion, divide a binary number into groups of three (starting from the right, just as when converting to hex), then substitute the appropriate octal digit for each group of three binary digits. An example of a binary to octal conversion appears in Figure 2.5.

To do the conversion:
1. Divide the binary number into groups of three, starting from the right.
2. Substitute the corresponding octal digit for each group of four binary digits.

Figure 2.5 Converting Binary to Octal
Macintosh's Microprocessor

A microprocessor is not a microcomputer. The term microcomputer refers to the whole machine, while the microprocessor is only a part of a microcomputer. In fact, a microcomputer needs not only a microprocessor, but also some RAM, enough code in ROM to boot the machine, pathways—known as buses—to carry data and addresses from one place to the other, some provision for I/O, and a clock.

The microprocessor, though, is truly the brain of the computer. The Macintosh’s microprocessor is Motorola's MC68000 (or just “68000”). You may read in some publicity releases that it is a “32-bit microprocessor.” That assertion is not completely true. While the 68000 has 32-bit registers (we'll get to registers shortly), its buses are smaller.

The 68000’s data bus is only 16 bits wide (this is the path along which data travel between RAM, ROM, and the microprocessor). The address bus (the path along which addresses travel from the microprocessor to RAM and ROM) is 24 bits wide.

The 24-bit address bus sets the limit on the maximum amount of memory Macintosh can address directly. These 24 bits (3 bytes) allow us to have a maximum address of $FFFFFF — 16 megabytes. Not all of this can be used for RAM, though. In order to access anything stored in ROM, the ROM must have its own address range, distinct from RAM. Macintosh has 64K of ROM which resides at $400000-$40FFFF.

Registers

Registers are special storage locations within a microprocessor. Almost all the actions a program performs on data occur while the data or their addresses are in the registers. The Macintosh’s 68000 microprocessor has four different kinds of registers: eight data registers, eight address registers, one status register, and one program counter (see Figure 2.6).

The data registers (numbered D0–D7) are used primarily for data manipulation. Because they are 32 bits wide, they can accommodate byte, word, and longword operations. The address registers (numbered A0–A7) are also 32 bits wide. In addition to allowing the data manipulation (though only on words and longwords), they can be used for addressing RAM (much more on this to come). Register A7 also has a special use with regard to the stack (see next section).

The status register is an extremely useful tool. While it is only 16 bits wide, it carries more than two bytes worth of information; the bits act individually as flags.

We say a bit is set if it has a value of 1; when we clear a bit, we make sure its value is 0. The bits in the status register are set at the end of many microcomputer operations. A program can check the condition of the bits in the status register to discover the result of executing an instruction.
Figure 2.6 Macintosh 68000 Registers
Figure 2.7 shows the 68000's status register. The eighth high-order bits are used by the computer itself and are therefore called the system byte. It contains a supervisor bit, a trace bit, and three bits which form an interrupt mask. Macintosh assembly language programmers will rarely use the system byte.

The supervisor-state bit is unnecessary because the Macintosh uses its microprocessor in a slightly unusual way. The standard 68000 microprocessor has two "modes": a user mode and a supervisor mode. A program running in the user mode is prohibited from using some of the microprocessor's instructions. The Macintosh, however, runs only in the supervisor mode. Therefore, the bit in the system byte which would ordinarily be used to switch between the user and supervisor modes is irrelevant.

The Macintosh does not recognize the 68000's trace mode. In fact, if the trace bit is set, the Macintosh will consider it a system error. (See Chapter 3 for more details on system errors.)

The interrupt mask bits are used to control which peripheral device (e.g., disk drives) can signal the CPU that they are in need of attention. The signal sent from the device is known as an interrupt, since it forces the CPU to interrupt whatever it is doing and take care of the device. Macintosh programs do not need to control interrupts through the system byte of the status register; they have a more powerful way to monitor what happens to the system. These are what the Macintosh calls events (discussed in detail in Chapter 8). Though some events are caused by hardware interrupts (e.g., inserting a disk into a disk drive, clicking the mouse
button, striking a key on the keyboard), others are generated by the operating system. The event mechanism is therefore more powerful and flexible than relying on an interrupt mask in the status register.

While a Macintosh application will probably never look at the system byte of the status register, it is virtually impossible to write an assembly language program without, at some time, consulting the user byte of the status register; the user byte is comprised of the eight low-order bits of the status register.

In the user byte, bit 0 is the carry bit. It is affected by integer addition and subtraction instructions as well as some other, less frequently used instructions. If the execution of an arithmetic instruction causes a carry out of the left-most bit (known as the most significant bit), the carry flag will be set. If there is no carry out, then the flag will be cleared.

To understand how the carry flag works, let's consider some simple binary addition. The binary addition table is very simple:

\[
\begin{align*}
0 + 0 &= 0 \\
0 + 1 &= 1 \\
1 + 1 &= 0 \text{ with a carry out of 1} \\
1 + 1 + 1 &= 1 \text{ with a carry out of 1}
\end{align*}
\]

Computers add only two numbers together at a time, working from the right-most (least significant) bit to the left, just as we do when performing decimal addition. A carry out from one bit position will cause a carry in to the bit position directly to its left. Therefore, the fourth expression above is the result of adding two 1's with a carry in from the previous bit.

Assume that a computer is executing the following addition:

\[
\begin{align*}
101010 \text{ Value 1} \\
+010010 \text{ Value 2} \\
\underline{111100} \text{ Result}
\end{align*}
\]

When the addition is performed on bit 1 (the second bit from the right) a carry is generated into bit 2, but this operation will nevertheless clear the carry bit. The most significant bit, bit 5 (since this is only a six-bit number), doesn't generate a carry. The carry bit will be set only if the carry is out of the most significant bit.

Consider, however, a slight modification to the problem:

\[
\begin{align*}
101010 \text{ Value 1} \\
+110010 \text{ Value 2} \\
\underline{1011100} \text{ Result}
\end{align*}
\]

The only change was in the most significant bit of Value 2 (it is a 1 rather than a 0 in this case). Now there is a carry out of the most significant bit. The carry flag will be set.

Another way to think of the carry bit is to visualize it as holding the value of a carry. In the first addition example above, there was actually a carry out of 0.
Therefore the carry bit is cleared. The second example caused a carry out of 1, setting the bit.

The second bit in the status register (bit 1) is the overflow flag. It is set whenever the result of an integer addition, subtraction, or division is too large to fit in the location where the result of the operation was to be stored. Other, less frequently used instructions also affect the overflow bit. While this at first may seem to be the same as the carry bit, it is not. The major difference is that the carry flag holds the value of a carry, while the overflow flag is a true flag, signaling the fact that an overflow occurred.

In many microprocessors, by the way, the distinction between the operation of the carry and overflow flags is different from that of the 68000. The carry bit is affected by operations on unsigned numbers, while the overflow flag monitors operations on signed numbers. That is not true with the 68000. The 68000's addition and subtraction instructions work only on signed numbers and affect both overflow and carry flags. While there are separate instructions for signed and unsigned multiplication and division, the multiplication instructions always clear the overflow and carry flags, regardless of the result of the operation. The division instructions, both signed and unsigned, clear the carry flag and affect the overflow flag based on the result of the operation.

Bit 2 is called the negative flag. It is set (i.e., gets a value of 1) whenever an operation produces a negative result. Note that other operations besides arithmetic ones can produce negative results. This most importantly includes comparison operations where you are trying to decide whether one quantity or character is larger than another.

The zero bit (bit 3) works very much like the negative bit. It is set whenever an operation gives a result of zero. Though it may seem a bit confusing at first, you need to remember that when bit 3 is 1, the result was 0; when bit 3 is 0, the result was non-zero. (You need to check bit 2, the negative bit, to know whether the result was negative or positive.)

Bit 4 is known as the extend bit. The extend bit functions, in most cases, just like the carry bit. It is used primarily for multiple-precision arithmetic operations (computations that span more than one longword).

Different instructions affect the status register differently. Therefore, as you learn the 68000 instruction set, you must not only be aware of what the instruction does, but also how it changes the user byte of the status register.

The Stack

As well as the registers just described, the 68000 microprocessor uses a special sort of storage area in RAM known as a stack. (Actually, the 68000 has two stacks, but the Macintosh uses only one.)

You can think of the stack as a tall silo that is 32 bits wide. Many pieces of data and address can be stored in the stack, one on top of the other (see Figure 2.8). Access to the stack is in last in, first out order.
The only path in and out is from the top!

![Stack Diagram]

**Figure 2.8 The Stack**

Register A7 is used as the stack pointer. It contains the address of the last item stored on the stack (called the "top" of the stack) so that you don't need to keep track of where the stack is physically or how many items are stored there. When writing programs, the stack pointer can be referred to as A7 or SP.

What is the stack used for? Often, the stack is used as an extra register for quick, temporary storage. (You *push* something onto the stack and *pull* it off, which sometimes leads to the image of the stack as a spring-loaded tube.) The stack is also the place where the microprocessor stores subroutine return addresses.

Have you ever wondered how a Pascal program knows where to return to when a procedure ends? Every time the program encounters a statement that calls a procedure, it pushes the address of the statement just after the call onto the top of the stack. Everytime it finds the **END** that finishes a procedure, it pulls the top address of the stack and resumes execution at that address. The last in, first out access to the stack ensures that nested procedures will return properly.

Assembly language subroutines affect the stack in exactly the same way. Whenever you issue a **JSR** (jump to subroutine) instruction, the address of the next program instruction is pushed onto the stack. The **RTS** (return from subroutine) instruction causes the address to be pulled from the stack and lets the system know where to resume the main program.
The Program Counter

The final register that an application uses is the program counter. The program counter contains the main memory address of the beginning of the statement following the one currently being executed. In other words, it is a 32-bit register that contains the address of the next program instruction. In fact, it is the contents of the program counter, often abbreviated to “PC,” that gets pushed onto the stack when you jump to a subroutine.

How Macintosh's RAM is Used

It may sound like a lot—128K RAM—but only a portion of that space is actually available to a program. Figure 2.9a shows how the Macintosh's RAM is divided between the user and the system in a 128K machine.

The bottom of RAM ($00–$FF) is used by the 68000 microprocessor for hardware exception vectors. These are rarely of concern to assembly language programmers. The next $300 bytes ($100–$3FF) are used by the operating system to store global variables that are shared by various parts of the system. (This is called the "system communication area.") There are more system globals in $800–$AFF.

The $400 bytes spanning $400–$7FF contain the System Dispatch Table. This table is the entry way to the ROM ToolBox routines. As a programmer, you don't need to know the exact address in ROM of any ToolBox routine you want to use. Instead, the assembler translates your call into a reference to the Dispatch Table (a "trap"), where the actual ROM addresses are stored. The table itself is stored in ROM and loaded into RAM when you start up the system.

At first this may seem like an extra, unnecessary step. Why look up the address in a table when a program could go to it directly just as easily? Because this arrangement gives added flexibility. If at some time in the future you upgrade your Macintosh and change the ROM, you won't have to modify any programs that use ToolBox routines. Using the Dispatch Table will also let you substitute a program of your own for any ToolBox routine. All you have to do is replace the address in the Dispatch Table with the starting address of your program (this is known as applying a patch). Since ROM can't be patched, it is essential that the Dispatch Table be in RAM in order to have the ability to change it.

The top of RAM (i.e., the high addresses $1FD00–$1FFE3) is used as a buffer for the Sound Driver. The Sound Driver is the part of the operating system that controls the sounds that come from the Macintosh's speaker. Just below the sound buffer ($1A700–$1FC7F) lies the main screen buffer. This area is used to map out what will be displayed on the screen.
Figure 2.9(a) 128K Macintosh RAM
When you use a debugger to help develop assembly language programs, it installs just below the screen buffer. (See Chapter 3 for a definition of a debugger and how to use one.) The region just below the debugger is set aside to hold data (called application globals) for an application program. The size of the area is not fixed; it is initialized when the program is loaded to allow only as much space as the program actually requires.

The remaining space, from $B00 to the beginning of the application globals, is under programmer control. At system startup, the area $B00 to $4CFF is initialized as the system heap. This area is used by the operating system when a program is running.

Under most circumstances, programs running on a 128K machine begin at $4D00, the start of the applications heap, and grow up in memory; the stack begins at the top of the application heap (below the application globals) and grows down in memory. If the program and the stack meet, then application has run out of memory. Program execution will stop, for example, if the program attempts to add anything else to the stack.

One of the most important things to understand from the preceding discussion is that there is nowhere near 128K for an application program. There are $15A00 bytes between the bottom of the application heap and the bottom of the screen buffer (about 71K), but part of this is lost to application globals and the stack. The space for source code is therefore rather limited, especially if a program needs tables of text stored in RAM.

Memory use in a 512K Macintosh is very similar to that in the 128K machine. If you look at Figure 9.2b, you'll see that the extra memory is concentrated in the application areas and the system heap. Instead of a 16.5K system heap like the 128K machine, the 512K Mac has a 46K system heap. Programs therefore generally begin at $C000 rather than at $4D00 as they do on a 128K machine. The remainder of the extra RAM is allocated to the application heap, the stack, and the various parameters and global values.

### Addressing RAM

When programming in Pascal, you don't have to worry about where data are stored in RAM. You use variable names as labels on storage locations; the loading/linking process assigns the actual addresses to the variable names, allowing a program to retrieve the data stored previously by simply specifying the particular variable wanted.

Assembly language, being closer to machine language, requires that the programmer keep track of where everything is stored in RAM. That includes not only the program itself but any data the program may need to use. Therefore, assembly languages provide a variety of ways of specifying where a data item is stored. The 68000 has thirteen different ways that fall into five general groups; these methods are known as addressing modes.
Figure 2.9(b) 512K Macintosh RAM
The purpose of the rest of this chapter is to introduce you to the 68000's addressing modes. Though, at this point, it may seem like overkill to have so many ways to indicate a main memory location, you will discover as you learn the instruction set and how to use the ToolBox and operating system routines that the flexibility that comes with these thirteen modes is essential to a well-written program.

To understand addressing, you must first know a little about the format of an assembly language statement. The format of assembly language statements is far more rigid than the format of high-level language statements. Statements are broken up into four fields. The first field, which may be left blank, is used for statement labels, known often as symbolic addresses. The second field contains the instruction mnemonic. The third field specifies either the data to be operated on or the address of where the data can be found. It often also indicates where the results of the operation should be placed. The data item itself is called the operand. The place where the operand can be found is its effective address. The fourth field is, like the label field, optional; it can be used for comments. Comment fields begin with a semicolon. Figure 2.10 shows a 68000 assembly language statement and its fields.

*This effective address field has two operands. The first, A1, is the effective address of the source operand. The second, -(SP), is the effective address of the destination operand.

Figure 2.10 Format of a 68000 Assembly Language Instruction

The instruction in Figure 2.10 takes the contents of register A1 and moves it onto the stack. The instruction therefore has two operands, one specifying the source of the data, and the other the destination. The two operands are separated by a comma. The comment ("put the pointer on the stack") is preceded by a semicolon.

To make addressing easier to understand, let's create a very simple computer—the "Extremely-Micro Computer"—to use in some of the examples. This computer has only two registers: a data register called D and an address register called A. It also has ten RAM locations, numbered in base 10 from 0 to 9.
Register Direct Modes

In register direct modes, the operand itself is loaded into either a data register or an address register.

Mode #1: Data Register Direct

Figure 2.11 shows the state of the Extremely-Micro Computer just before an operation using Data Register Direct addressing. The value 224, which is stored in RAM location 7, has been copied into the data register D. The effective address of that value is specified by simply coding:

\[ D \]

Whatever operation is indicated by the assembly language instruction will act on the value that has been stored in register D.

To do Data Register Direct addressing using the 68000 microprocessor, replace D in the Extremely-Micro Computer statement with Dn, where n is the number of the data register.

![Diagram of Data Register Direct Addressing]

Figure 2.11 Using Data Register Direct Addressing
Mode #2: Address Register Direct

Address Register Direct addressing works exactly like Data Register Direct addressing. The only difference is that the operand is contained in one of the address registers rather than in a data register. The assembly language format for an address register direct effective address is:

\( A_n \)

where \( n \) is the number of the address register.

Never use register A7 for direct addressing or for any sort of addressing that requires changing the value in a register, since it is used as the stack pointer. Register A5 always contains the address of the top of the applications globals area. It too should never be used for any sort of addressing that requires a change in the quantity stored in the register.

Register Indirect Addressing

The basic principle behind register indirect addressing is that instead of putting the operand itself into a register, a program loads the register with the address where the operand can be found. Register Indirect addressing can be done only with the address registers.

Mode #3: Address Register Indirect

To perform Address Register Indirect addressing, store the location of the operand in an address register. For example, Figure 2.12 shows the Extremely-Micro Computer just before execution of a statement using Address Register Indirect addressing.

The operand is still the quantity 224, but the contents of the address register A is 7. The 7 is a pointer to the RAM location where 224 is stored. The effective address would appear as:

\((A)\)

The parentheses are required. They can be read as “the contents of.” Therefore, \((A)\) translates to “the effective address is the contents of register A.”

For the 68000, add the number of the address register to the Extremely-Micro format:

\((A_n)\)

Be sure to replace the \( n \) with the number of the specific address register being used.
The location of the operand is placed in the address register.

Figure 2.12 Using Address Register Indirect Addressing

Mode #4: Address Register Indirect with Postincrement

When a program needs to process a series of data items, such as when data are stored in an array, Pascal makes life easy by allowing the program to step through the array by using a variable as a subscript. Since you can’t use variable names in assembly language, you might have to process the series of data values as follows:

1. Store the address of the first data value in an address register.
2. Process the value.
3. Increment the address so that it now reflects the location of the next data value.
4. Repeat steps 3 and 4 until all data values have been processed.

Address Register Indirect with Postincrement addressing, more simply called “Postincrement” addressing, is one way to do steps 2 and 3 with only one assembly
Prior to executing this statement, load the location of the first data value into register A. Suppose, for example, we want to process the values in RAM locations 0–4. Figure 2.13 shows the state of the Extremely-Micro Computer just before beginning that processing; 0 has been stored in register A, since it is the lowest address in the series we want to process.

![Diagram of address register indirect with postincrement addressing]

Figure 2.13 Using Address Register Indirect with Postincrement Addressing

When the computer executes the statement that processes the data, not only will the operation specified by the instruction be performed, but the address in register A will be increased by one, so that register A will then contain the address of the next value. First the operation is performed, then the address is incremented (thus the word "postincrement" in the name of this addressing mode).

While we've been using the Extremely-Micro Computer, we haven't worried about the size of the operands. The precise operation of Postincrement addressing, though, does depend on operand size. When the instruction specifies an
operation on one byte, the increment will be only one byte. For word operations, the increment will be two bytes; for longword operations, the increment will be four bytes.

**InstructionMnemonic.B (An)**

 describes an operation on a byte. (As always, the n should be replaced by the number of the address register being used.) Note that this is not a complete assembly language statement; many statements include not only the effective address of an input (source) operand, but the destination location for the results of the operation.

**InstructionMnemonic.W (An)**

\[ (An) + \]

= operation on a word

**InstructionMnemonic.L (An)**

\[ (An) + \]

= operation on a longword

We will discuss when to use which extension (.B, .W, or .L) as we discuss the individual 68000 instructions.

**Mode #5: Address Register Indirect with Predecrement**

Address Register Indirect with Predecrement addressing ("Predecrement" for short) is very similar to Postincrement addressing. When you use Predecrement addressing, the address found in the address register is decremented (decreased) prior to performing the operation specified by the assembly language instruction. The size of the decrement (byte, word, or longword) depends on the extension you put on the instruction mnemonic, just like it does with Postincrement addressing.

Predecrement addressing is specified by:

\[ - (An) \]

where \( n \) = address register number.

**Mode #6: Address Register Indirect with Displacement**

The two types of Displacement addressing available on the 68000 are additional ways to easily address data in a series of memory locations. Suppose (for whatever reason) your data are placed in every other location, as they are in the Extremely-Micro Computer example in Figure 2.14. Predecrement and Postincrement addressing will only let a program move one location at a time, but in this case you want to move two. What can you do?

Address Register Indirect with Displacement addressing allows you to specify a quantity (the displacement) which will be added to the contents of the address register. In a general form, we would use:

\[ d(A) \]

where \( d \) = the displacement.
In Figure 2.14, we want to move two memory locations. Therefore, the displacement is 2 and the general form becomes:

2(A)

When the computer executes a statement using the effective address specification, the displacement (2) will be added to the contents of register A (0) to give us the effective address (2). This statement will process the operand in location 2.

When Address Register Indirect with Displacement addressing is used with the 68000, there are two restrictions on the value of the displacement. First, it must be an integer, though it can be either positive or negative. Secondly, it must occupy no more than 16 binary digits, which translates to a value of $7FFF. (That means that bit 15 is not used as a part of the quantity; it is reserved to indicate the sign of the displacement.) The 68000 format is:

\[ d(An) \]

where \( d = 16 \)-bit displacement
\[ n = \text{address register number} \]

If we use 2(A) to specify the effective address, the displacement of 2 will be added to the contents of A to generate the location of the operand.

Figure 2.14 Using Address Register Indirect with Displacement Addressing
Really, then, what good is Address Register Indirect with Displacement addressing? It comes in handy when you want to access data in file structures.

Assume, for example, that you are working with a direct access file. The file will have a fixed number of bytes allocated for each field. (Without fixed field lengths you can't do direct access.) The file might have the following fields:

<table>
<thead>
<tr>
<th>Name</th>
<th>25 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>1 byte</td>
</tr>
<tr>
<td>Sex</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

You want to read an entire 27-byte record at one time from the disk into main memory. How, then, can you retrieve one particular field? If you know how many bytes any given field is offset from the beginning of the record, you can use Address Register Indirect with Displacement addressing to locate the field you want.

To locate the Age field, first load the starting address of the record into address register A2. Then specify the effective address of the Age field by using:

\[ 25(A2) \]

Note that while Age is the 26th byte of the record, it is offset only 25 bytes from the first byte in the record.

We'll see much more of this technique when we talk about the File Manager in Chapter 11.

**Mode #7: Address Register Indirect with Index**

Address Register Indirect with Index addressing (the other form of displacement addressing) adds an additional wrinkle. The effective address will not only be the sum of the contents of an address register and a displacement, but the contents of an index register will also be needed. An index register is any data or address register that you decide to use to hold an index value. That, by the way, isn't as much a circular definition as it might seem at first glance.

Consider the Extremely-Micro Computer example in Figure 2.15. Suppose we want to process the values in locations 3–6. We load the address 3 into register A. We load a starting index value of 0 into register D. (In this case, we don't have any choice of what register to use as an index register since we only have two and we must use the address register to hold the memory address.) The effective address is computed as shown in 2.15(a). The address in register A (3) is added to the displacement (in this example, 0) which is added to the value in register D (also 0). This instruction will therefore process the value stored in memory location 3.

In order to process the next memory location, all we need to do is increment the value in register D. (As you'll see in Chapter 4, the incrementing can be done with a single 68000 statement.) In 2.15(b) register D contains a value of 1. When we repeat the same instruction, the effective address becomes 4. Note that though this example used a displacement of zero; in practice you may use other values.
The 68000 form of Address Register Indirect with Index addressing is:

\[ d(A_n,R_n) \]

\[ d = \text{displacement} \]
\[ n = \text{register number} \]
\[ R = \text{either "A" or "D"} \]

When using this addressing mode, you are limited to an 8-bit displacement (a range of \(-128\) to \(+127\)). The \( R \) above should be replaced by either an \( A \) if you are using an address register, or \( D \) if you are using a data register for the index register.

We'll see this mode in action at the end of Chapter 5 when we discuss the handling of arrays.
Absolute Data Addressing

Absolute Data addressing allows you to follow the instruction mnemonic with the actual address of the operand. No registers are needed.

Mode #8: Absolute Short Address

To use Absolute Short addressing, follow an instruction mnemonic with 16-bit address:

**InstructionMnemonic.W 16-bit address** (Remember that there may also be a destination specified in the 68000 statement along with the address of the operand)

The assembler "extends" this address to a 24-bit effective address by copying bit 15 into bits 16–31 of the next word. (Though the extension is to a full 32 bits, only 24 can be used for an address since the 68000 has that 24-bit address bus.)

The extension means that when a program uses absolute short addresses of $0000 to $7FFF, the effective address will be in the range $000000 to $007FFF. To understand why, we need to look at the binary equivalent of these addresses.

$7FFF = %0111 1111 1111 1111

Bit 15 is 0. When we extend that value, we get an effective address of:

%0000 0000 0111 1111 1111 1111 or $007FF.

But look at what happens if we specify an address of $8000:

$8000 = %1000 0000 0000 0000.

After the extension we get:

%1111 1111 1000 0000 0000 0000 or $FF8000.

In other words, when a program uses Absolute Short addressing on an address in the range $8000 to $FFFF, the assembler generates an effective address of $FF8000 to $FFFFFF. But the 128K Macintosh has a maximum RAM address of $1FFFFF and the 512K Mac a maximum of $7FFFFF. For all practical purposes, then, this addressing mode is only good for addresses in the lower portion of memory — $0000 to $7FFF.

Mode #9: Absolute Long Address

You can still use absolute addressing, even though Absolute Short addressing won't access the entire address range, by using Absolute Long addressing.
Absolute Long addressing has the form:

```
InstructionMnemonic.L  32-bit address
```

The .L following the instruction mnemonic tells the assembler not to extend whatever address follows. Therefore, the address specified will be used as the effective address without any changes.

### Symbolic Addresses

In most applications, you will never use either absolute addressing mode. In fact, it is not only possible, but desirable to write programs without reference to absolute addresses. Instead, you will use what are known as *symbolic addresses*.

A symbolic address is a name (or label) assigned to either a program instruction or a main memory location where some data are sorted. Through the assembly and linking processes, the symbolic addresses are translated into absolute addresses in object code. But when writing the program, you need not worry about specific RAM locations. You can refer to the address of any instruction in the program by simply using its label; you can refer to the storage location of a piece of data by using the name you assigned to it. You can also assign symbolic addresses to data structures. There is much, much more about this in Chapters 4 and 5.

For example, suppose a program has just performed a comparison operation to determine if two quantities are equal. If they are not equal, the program should branch to another portion of the program. The mnemonic for an unconditional branch is **BRA**. You could write the instruction using an absolute address:

```
BRA $A123
```

This statement assumes that you know exactly what program instruction begins at memory location **$A123**. If you change your program (perhaps you had an error to correct), it's likely that many of the instructions will shift their places in RAM. What you originally expected to find at **$A123** will no longer be there.

If however, you write the statement as:

```
BRA Label1
```

then the program will branch to whatever instruction has **Label1** in its label field. **Label1** is a symbolic address. It will be replaced by an absolute address in the object code when the program is assembled and linked.

Symbolic addresses can be used anywhere an absolute address is required. There are rules for constructing legal symbolic addresses:

1. If the symbolic address does not begin in column 1 (at the far left of the Editor's input window), you must follow it with a colon.
2. There is no limit to the number of characters in a symbolic address, but for practical considerations, attempt to keep them to under 15 characters. All characters are significant.

3. The first character must be a letter, period (.), or under bar (_).

4. All other characters must be selected from among letters, numbers, periods, underbars, and dollar signs. Blanks are not allowed.

5. Symbolic addresses must not be the same as 68000 instructions, nor can they duplicate the names of ToolBox or operating systems routines.

**Program Counter Relative Addressing**

As you remember, the program counter is a special register that holds the main memory address of the start of the next program instruction to be executed. The 68000 microprocessor has two addressing modes that let you specify effective addresses as relative to the current contents of the program counter.

**Mode #10: Program Counter with Displacement**

Program Counter with Displacement addressing works very much like Address Register Indirect with Displacement addressing (mode #6). The 68000 format for specifying an effective address is:

\[ d(\text{PC}) \quad d = \text{displacement} \]

The assembler computes the effective address by adding the displacement to the current contents of the program counter.

As with displacement addressing using an address register, the displacement must be a 16-bit integer. You should also note that the expression \((\text{PC})\) is used exactly as shown. (Remember that the parentheses mean "the contents of," so \((\text{PC})\) means "the contents of the program counter.")

**Mode #11: Program Counter with Index**

This second program counter mode is also analogous to an address register mode — Address Register Indirect with Index addressing (mode #7). The effective address is the sum of the contents of the program counter, a 16-bit displacement, and contents of an index register. (You may use either a data or an address register.)

The effective address specification must indicate whether the index value is 16 bits. Therefore, the 68000 format has two possible forms:

\[ (\text{PC},\text{Rn}.W) \quad \text{or} \quad (\text{PC},\text{Rn}.L) \]

\[ d = \text{displacement} \]

\[ R = \text{either A or D} \]

\[ n = \text{register number}. \]
Just like other addressing modes that use a displacement, the displacement may be a positive or negative integer.

The Macintosh has a variation on Program Counter with Index addressing that is not standard for the 68000 microprocessor. If you specify an effective address as:

\[ d(Dn) \]

\[ d = \text{displacement} \]

it will assemble as if you had written:

\[ d(\text{PC}, Dn) \]

Though this shorthand for Program Counter with Index addressing looks like a Data Register Indirect with Displacement mode, it is not. There is no Data Register Indirect with Displacement addressing available with the MC68000 chip; that form of addressing can be performed only with an address register.

**Immediate Data**

Using immediate data doesn't qualify as addressing RAM, though it's usually discussed along with the other address modes. When you use immediate data, the operand itself is part of the assembly language statement.

**Mode #12: Immediate**

The major problem when using immediate data is finding a way to indicate the difference between immediate data and absolute addressing. In other words, how will the assembler know the difference between:

\[ $FF \]

when the \$FF refers to RAM location $0000FF and:

\[ $FF \]

when the \$FF refers to the quantity 255? To avoid the confusion, all immediate data is preceded by a \#. Therefore, the quantity 255 should be written:

\[ #$FF \]

If you have assigned symbolic addresses to data, you can use those symbolic addresses instead of the actual values. For example, to set the output type font you need to give the TextFont routine a code number that represents the font you want. Remembering the codes is difficult, so each one is assigned a symbolic address. The font called Geneva is coded as 3. We could specify that font as \#3.
But if we assign the value 3 to the symbolic address geneva, then we can use #geneva to represent the actual quantity associated with that address.

Immediate data can be character (or string) data as well as quantities. Strings are surrounded by paired single or double quotes. For example:

#'AB' or #"AB"

will assemble as the ASCII codes of the characters A and B. Strings occupy one byte of space per character.

**Mode #13: Quick Immediate**

The expression "quick immediate" refers to a special type of immediate data. Some of the 68000 instructions have a variation that embeds the operand into the machine language instruction code (the op code) itself upon assembly, though the specification of the operation in the source code is the same as standard immediate data.

Because the operand becomes a part of the op code, quick immediate data is limited to very small operands. Just how small depends on the individual instruction.

Why are quick immediate instructions of any use? They save space. A statement using immediate data takes a minimum of two words when assembled (one for the op code and one for the data); if there is a destination for the result specified in the instruction then at least three words will be needed. Quick immediate instructions use one less word of space, since op code and data assemble into a single word rather than two.

---

**Questions and Problems**

1. Convert the following decimal numbers to binary. Then convert the binary to octal and hexadecimal.

   a. 8  
   b. 19  
   c. 67  
   d. 136  
   e. 506  
   f. 695  
   g. 1023  
   h. 1028

2. Convert the following hexadecimal numbers to binary.

   a. 00FC  
   b. 0A03  
   c. E216  
   d. FFAD  
   e. CC12  
   f. 2390  
   g. 01AE  
   h. D333
A. Consider the user byte of the 68000's status register. Assuming that the unused bits (5-7) are always cleared, show the contents of the user byte when the execution of a word-sized instruction produces a result of:

a. -6  
b. 28  
c. 0  
d. 40,000  
e. -65,000

B. It's difficult to determine the value of one of the five flags without knowing exactly what kind of instruction was executed. Which flag is it?

A. If a microcomputer has a 16-bit address bus, what is the maximum address that bus can carry? Express your answer in hexadecimal.

B. What is the maximum address that a 32-bit address bus can carry?

Problems 5 and 6 refer to the Extremely-Micro Computer. As you will remember, it has an address register, A, and a data register, D. Main memory consists of storage locations numbered 0 through 9.

5. Assume that A contains 6 and D contains 2.

A. What location is indicated by each of the address specifications below?

a. 6  
b. #6  
c. A  
d. (A)  
e. 2(A)  
f. 1(A,D)  
g. D  
h. -(A)

B. Which of the 68000's addressing modes is being used?

6. Assume now that A contains A, D contains 3, and the program counter (PC) contains 2. Repeat questions A and B from problem 5 for the following effective address specifications.

a. 2  
b. (A)  
c. -(A)  
d. 2(PC)  
e. 2(PC,D)  
f. (D)  
g. 2(PC,A)  
h. #2

7. A. What effect will the effective address specification (SP)+ have on the 68000 register A7?

B. What effect will -(SP) have on register A7?
8. Indicate whether the following are legal or illegal 68000 effective address specifications. For each illegal specification, state why it is illegal.

a. D6  
b. D8  
c. (D3) +

d. A0  
e. (A) +  
f. (A4) +

g. (A4) –  
h. (D0)  
i. 6(A4)

j. – 8(A4)  
k. – 256(A4)  
l. – 256(A4,D3)

9. Assuming that a program is performing word-sized operations, what address will be generated by the assembler from the following absolute short addresses?

a. 0023  
b. A100  
c. FF39  
d. EE9B
Chapter Objectives

1. To learn the steps needed to create a Macintosh Assembly language application
2. To acquire proficiency in using the Macintosh 68000 Development System
3. To understand the purpose of a debugger and how it is used to aid program development

Introduction

This chapter is designed to familiarize you with the software that supports assembly language programming on the Macintosh. Though you can work with this software with only the internal disk drive, you will find that adding the external drive will save a great deal of disk-swapping and file-moving frustration. The figures in this chapter assume that you are using a two-disk system, though you will find instructions for shuffling files for operating with only one.

The software will run quite acceptably on a 128K Macintosh with one exception (see the discussion on debugging toward the end of this chapter). The 128K will, however, severely limit the size of application that can be developed. If you intend to pursue Macintosh program development beyond the course you are now taking, you should seriously consider upgrading a 128K.
Regardless of what size machine you are using, you should install the programmer's switch. That's the mysterious little piece of plastic that came with your Mac but without instructions. The programmer's switch snaps into place through the slots on the left hand side of the machine, all the way back and down. Place it so that the switch labeled RESET is toward the front of the machine. Pressing the RESET button will allow you to restart the system after a system error or when it is "hung" without having to turn the power off and on again. The other button, INTERRUPT, can be used to invoke the debugger.

To get the most out of the rest of this book, practice using the software now, before you become concerned with the 68000 instruction set. A sample program to be entered, assembled, linked and run appears in Listing 3.1. This program opens a window, prints a line of text, and then waits for the user to hit any key or click the mouse button before returning to the Finder.

Listing 3.1 Sample Assembly Language Program

```
Include MacTraps.D ;Includes addresses of ToolBox routines
Include ToolEqu.D ;Includes the ToolBox equates
Include SysEqu.D ;Includes the System equates

PEA -4(A5) ;Initializes QuickDraw
_InitGraf
_InitWindows ;Initializes the Window Manager
_InitMenus ;Initializes the Menu Manager
_InitFonts ;Initializes the Font Manager

CLR.L -(SP) ;Clear space for WindowPtr result
PEA StoragePointer ;Window Storage pointer
PEA BoundsRect ;Exterior coordinates of window
PEA 'MAL Output Window' ;Title
ST -(SP) ;Make the window visible
MOVE #documentProc,-(SP) ;Make it a standard document window
MOVE.L #-1,-(SP) ;Put the window in front
ST -(SP) ;Draw a go-away box
CLR.L -(SP) ;Place for window's reference value
_NewWindow ;Draw a standard document window

LEA WindowPtr,A0 ;load destination address for pointer
MOVE.L (SP)+,(A0) ;retrieve pointer

MOVE.L WindowPtr,-(SP) ;SelectWindow
MOVE.L WindowPtr,(SP) ;Put pointer back on the stack
_SetPort ;make this window the current grafport
_InitCursor ;set the cursor to the arrow
```

(continued)
MOVE.W #7,-(SP) ;7 = athens
_TextFont
;Set the text font

MOVE.W #18,-(SP) ;18 for 18-point type
_TextSize
;Set the text size

MOVE.W #65,-(SP) ;Horizontal coordinate
MOVE.W #100,-(SP) ;Vertical coordinate
_MoveTo
;Move the pen

PEA 'HOORAY!! You did it!'
_DrawString

MOVE.L everyEvent,D0 ;Mask to select all events
_DrawString

_FLUSHEvents
;Clear the event queue

Event CLR -(SP) ;Space for boolean result
MOVE #%0000000000111110,-(SP) ;Mask for keyboard and mouse
PEA EventRecord ;Place to receive event info
_GetNextEvent ;Get next event from queue

MOVE (SP)+,D0 ;Has a keyboard or mouse event occurred?
CMP #0,D0
BEQ Event ;If no event, branch to look again

RTS ;Return to the Finder

WindowPtr DC.L Ø
BoundsRect DC.W 40,20,300,350
everyEvent DC.L $0000FFFF
EventRecord DC.B windowSize,0 ;where GetNextEvent Puts its result
What DC Ø
Message DC.L Ø
When DC.L Ø
Point DC.L Ø
Modify DC Ø

StoragePointer DCB.W windowSize,Ø

END

The Macintosh 68000 Development System (the MDS) is the formal name for the set of programs that enable a programmer to enter, assemble, link, and run assembly language programs. It also includes a family of debuggers, programs that, among other things, display what's happening in the Macintosh's registers while a program is running.
On the disk named MDS1 (Figure 3.1) you will find:

1. the Editor (Edit) — allows you to enter assembly language source programs.
2. the Executive (Exec) — automates the assembling and linking process
3. the Assembler (Asm) — translates source code created by the Editor into binary object code
4. the Linker (Link) — links separately assembled modules of source code into an executable application
5. the Resource Compiler (RMaker) — creates files that define windows, menus, etc.
6. Debug Nubs — files used by some of the debuggers
7. Assembler Support Files (in the folder ASM Stuff)

![Image of MDS1 disk]

**Figure 3.1** The Disk MDS1

The disk named MDS2 (Figure 3.2) contains:

1. the Macintosh Debuggers (in the folder Debuggers)
2. the Equates Files (in the Equ Files folder) — handy definitions that the ToolBox uses
3. the Symbol Packer (PackSyms) — a program that compacts Equates Files so they will take up less room in your source files
4. Packed Symbol Files (in the .D Files folder) — what you get when you put Equates Files through the Symbol Packer

5. Trap Files (in the Trap Files folder) — files that assign names to the instruction words that reference the ToolBox Dispatch Table

6. some Sample Programs

![App Icon]

Figure 3.2 The Disk MDS 2

---

**Using the Editor**

The Macintosh 68000 Development System comes with its own text editor for creating program source files. You may also use MacWrite, but save the document as text only, without any formatting information. The MDS editor is "disk based." That means you can edit files much larger than what will fit in RAM; the editor shuffles bits and pieces of text between the disk and RAM as needed.

Invoke the editor by double-clicking on its icon. (There are two other ways to get into the editor, but this will do for now.)

Assembly language source files are more or less free form (i.e., there are no set columns in which particular parts of the statements must appear). The only rules are:

1. The first field is reserved for symbolic addresses. If a statement doesn't have a symbolic address, then it must begin with at least one blank. Symbolic
addresses don't necessarily have to start in column one (the far left-hand position on the screen), but if they don't, they must be followed by a colon (:) .

2. The second field is reserved for the instruction mnemonic. It must be separated from the symbolic address (if one is present) by at least one space.

3. The third field holds one or more operands (either the operands themselves or their effective addresses). The operand field must be separated from the mnemonic by at least one space.

4. The fourth field may contain a comment. Comments begin with semicolons (;) and must be separated from the operand field by at least one space. You may also have a line in your source file that is all comment. In that case you must either have a semicolon or an asterisk (*) in column one.

For readability, we usually line up the fields. The MDS editor comes with preset tab stops which can be changed by using the `FORMAT` menu (see Figure 3.3).

To make indentation to the mnemonic field easier, the editor also provides automatic indentation. Once you have tabbed to a particular spot without entering text in any preceding tab zone, the RETURN key will place the cursor at that tab stop instead of in column one. To type something to the left, hit the BACKSPACE key. Automatic indentation can be turned off from the `FORMAT` menu (Figure 3.3).

![Figure 3.3 The MDS Editor's Format Menu](image)

The editor provides some basic features for changing source code. Cut, copy, and paste work just as they do in MacWrite. You can also align all the text in a selected block (select with the mouse as when using MacWrite) with options
available from the EDIT menu (see Figure 3.4). The SEARCH menu (Figure 3.5) provides standard find and change capabilities.

When you have finished entering the sample program, save it to disk. The FILE menu (Figure 3.6), just like the MacWrite FILE menu, allows you to name the file before you save it.
How you name your file is important. The various programs that make up the Macintosh 68000 Development System look for files with specific extensions to their names. Assembly language source files should have the extension .ASM. You could, for example, name the sample program Sample.Asm.

The Assembler

There is very little unused space on the disk MDS1. Therefore, if you are working with a single disk system, you will have to create a special disk for the assembly process. On it you should put your source file, any equates and trap files it uses (for the sample program in Listing 3.1 copy Mactraps.D, ToolEqu.D, and SysEqu.D from MDS2), the Assembler, and the folder ASM Stuff.

With a two-drive system, copy the equates and trap files onto the text disk which also holds your source file. Put the text disk in the external drive and leave MDS1 in the internal drive.

If you are in the Editor and using a two-drive system, you can invoke the Assembler from the Editor's TRANSFER menu (Figure 3.7). With a single-disk system you must copy your source file onto your special Assembler disk. You can then enter the Assembler by double-clicking on its icon from the Finder (this method will obviously also work for a two-drive system).

The Assembler will present a list of the files which it can identify as possible candidates for assembly (Figure 3.8). If you have a large number of source files on
Include MacTraps.D
Include ToolEqu.D
Include SysEqu.D

PEA -4(AS)
InitGraf ;Initializes Output Manager
InitWindows ;Initializes the Window Manager
InitMenus ;Initializes the Menu Manager
InitFonts ;Initializes the Font Manager

_Debugger
CLR.L -(SP) ;Clear space for WindowPtr result
PEA LoopStorage ;Window Storage pointer
PEA BoundsRect ;Exterior coordinates of window
PEA 'MAL Output Window' ;Title
ST -(SP) ;Make the window visible
MOVE rDocProc,-(SP) ;Make it a standard document window
MOVE. L #1,-(SP) ;Put the window in front
ST -(SP) ;Draw a go-away box
CLR.L -(SP) ;Place for window's reference value
NewWindow ;Draw a standard document window

Figure 3.7 The MDS Editor's Transfer Menu

Figure 3.8 Assembler File Select Screen
your disk, select the Filter by Time option from the FILE menu (Figure 3.9). This will display only those files that have been modified since they were last assembled. Double-click on the file name and the assembly process will begin. The assembled version of the program is written to a file with the extension .REL (e.g., assembling Sample.Asm will produce Sample.REL).

Before beginning assembly, you can make some choices about the kind of output the assembler will produce. By default you will get no listing of the assembled version of your program. If you want a listing, select it from the OPTIONS menu (Figure 3.10). The listing can be displayed on the screen or written to a file. If you choose a file listing (the smart choice, since screen listings will rapidly scroll out of sight), the listing will be written to a file with the extension .LST (e.g., a source file named Sample.Asm will generate a listing file named Sample.LST). Note that assembling with a listing significantly lengthens the time it takes to assemble a program.

The Assembler listing for the Sample program appears in Listing 3.2. The leftmost column is a line number for your reference only. The second column from the left contains the hexadecimal RAM address where each program line begins. By default, the Assembler starts all programs at $0000. This is not where the program will end up in RAM when the program is run. The operating system will add all the program locations to a fixed base address at run time.

The remaining numbers are the hexadecimal equivalents of the instruction mnemonics and their operands. You will have noticed that there are x's in some places rather than hexadecimal numbers. The x's fill in places for absolute addresses which the assembler was unable to identify. They will be replaced with addresses by the Linker when space for storage locations the applications globals area is allocated.

You can also specify that what is written to the .REL file should be the minimum necessary to create a working application (Normal Output) or that the .REL file should include extra information to permit creation of a Linker listing (Verbose Output). Verbose Output will lengthen both the assembly and linking processes.

If any errors are detected during assembly, they will be stored in a file with extension .ERR (e.g., if your source file is Sample.Asm, then the errors will be listed in Sample.ERR). The error file will be placed on the same disk as your source file. The errors will also display on the screen as they are discovered, but they generally scroll by too fast for you to read and remember them.

Though a .REL file is created for an assembly in which errors were detected, you will not be able to successfully link or execute any program with errors. Therefore, if your program has errors, return to the Editor. There you can examine the .ERR file at your leisure (printing it out if necessary) and then make the needed changes to your source file.

If you are using a two-disk system, you can return to the Editor through the Assembler's TRANSFER menu (Figure 3.11). With a single-disk system, you must transfer the .ERR file back to the disk that contains the Editor and then enter the Editor from the Finder.
Figure 3.9 Assembler File Menu

Figure 3.10 Assembler Options Menu
Listing 3.2 Assembler Listing of Sample Program

```
0000  Include MacTraps.D ;Includes addresses of ToolBox routines
0000  Include ToolEqu.D ;Includes the ToolBox equates
0000  Include SysEqu.D ;Includes the System equates
0000  486D FFFC          PEA -4(A5)
0004  A86E               _InitGraf ;Initializes QuickDraw
0006  A912               _InitWindows ;Initializes the Window Manager
0008  A930               _InitMenus  ;Initializes the Menu Manager
000A  A8FE               _InitFonts  ;Initializes the Font Manager
000C
000C  42A7               CLR.L -(SP) ;Clear space for WindowPtr result
000E  4840 XXXX           (PX) PEA StoragePointer ;Window Storage pointer
0012  4840 XXXX           (PX) PEA BoundsRect ;Exterior coordinates of window
0016  4840 XXXX           (PX) PEA 'MAL Output Window' ;Title
001A  50E7               ST -(SP) ;Make the window visible
001C  3F3C 0000           MOVE #documentProc,-(SP) ;Make it a standard document window
0020  2F3C FFFF FFFF     MOVE.L #1,-(SP) ;Put the window in front
0024  50E7               ST -(SP) ;Draw a go-away box
0028  42A7               CLR.L -(SP) ;Place for window's reference value
002A  A913               _NewWindow ;Draw a standard document window
002C
002C  41C0 XXXX           (PX) LEA WindowPtr,A0 ;Load destination address for pointer
0030  209F               MOVE.L (SP)+,(A0) ;Retrieve pointer
0032
0032  2F3A XXXX           (R) MOVE.L WindowPtr,-(SP)
0036  A91F               _SelectWindow
0038
0038  2F3A XXXX           (R) MOVE.L WindowPtr,-(SP) ;Put pointer back on the stack
003C  A873               _SetPort ;Make this window the current grafport
003E
003E  A850               _InitCursor ;Set the cursor to the arrow
0040
0040  3F3C 0007           MOVE.W #7,-(SP) ;7 = athens
0044  A887               _TextFont ;Set the text font
0046
0046  3F3C 0012           MOVE.W #18,-(SP) ;18 for 18-point type
004A  A88A               _TextSize ;Set the text size
004C
004C  3F3C 0041           MOVE.W #65,-(SP) ;Horizontal coordinate
0050  3F3C 0064           MOVE.W #100,-(SP) ;Vertical coordinate
0054  A893               _MoveTo ;Move the pen
0056  4840 XXXX           (PX) PEA 'HOORAY!!! You did it!'
005A  A884               _DrawString
005C
005C  203A XXXX           (R) MOVE.L everyEvent,D0 ;Mask to select all events
```
The Linker

A .REL file contains an object code that is *relocatable* (capable of being moved around in main memory). Though it is in the binary, machine language form that the computer will understand, it is not an executable application since many of the absolute addresses are missing. The Linker provides the final step in the process.
The Linker generates two types of output. Assuming that no errors are detected during the linking process, you will get an executable application (appears on the desktop as a diamond with a hand holding a pen) and a file with a .MAP extension. A .MAP file contains a symbol table (exactly where everything is when your program is in RAM) and also the Linker listing, if you requested one.

The operation of the Linker is determined by a Linker control file. A control file contains the names of the .REL files to be linked (you can assemble a large program in small parts and then have the Linker combine them into a single application) and, optionally, a symbolic address that indicates which instruction in your source code is the start of your program; instructions on how the program can be segmented (it is possible to break a program which is too large to fit into memory into segments which are then loaded as needed); and options that control the contents of the Linker output file.

Linker control files are text files that are created with the Editor. They must be given the extension .LINK (e.g., the Linker control file for the Sample program should be called Sample.LINK). At a minimum, a Linker control file must contain the name of the program to be linked and a $ that marks the end of the file.

For the Sample program, create a text file that contains:

```
[ Sample $
```

The [ will turn on the listing to the .MAP file and is therefore optional.

If you are working with a two-drive system, save the Linker control file on your text disk. With a single-drive system, put the .REL file, the Linker control file, and the Linker on one disk before beginning the linking process.

You can enter the Linker from the Finder, or from the Assembler's TRANSFER menu (Figure 3.11). The Linker displays a list of Linker control files on the current disk (Figure 3.12). Double-clicking on the file name will then begin the linking process.

If the Linker encounters any errors, they will be stored in a file with a .LERR extension (e.g., for the Sample program, Linker errors will be written to Sample.LERR). A .LERR file can be examined from the Editor, just like .ERR files.

If you include a [ in a Linker Control file, the .MAP file will include a program listing like the one in Listing 3.3. This listing differs from an Assembler listing in one important way: the x's in the Assembler listing have been replaced with absolute addresses. This is the version of the program that will actually run.
Figure 3.11 Assembler Transfer Menu

Figure 3.12 Linker File Select Screen
Listing 3.3 Linker Listing of Sample Program

Sample.Rel

Include MacTraps.D ;Includes addresses of ToolBox routines
Include ToolEqu.D ;Includes the ToolBox equates
Include SysEqu.D ;Includes the System equates

;Includes ToolBox routines
-4(A5)
_InitGrf
_InitWindows
_InitMenus
_InitFonts

;Includes the Window Manager
;Includes the Menu Manager
;Includes the Font Manager

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000:</td>
<td>48 6D FF FC</td>
</tr>
<tr>
<td>00000004:</td>
<td>A8 6E</td>
</tr>
<tr>
<td>00000006:</td>
<td>A9 12</td>
</tr>
<tr>
<td>00000008:</td>
<td>A9 30</td>
</tr>
<tr>
<td>0000000A:</td>
<td>A8 FE</td>
</tr>
<tr>
<td>0000000C:</td>
<td>42 A7</td>
</tr>
<tr>
<td>0000000E:</td>
<td>487A 0088</td>
</tr>
<tr>
<td>0000012:</td>
<td>487A 0068</td>
</tr>
<tr>
<td>0000016:</td>
<td>487A 01B8</td>
</tr>
<tr>
<td>000001A:</td>
<td>50 E7</td>
</tr>
<tr>
<td>000001C:</td>
<td>3F 3C 00 00</td>
</tr>
<tr>
<td>0000020:</td>
<td>2F 3C FF FF FF FF</td>
</tr>
<tr>
<td>0000026:</td>
<td>50 E7</td>
</tr>
<tr>
<td>0000028:</td>
<td>42 A7</td>
</tr>
<tr>
<td>000002A:</td>
<td>A9 13</td>
</tr>
<tr>
<td>000002C:</td>
<td>41FA 004A</td>
</tr>
<tr>
<td>0000030:</td>
<td>2D 9F</td>
</tr>
<tr>
<td>0000032:</td>
<td>2F 3A 00 44</td>
</tr>
<tr>
<td>0000036:</td>
<td>A9 1F</td>
</tr>
<tr>
<td>0000038:</td>
<td>2F 3A 00 3E</td>
</tr>
<tr>
<td>000003C:</td>
<td>A8 73</td>
</tr>
<tr>
<td>000003E:</td>
<td>A8 5C</td>
</tr>
<tr>
<td>0000040:</td>
<td>3F 3C 00 07</td>
</tr>
<tr>
<td>0000044:</td>
<td>A8 87</td>
</tr>
<tr>
<td>0000046:</td>
<td>3F 3C 00 12</td>
</tr>
<tr>
<td>000004A:</td>
<td>A8 8A</td>
</tr>
<tr>
<td>000004C:</td>
<td>3F 3C 00 41</td>
</tr>
<tr>
<td>0000050:</td>
<td>3F 3C 00 64</td>
</tr>
<tr>
<td>0000054:</td>
<td>A8 93</td>
</tr>
<tr>
<td>0000056:</td>
<td>487A 018A</td>
</tr>
<tr>
<td>000005A:</td>
<td>A8 84</td>
</tr>
<tr>
<td>000005C:</td>
<td>20 3A 00 26</td>
</tr>
</tbody>
</table>

Include ToolEqu.D
Include SysEqu.D

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000004:</td>
<td>CLR.L -(SP)</td>
</tr>
<tr>
<td>000000E:</td>
<td>PEA StoragePointer</td>
</tr>
<tr>
<td>0000012:</td>
<td>PEA BoundsRect</td>
</tr>
<tr>
<td>0000016:</td>
<td>PEA 'MAL Output Window'</td>
</tr>
<tr>
<td>000001A:</td>
<td>ST -(SP)</td>
</tr>
<tr>
<td>000002C:</td>
<td>MOVE #documentProc,-(SP)</td>
</tr>
<tr>
<td>000002C:</td>
<td>CLR.L -(SP)</td>
</tr>
<tr>
<td>0000030:</td>
<td>PEA StoragePointer</td>
</tr>
<tr>
<td>0000032:</td>
<td>CLR.L -(SP)</td>
</tr>
<tr>
<td>0000036:</td>
<td>CLR.L -(SP)</td>
</tr>
<tr>
<td>0000038:</td>
<td>MOVE.LWindowPtr,-(SP)</td>
</tr>
<tr>
<td>000003E:</td>
<td>_SelectWindow</td>
</tr>
<tr>
<td>000003C:</td>
<td>A8 73</td>
</tr>
<tr>
<td>000003E:</td>
<td>A8 5C</td>
</tr>
<tr>
<td>0000040:</td>
<td>MOVE.W #7,-(SP)</td>
</tr>
<tr>
<td>0000044:</td>
<td>_TextFont</td>
</tr>
<tr>
<td>0000046:</td>
<td>MOVE.W #18,-(SP)</td>
</tr>
<tr>
<td>000004A:</td>
<td>_TextSize</td>
</tr>
<tr>
<td>000004C:</td>
<td>MOVE.W #65,-(SP)</td>
</tr>
<tr>
<td>0000050:</td>
<td>MOVE.W #100,-(SP)</td>
</tr>
<tr>
<td>0000054:</td>
<td>_MoveTo</td>
</tr>
<tr>
<td>0000056:</td>
<td>PEA 'HOORAY!!! You did it!'</td>
</tr>
<tr>
<td>000005A:</td>
<td>_DrawString</td>
</tr>
<tr>
<td>000005C:</td>
<td>MOVE.LeveryEvent,D0</td>
</tr>
</tbody>
</table>

(continued)
USING THE MACINTOSH 68000 DEVELOPMENT SYSTEM

FlushEvents: ;Clear the event queue

_EVENT CLR -(SP) ;Space for boolean result
MOVE #%0000000000111110,-(SP) ;Mask for keyboard and mouse
PEA EventRecord ;Place to receive event info
_GetNextEvent ;Get next event from queue

occurred?

CMP #Ø,DØ
BEQ Event

RTS ;Return to the Finder

WindowPtr DC.L Ø

BoundsRect DC.W 40,20,300,350
everyEvent DC.L $0000FFFF

EventRecord ;where GetNextEvent Puts its result

Message DC.L Ø

When DC.L Ø

Point DC.L Ø

Modify DC Ø

StoragePointer DCB.W windowSize,Ø

endianness

11 4D 41 4C 20 4F 75 74 70 75 74 20 57 69 6E 64 6F 77
0001E2: 16 48 4F 4F 52 41 59 21 20 20 59 6F 75 20 64 69 64 20 69 74 21
0001F9: 00
A caveat is in order with regard to the Linker. If your attempt at linking gives system error #28, then the Finder has run out of memory (the stack has run into the heap) and cannot place your application file in the disk directory. A disk should theoretically hold somewhere near one hundred files, but if you are working with a 128K Mac you may see this error with less than 20 files on your disk. If this occurs, delete some files or transfer just the few files you absolutely need to another disk to successfully complete the linking. (.MAP, .ERR, .LST and .LERR files are good candidates for deletion.)

Running an Application

After a successful linking, there are two ways to execute an application. The successful linking will add an extra option to the Linker's TRANSFER menu (Figure 3.13). You can run the program by selecting that option. You can also run any application at any time by double-clicking its icon from the Finder.

Assuming that you have successfully entered, assembled, and linked the Sample program, your output will appear as in Figure 3.14

Run-Time System Errors

There are some errors that the Assembler's error-checking capabilities will not catch. These often don't show up until an application is running and appear as system errors that require resetting the system to recover (such as error #28 mentioned above).

For example, assume that you wanted to specify an operand as immediate data. To correct, you should have used:

InstructionMnemonic #SomeQuantity, D0

Unfortunately, you left off the # which means that your source code contained:

InstructionMnemonic SomeQuantity, D0

The Assembler interpreted the quantity as an absolute address; what was in the source file was a totally correct use of Absolute addressing. The problem, though, is that you don't want what is stored at whatever address the quantity represents; you want the quantity itself. Nonetheless, since the syntax of the statement is correct, the Assembler won't pick up the error.
Figure 3.13  Linker Transfer Menu after a Program has been Successfully Linked

Figure 3.14  Output From Sample Program
When you run the application, all sorts of strange things can happen. More often than not, a system error #02 (bad address) will occur.

You'll find the error messages associated with the system errors in Table 3.1. Though such errors are extremely difficult to interpret, the table includes some suggestions as to causes of the more common ones and their solutions.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Bus Error</td>
<td>Not applicable on the Macintosh</td>
</tr>
<tr>
<td>02</td>
<td>Address Error</td>
<td>Your program has attempted to use an address which makes no sense to the operating system (a word or longword reference has been made to an odd address). Can be caused when an immediate operand is missing its #.</td>
</tr>
<tr>
<td>03</td>
<td>Illegal Instruction</td>
<td>The code in an instruction field does not represent any instruction in the 68000's instruction set. Check immediate addressing for missing #.</td>
</tr>
<tr>
<td>04</td>
<td>Zero divide</td>
<td>Just what is says -- your program has attempted to do a division by zero.</td>
</tr>
<tr>
<td>05</td>
<td>Range Check Error</td>
<td>Failure of one particular 68000 instruction -- CHK (checks one word of a data register against an upper-bound value).</td>
</tr>
<tr>
<td>06</td>
<td>Overflow</td>
<td>Failure of one particular 68000 instruction -- TRAPV (executes a trap if the overflow flag in the status register is set).</td>
</tr>
<tr>
<td>07</td>
<td>Privilege violation</td>
<td>Not terribly important since all assembly language programs run in the &quot;supervisor&quot; mode, where you have access to all instructions.</td>
</tr>
<tr>
<td>08</td>
<td>Trace Mode Error</td>
<td>Trace mode is initiated by setting one of the bits in the user byte of the status register. The Macintosh never uses trace mode; therefore, this error will occur whenever the trace-mode bit is accidentally set.</td>
</tr>
<tr>
<td>09</td>
<td>Line 1010 Trap</td>
<td>&quot;Line 1010 Trap&quot; has to do with calling Toolbox routines (see Chapter 6).</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Line 1111 Trap</td>
<td>Another trap not used on the Macintosh. Line 1111 traps are reserved for further expansion of the instruction set (details are in Chapter 6).</td>
</tr>
<tr>
<td>11</td>
<td>Hardware Exception Error</td>
<td>The system thinks some other sort of trap has occurred. This usually means that the machine is seeing some sort of illegal binary instruction code. If you get this, check for addresses and/or operands that are the wrong size.</td>
</tr>
<tr>
<td>12</td>
<td>Unimplemented Core Routine</td>
<td>Can occur when a program invokes the debugger when the debugger isn't present in memory.</td>
</tr>
<tr>
<td>13</td>
<td>Uninstalled Interrupt</td>
<td>Can occur when a program invokes the debugger when the debugger isn't present in memory.</td>
</tr>
<tr>
<td>14</td>
<td>I/O Core Error</td>
<td>Problem with file access.</td>
</tr>
<tr>
<td>15</td>
<td>Segment Loader Error</td>
<td>Caused by failure of an attempt to load a program segment into main memory.</td>
</tr>
<tr>
<td>16</td>
<td>Floating Point Error</td>
<td>The problem lies in whatever part of the program calls FP68K, the Macintosh's floating point arithmetic package.</td>
</tr>
<tr>
<td>17-24</td>
<td>Packages 0-7 missing</td>
<td>Packages are self-contained routines present in the system (see Chapters 6, 11 and 12 for more information).</td>
</tr>
<tr>
<td>25</td>
<td>Memory Full</td>
<td>You have two options -- upgrade to 512K or segment your program into portions that don't need to be memory co-resident.</td>
</tr>
<tr>
<td>26</td>
<td>Bad Program Launch</td>
<td>Usually caused by an attempt to launch a file that isn't an executable application.</td>
</tr>
<tr>
<td>27</td>
<td>File System Map Trashed</td>
<td>Something is wrong with a disk's directory.</td>
</tr>
<tr>
<td>28</td>
<td>Stack Ran Into Heap</td>
<td>Another sort of out-of-memory error.</td>
</tr>
<tr>
<td>29</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Disk Insertion Error</td>
<td>Generates the &quot;Please insert the disk:&quot; alert.</td>
</tr>
</tbody>
</table>

(continued)
Table 3.1 (continued)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Message</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>32-56</td>
<td>Memory Manager Errors.</td>
<td>Indicate problems with the routines that manage the use of Macintosh RAM.</td>
</tr>
<tr>
<td>41</td>
<td>No Finder</td>
<td>The Finder isn't on any disk currently in the system's drives.</td>
</tr>
<tr>
<td>100</td>
<td>Bad startup disk</td>
<td>System can't boot because something is wrong with the startup disk. Causes a blank screen with a disk icon in the center. The disk icon contains a question mark.</td>
</tr>
</tbody>
</table>

The Executive

If you have been working along with this chapter, you may have decided that transferring from the Editor to the Assembler to the Linker and back again is a giant pain. There is a way, though, to "automate" most of the tedious steps in the process by using the Executive.

The actions of the Executive are controlled by a file created with the Editor and given the extension .JOB. A .JOB file has four fields, separated by tabs. The first field contains the names of the application to be executed (e.g., ASM or LINK). The second field contains what input the application requires (usually a file name). The third field is the application to which the Executive should return if the execution of the application in the first field is successful (usually the Executive). The fourth field is the application that should be executed if the execution of the application in the first field is not successful (usually the Editor).

An Executive control file for the Sample program might appear as:

```
ASM        Sample.Asm
LINK       Sample.Link
Exec       text.Disk:Sample
Edit       Edit
```

When setting up Executive control files, you need to pay attention to what disk your files are on. All applications should be on the startup disk (i.e., the internal drive). Source files (source code and Linker control files) should be on the same disk as the .JOB file (preferably on a text disk in the external drive). Because of disk space considerations, it will be very difficult to use the Executive with a single-drive system.

If you want the Executive to automatically run your program after it finishes linking (assuming your source files and the completed application are on a text disk in the external drive), precede the program's name with the name of the disk. For
example, if your text disk is named Text.Disk as in the sample .JOB file above, specify the name of the application to be created by the Linker as:

**Text.Disk:Sample**

The name of the application is separated from the name of the disk by a colon.

To initiate the actions specified in an Executive control file, enter the Executive. Usually, you will do so by either double-clicking on its icon from the Finder or transferring to it from the Editor.

The Executive's file select screen (Figure 3.15) lets you select the .JOB file to execute. Once you double-click on the file name, the process becomes automatic.

![Figure 3.15 The Executive's File Select Screen](image)

The two-line .JOB file above will perform the following actions:

1. Assemble the file **Sample.Asm**
2. If the Assembler detects errors, execute the Editor
   a. Make the file **Sample.ERR** the active window
   b. When **Sample.ERR** is closed, make **Sample.Asm** the active window
3. After a successful assembly, link the file **Sample.REL**, using **Sample.LINK** as the Linker control file

4. If the Linker detects errors, execute the Editor
   
   a. Make the file **Sample.LERR** the active window
   
   b. When **Sample.LERR** is closed, make **Sample.LINK** the active window

5. If the linking is successful, execute the completed application, **Sample**

Though using the Executive does not speed up the processes required to prepare an assembly language program (the Editor, Assembler and Linker still have to be loaded into memory every time you need them), it will decrease the amount of work you have to do. Set the Executive running and go get a soda...

The time it takes to prepare an assembly language program for execution is severely constrained by the Macintosh's disk access speed. When using the 68000 Development System as it is distributed by Apple there is no way to keep the Editor, Assembler, and Linker continuously in RAM. There are, however, two ways to get around the problem. The first addresses the problem by keeping the Editor, Assembler, and Linker in RAM; the second deals with disk access speed.

If you have a 512K Mac you can use a portion of that memory as a RAM disk. To do so, purchase *Mac Memory Disk* by Assimilation Process (available for about $30). There is just enough room on the RAM disk for the system files and the Editor, Assembler, and Linker. There is no room for the Executive; the editing, assembling, and linking process must be managed manually. That is far less of a disadvantage than it might seem. Since all three programs are in RAM, transfer between them is almost instantaneous. The major drawback to using the RAM disk is that it doesn't leave enough room in memory for a debugger.

The only way to speed up disk access time is to use a hard disk. In terms of cost, a hard disk is not always a viable option. In fact, upgrading a 128K machine to 512K and purchasing the RAM disk software will cost far less than purchasing a hard disk.

When you use the Executive, you no longer have access to the Assembler and Linker OPTIONS menus (e.g., to control listings). You must therefore specify the options you want in your source file (see the section on Assembler Directives in Chapter 4).

---

**Debugging**

An assembler, like an interpreter or compiler, checks for syntax errors as it translates source code to object code. None of the three translation programs, however, can catch logic errors; they simply aren't capable of "understanding"
what a programmer intended. Finding logic errors is therefore the toughest part of the programming. A debugger is a program designed to help the assembly language programmer with that task.

When debugging a Pascal program you may have placed writeln statements at strategic places in the code to display the contents of important variables. This allowed you to monitor the contents of the variables as they changed and helped you pin-point the exact spot in a program where something went wrong. The same strategy isn't sufficient, however, when you are working in assembly language.

Assembly language programs have much greater control over the computer than high-level language programs in the sense that as well as manipulating data storage locations (i.e., variables) they have direct access to the CPU's registers. Therefore, in order to find the source of an error it is usually necessary to see what is happening within the registers while the program is running.

A debugger is a program that, among other things, will do the following:

1. Run an assembly language program one instruction at a time
2. Display the contents of the CPU's registers after each instruction is executed
3. Display the contents of main memory locations
4. Disassemble program instructions from either RAM or ROM.

It is generally very difficult to successfully complete an assembly language program without at some point employing a debugger.

If you open the Debuggers folder on MDS2, you will find not one, but six debuggers. The best one is MacDB. Unfortunately, you need two Macintosh's hooked together to use it (one runs the program and the other runs the debugger). Of the other five, two require external terminals (TermBugA and TermBugB) and one runs on the Lisa (LisaBug). Both MidiBug and MaxBug, though, will run on a single, free-standing Macintosh.

MaxBug will run only on a 512K machine. MidiBug will run with 128K, but (and this is a very big "but") once MidiBug is installed, there is no room in memory for any other application (the Editor, Linker, etc.). Why is this such a problem? Debuggers can't be executed like other applications (i.e., by clicking an icon from the Finder). Instead, whenever you boot a disk containing a file called MacsBug (regardless of whether that file was originally MidiBug or MaxBug), that debugger will be automatically placed in memory. It will sit in memory until invoked by an "exception" in your program.

This means that whenever you want to run a program and use MidiBug with a 128K machine, you must:

1. Create a special debugging disk with a file named MacsBug on it. (Be sure the file is a renamed MidiBug since MaxBug won't fit, no matter how hard you try.)
2. Use MDS1 to boot your Macintosh and complete the assembly and linking process

3. Copy the completed application to the debugging disk

4. Reboot the system with the debugging disk as the startup disk

This long procedure would appear to be the only way to use a debugger with a 128K machine.

The presence of a debugger in memory does not necessarily mean that the debugger will be activated when you run an application. The debugger must be "invoked." Though there are several ways to do so, the easiest is to include the instruction:

```
.Debugger
```

in your source code at the point you wish the debugger to take over.

MidiBug and MaxBug provide the same kind of display; with MaxBug you simply get more of it. Figure 3.16 shows the information you receive after the execution of a single instruction.

```
00CCFE: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
PC=0000CCFE DR=0000A014
D0=00000000 D1=00000000 D2=00000000 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
A0=000022C0 A1=0000021F A2=0001437 A3=00070364
A4=000142AF A5=00070E42 A6=00070680 A7=00070D3C
```

**Figure 3.16** MidiBug and MaxBug Display

The debugger first prints the starting address of the instruction in main memory (in Figure 3.16, $00CCFE). It then disassembles and prints the instruction itself. It is important to remember that what is being disassembled is the object code that is stored in RAM. That means that the symbolic addresses that you used in your source code will not appear; instead you will see the absolute addresses that were substituted for the symbolic addresses during the assembly and linking process. All addresses and quantities are expressed in hexadecimal, regardless of the numbering system used in your source code. The stack pointer disassembles as A7, even though it may have been referred to as SP in the original program.
The remainder of the debugger's output displays the contents of the 68000's registers. **PC** refers to the program counter, **SR** to the status register, **D0–D7** to the eight data registers, and **A0–A7** to the eight address registers. All register contents are in hexadecimal.

Once a debugger is invoked, it will print its > prompt, display information about the current instruction, print another >, and wait for your command. Though there are many commands to control action of the debugger, two will be of the most use. **T** (for Trace) executes a single instruction. Traps (calls to ToolBox and operating system routines) are handled as if they were one instruction; the debugger will not trace the instructions that are part of the ToolBox or operating system routine.

**S** (for Step) when used alone, will also execute one instruction. Traps, though, are not treated as single instructions; the debugger will display each step in any ToolBox or operating system routines. You can also execute a series of instructions with Step by appending a quantity to the command that represents the number of commands to be executed. For example,

```
S 6
```

will execute six instructions, printing the debugging information about each one.

**MidiBug** replaces the very bottom of the screen with output for one instruction. The rest of the screen displays the output from the program being executed. As you execute successive instructions, the display for the previous instruction will scroll out of sight.

**MaxBug** replaces the entire screen with its own output and can therefore display information for up to five instructions at one time. If a program affects Macintosh's screen, then MaxBug will briefly show program output each time the screen changes and then return to the debugging display. The ' key (the key above and to the left of the TAB) will also toggle between the application's screen and the debugger's screen.

Using a debugger does present one problem. Since the debugger is monitoring the keyboard for your commands, it effectively prevents a program from getting input from either the keyboard or the mouse. If a program expects input to stop a loop, then when you run the program from within the debugger, you won't be able to stop the loop the same way you would if the program were running on its own. The situation can be somewhat distressing, since a disk drive may be spinning continually when you are using the debugger. (There is a process for stopping a drive while using a debugger; see the MDS manual.)

Ultimately, most loops stop by checking one of the flags in the status register. For example, the Sample program uses an instruction that checks the zero flag (bit 2). If the zero bit is set, the loop continues; if the bit is clear, the loop will end and the program stop. The solution, then, is to trick the program into thinking that it has required input by manually clearing the zero bit.
In Figure 3.16, the contents of the status register is $0000A014, which means that the zero bit is set. How in the world can you tell? Remember that each hexadecimal digit represents four binary digits. Therefore, the 4 in the right-most position actually represents $0100. The zero bit is bit 2 (the third bit from the right). What we need to do is replace the 4 with the hexadecimal representation of any of the following code groups: %0000, %0001, %0010, %0011, %1000, %1001, %1010, or %1011 (in hexadecimal: 0, 1, 2, 3, 8, 9, A, or B). The trick is that the third bit must be zero; the contents of the others is irrelevant.

The command:

**SR 0000A010**

will replace the contents of the status register with whatever follows **SR**. Give the debugger this command just before executing the instruction that tests the zero bit.

The debuggers allow you to change the contents of any register at any time.

**Dn new contents**

will replace the contents of data register **n**.

**An new contents**

will do the same for address register **n**.

To replace the contents of the program counter, use:

**PC new contents**

Be very careful when changing the program counter, since the instruction executed after a **Trace** or **Step** instruction will be whatever instruction begins at the address in the program counter.

To see the assembly language version of an application's instructions as they are stored in memory, use **ID** (instruction disassemble). Used alone, **ID** will disassemble the instruction at the current contents of the program counter. Follow **ID** with an address and it will disassemble the instruction at that address.

The debugger command **SM** (set memory) will change the contents of a memory location. It's general form is:

**SM main memory address new contents**

For example:

**SM 1A2B 33**

will place $33 in location $1A2B. Note that the debugger expects all addresses and quantities to be expressed in hexadecimal; no leading $ is necessary.
In some cases, you may wish to trace a few steps of a program and then let it run on its own again. The command **G**, for **GO**, will resume normal program execution, sending the debugger back into the background. It is therefore possible to place the trap that invokes the debugger at several places in a program. This will allow you to trace a few steps at whatever parts of the program are of interest.

If a program is so full of bugs that it cannot terminate successfully on its own, there are two ways to exit the debugger. The debugger command **ES** (exit to shell) will generally return to the Finder (note that some program errors will cause this command to fail and your only recourse is to reboot). **RB** (reboot) will reset the machine.

The successful use of a debugger is something that cannot be directly taught; it's something that comes from practice. To begin to understand what a debugger does, insert `__debugger` in the Sample program just below the `__initFonts` statement. Copy the appropriate debugger onto a disk that also contains a System Folder. For a single drive system, place the final version of the Sample program on this disk as well; in a two-drive system, the Sample program should be on a text disk in the external drive. Boot the system to install the debugger and then run the Sample program by double-clicking on its icon. The debugger screen will appear almost instantaneously.

Monitor the progress of the program using the **T** command. Keeping a printed listing of the program handy will also aid in understanding what appears on the screen. Look primarily at how each instruction changes the contents of the CPU's registers. Experiment with the other debugger commands. When you are finished, type **G** to return control to Sample so that it can terminate with a click of the mouse button or a key press.
Chapter Objectives

1. To create an I/O shell program that can be used to explore the 68000 instruction set
2. To understand the purpose and use of assembler directives
3. To understand data manipulation instructions (MOVE, LEA, PEA)
4. To understand instructions used to make comparisons in assembly language programs
5. To take a first look at creating a loop within an assembly language program, including instructions which execute unconditional branches

Creating an I/O Shell

Since all Macintosh I/O is done exclusively through the ToolBox, if you are going to see the result of executing even the simplest 68000 instruction, you'll need to be able to use the ToolBox right away. That would seem to mean that you must learn how to use the ToolBox at the same time you are learning the instruction set.

If, though, you modify the Sample program from Chapter 3 so that it appears as in Listing 4.1, you will have a ToolBox "shell" into which you can insert bits of 68000 code. The shell will display the results of executing those instructions. You can
then, for the most part, leave worrying about the ToolBox until you understand the instruction set. Therefore, many of the program listings in this chapter are designed to be inserted into the shell (as indicated in Listing 4.1) before they are run.

**Listing 4.1 Sample Macintosh Assembly Language Program**

```assembly
Include MacTraps.D ;Includes addresses of ToolBox routines
Include ToolEqu.D ;Includes the ToolBox equates
Include SysEqu.D ;Includes the System equates

PEA -4(A5)
_InitGraf ;Initializes QuickDraw
_InitWindows ;Initializes the Window Manager
_InitMenus ;Initializes the Menu Manager
_InitFonts ;Initializes the Font Manager

CLR.L -(SP) ;Clear space for WindowPtr result
PEA StoragePointer ;Window Storage pointer
PEA BoundsRect ;Exterior coordinates of window
PEA 'MAL Output Window' ;Title
ST -(SP) ;Make the window visible
MOVE #documentProc,-(SP) ;Make it a standard document window
MOVE.L #1,-(SP) ;Put the window in front
ST -(SP) ;Draw a go-away box
CLR.L -(SP) ;Place for window's reference value
_NewWindow ;Draw a standard document window

LEA WindowPtr,A0 ;load destination address for pointer
MOVE.L (SP)+,(A0) ;retrieve pointer

MOVE.L WindowPtr,-(SP) ;SelectWindow

MOVE.L WindowPtr,-(SP) ;put pointer back on the stack
_CallPort ;make this window the current grafport
_InitCursor ;set the cursor to the arrow

MOVE.W #7,-(SP) ;7 = athens
_TextFont ;Set the text font

MOVE.W #18,-(SP) ;18 for 18-point type
_TextSize ;Set the text size

MOVE.W #65,-(SP) ;Horizontal coordinate
MOVE.W #100,-(SP) ;Vertical coordinate
_MoveTo ;Move the pen
```

(continued)
Listing 4.1 (continued)

```
PEA  'HORAY!!! You did it!'       REMOVE THESE STATEMENTS
     _DrawString TO CREATE THE TOOLBOX

     SHELL

MOVE.L everyEvent,D0;Mask to select all events
     _FlushEvents            ;Clear the event queue

Event CLR -(SP)            ;Space for boolean result
     MOVE #%0000000000111110,-(SP)    ;Mask for keyboard and mouse
     PEA EventRecord                  ;Place to receive event info
     _GetNextEvent                   ;Get next event from queue

     MOVE (SP)+,D0                   ;Has a keyboard or mouse event occurred?
     CMP #0,D0                      ;If no event, branch to look again
     BEQ Event

     RTS                             ;Return to the Finder

WindowPtr  DC.L  ø
BoundsRect  DC.W 40,20,300,350
everyEvent  DC.L $0000FFFF
EventRecord  ;where GetNextEvent Puts its result
     What  DC  ø
     Message  DC.L  ø
     When  DC.L  ø
     Point  DC.L  ø
     Modify  DC  ø
```

Assembler Directives

Macintosh assembly language source file may contain more than just 68000
ions. It can also include assembler directives. Assembler directives are
mnemonics that give the assembler directions that are to be followed
the assembly process. Most of them involve setting aside space for storage.
an also assign values to symbolic addresses and cause external source files
cluded as a part of the file being assembled.
EQU (Equate)

One of the most useful assembler directives is EQU (equate). EQU assigns a permanent value to a symbolic address. For example:

```
Name EQU 0
```

assigns the value 0 to the symbolic address Name. Then, instead of using 0 in source code, use Name. When the program is assembled, the value 0 will be substituted for Name everywhere it appears.

An equate is directly equivalent to assigning a constant value to an identifier in the const block of a Pascal program. Like Pascal constants, the values assigned to symbolic addresses by EQU cannot be changed during program execution.

To handle equates and other symbolic addresses, the assembler builds a symbol table. Think of a symbol table as a two-dimensional array kept in RAM while the assembler is running. One column holds the symbolic addresses; there is therefore one row in the symbol table for each symbolic address. A second column in the table identifies the type of symbolic address (e.g., whether it is an equate or a statement label). The assembler enters a symbolic address into the symbol table when it is first encountered. For an equate, a third column in the array holds the value assigned to the symbolic address. For statement labels, the third column holds the address of the program statement to which the label refers.

Each time the assembler recognizes a reference to a symbolic address in the program being assembled, it checks the symbol table to see if it can find an entry for that symbolic address. If the symbolic address is an equate, then the assembler merely substitutes the value of the equate in the table for the symbolic address in the source code.

Because the assembler expects to find an entry for an equate in the symbol table, EQU statements must appear before their symbolic addresses are used in program instructions; otherwise, the program simply will not assemble. It is therefore good programming to group all EQU statements together (along with comments explaining what they reference) immediately after the INCLUDE directives (discussed directly below) at the very beginning of the program.

You can EQU addresses as well as constant numeric data. For example, if you include:

```
Address_1 EQU $1A3B
```

in source code, you can use Address_1 in any place where you need to reference the address $1A3B. It is acceptable in any of the Macintosh's addressing modes that accept absolute addressing.

What, then, is in those equates files (e.g., ToolEqu.D and SysEqu.D) that came with your Macintosh 68000 Development System? If you look at the source listings (ToolEqu.Txt and SysEqu.Txt) you'll see that both files are nothing more than a series of EQU statements. They set up constants that are useful when working with ToolBox and operating system routines.
INCLUDE

To make the symbolic addresses available in the equates file to any program you write, INCLUDE the equates files. INCLUDE is another assembler directive. It instructs the assembler to seek another source file which is to be inserted into a program. To use INCLUDE, specify:

```
INCLUDE  fname
```

where fname is the name of the source file to be included in the program being assembled.

Data Allocation

There are two assembler directives that fall into the classification “data allocation directives.” These set up symbolic addresses for storage locations in either the program itself or the applications globals area of RAM. You can think of them as analogous to variable names (i.e., the symbolic address represents the location of one or more pieces of data). The contents of storage locations identified by such symbolic addresses can be changed while the program is running.

DC (define constant) assigns one or more values to a symbolic address. The statement:

```
Label  DC  0
```

will, for example, cause the following actions during assembly:

1. An address for Label will be selected at the end of the source code. (If you look at the bottom of the assembler listing for the Sample program, you will see the space that has been allocated for each DC directive.
2. Label will be associated with that address.
3. The address associated with Label will be given an initial value of 0.

There are four variations on the define constant directive: DC, DC.B, DC.W, DC.L. The extensions determine whether the data will be aligned on byte, word, or longword boundaries. If no extension is present, the data will be aligned on word boundaries by default.

At first glance, it might seem that DC isn't much different from EQU. Remember, though, that EQU assigns a permanent value to a symbolic address, whereas values assigned by DC are only initial values and can be changed by the instructions within a program.

The fact that DC allows changing the value associated with a symbolic address does not mean that you should necessarily do so. It is good practice to use DC only
to store constants and not as locations for data that will change (i.e., consider a location established by **DC** as if it were in ROM, useful for read-only operations). The major exception to this rule occurs when an application does printing (see Chapter 10 for details).

**DC** is also used to assign a series of storage locations, each with its own unique value, to a single symbolic address. The statement:

\[
\text{Label DC 0,16,'A Sample Window'}
\]

reserves enough storage to store the values 0, 16, and the string "A Sample Window." Use of the symbolic address **Label** will reference the two numeric values and the string. This capability is important when preparing data for use with ToolBox routines.

**DCB** (define constant block) sets aside a block of memory locations, all of which will be initialized to the same value. (Notice that this is not the same as using **DC** to reference a series of values, since the **DC** values can be different from one another.) To use **DCB**, you must not only specify the initial value for the storage locations, but the length of the block of locations to be reserved. For example:

\[
\text{Label DCB 12,0}
\]

will reserve twelve words of storage, beginning at the symbolic address **Label**. Each location will be given the initial value 0.

The general form of the **DCB** assembler directive is:

\[
\text{Symbolic address DCB length of block, initial value}
\]

The actual number of bytes reserved depends on the extension applied to the **DCB** directive. If there is no extension, or if you use an extension of **.W**, the "length of block" parameter will refer to the number of words to be set aside. An extension of **.B** indicates that the length is expressed in bytes; **.L** specifies a length in number of longwords.

**DS** (define storage) also reserves a block of storage locations. This storage does not become a part of an assembled program. Rather, it is allocated in the applications globals area at run time. This form of storage allocation should be used for all read/write operations (i.e., a program should avoid writing into its own code, as it would if you wrote to a **DC** location).

The applications globals area begins at **$-100(AS)** and grows **down** in memory. All storage locations allocated by **DS** must therefore always be referenced relative to A5 with what looks like Address Register Indirect with offset addressing. Since A5 always contains the starting location of the applications globals area, its contents should never be changed during program execution.

The general form of the statement is:

\[
\text{Symbolic address DS length of block}
\]
Therefore, the statement:

```
Label  DS  12
```

will set aside twelve words of storage. No initial value is given to the storage.

As with the `DCB` directive, how the length parameter is interpreted depends on the extension affixed to the mnemonic. No extension or an extension of `.W` refers to words, `.B` to bytes, and `.L` to longwords.

Access to the storage set aside by `Label` above appears as:

```
Label(A5)
```

The amount of space needed for `DS` locations appears on the Linker screen during the linking process beside the label “Data Size.”

### End of Source

Another essential assembler directive is `END`. `END` is the last statement in a source code file. Any statements after `END` will be ignored by the assembler. It is important to remember that `END` is the physical end of the source code. It has nothing to do with the logical end of a program.

### Printing Control Directives

If you are using the Executive, you cannot control listing options from the `OPTIONS` menus in the Assembler and Linker. You can, though, specify the same choices in your source code.

- `.EJECT` will cause the printer to start a new page. This directive will take effect when creating a hard copy of either an assembler or linker listing.
- `.Verbose`, `.ListToFile`, and `.ListToDisp` have the same effect as selecting those commands from the `OPTIONS` menus (see Chapter 3). To turn off verbose assembly or a listing use `.NoVerbose` or `.NoList` respectively.

### Data Manipulation Instructions

An important part of any microprocessor's instruction set is concerned with moving data around in memory. Arithmetic instructions require that at least one operand be located in a data or address register. Even more importantly, the Macintosh’s ToolBox routines look for parameters which have been placed on the
stack; operating system routines expect their parameters to be pointed to by addresses in registers.

The most frequently used 68000 data manipulation instructions used in Macintosh assembly language programs are **MOVE, PEA, LEA**.

### MOVE

The **MOVE** instruction takes a piece of data and shifts it from one location to another. Like an assignment statement in a high-level language (e.g., \( C = A \)), the data in the source location is *copied* into the destination location; the contents of the source location are not altered.

The format of the **MOVE** instruction is:

```
MOVE source address, destination address
```

For example:

```
MOVE #12,D1
```

will put the decimal quantity 12 into data register D1. (Remember that when \# precedes a number it will be interpreted as a quantity rather than as an address.)

The size of the operand transferred by a **MOVE** statement depends on the extension given the instruction. **MOVE** or **MOVE.W** will move one word of data. **WORD.B** will move a byte and **MOVE.L** will move a longword.

Source and destination addresses can be specified using most of the 68000's addressing modes. The examples which follow will show you the ones most commonly used.

In order to see the results of **MOVE** statements, let's use a ToolBox routine to display a single character on the screen. This routine is called **DrawChar** and it expects to find the ASCII code for the character to be printed on the top of the stack. Therefore, the step that immediately precedes the call to **DrawChar** must **MOVE** a character onto the stack.

All ToolBox routines are called by their names. To let the assembler know that the statement is a call, an underbar (__) is put in front of the routine name. Therefore, if you put the line:

```
__DrawChar
```

into your source code, it will execute the **DrawChar** routine. More detail on how such calls work appears beginning in Chapter 6.

The ASCII code for a character is placed on top of the stack using Address Register Indirect with Predecrement addressing. (In fact, putting things on the stack is a very common use of this addressing mode.) For example:

```
MOVE source address, -(SP)
```
This statement will cause the Mac to first decrement the contents of the stack pointer (SP or A7). The data pointed to by the source address will then be moved to the new address contained in the stack pointer.

Why is the address decremented rather than incremented? Remember that the stack starts high in memory and grows down (i.e., the bottom of the stack has a high address; the top of the stack will always have an address lower than the bottom). Therefore, each time we put something on the stack, the address of the top must first be decreased.

Insert these statements into the ToolBox shell:

```
MOVE #$0040, -(SP)
_DrawChar
```

When you assemble, link, and run the program, the character "@" will print on the screen (see Figure 4.1). $0040 is the ASCII code for "@". Because $0040 is preceded by #, the quantity $0040 is moved to the stack. This is an example of using immediate data as the source address in a MOVE statement. (Note: The DrawChar routine expects to find an entire word of data on the stack. Though ASCII codes occupy only a single byte, you must nevertheless move a word onto the stack with the ASCII code in the low-order byte. Thus we move $0040 onto the stack, forcing the ASCII code into bits 0–7.)

![Figure 4.1 Output From a Single Call to DrawChar](image-url)
It is also possible to use **MOVE** to take things off the stack. This is important because many of the ToolBox routines return information needed later in a program. That information is placed on the top of the stack. If you use the instruction:

```
MOVE (SP) + , D1
```

the contents of the RAM location pointed to by the contents of the stack pointer will be moved to data register D1. Then the stack pointer will be incremented.

As we have previously discussed, when an operand or address is placed on the stack, the contents of the stack pointer must be decremented. Similarly, when something is taken off the stack and effectively "removed" from the stack, the stack pointer must be incremented. Therefore, the example above uses Address Register Indirect with Postincrement addressing. The contents of the stack pointer are incremented *after* the instruction is executed. This is probably the most common situation in which this particular addressing mode is used.

Other addressing modes are also commonly used with the **MOVE** statement. Remove the two statements you previously placed in the ToolBox shell and insert the following:

```
MOVE #$0040, D1
MOVE D1, – (SP)
_DrawChar
```

Running the program should still print that "@." (If you're getting tired of "@," substitute the hexadecimal equivalent of any other ASCII code Macintosh uses.)

The first **MOVE** uses Data Register Direct addressing to specify the destination address. The $0040 will be stored in data register D1.

The second **MOVE** uses the same addressing mode to specify the source address. The contents of data register D1 are moved onto the top of the stack (after, of course, the contents of the stack pointer [A7] are decremented).

You can also move data stored under symbolic addresses. For example, try this in the shell:

```
Data EQU $0040
MOVE #Data, – (SP)
_DrawChar
```

The **EQU** permanently associates the symbolic address **Data** with the value $0040. Using the symbolic address in the **MOVE** statement has the same effect as using $0040 as immediate data. Notice that just like the number $0040, the symbolic address was preceded by a # so that the assembler realized that the quantity stored as **Data** was to be used as immediate data rather than as an address.
Symbolic addresses assigned values by EQU can be used anywhere you would use data. For example:

```
Data EQU $0040
MOVE #Data,D1
MOVE D1, -(SP)
__DrawChar
```

will put $0040 into data register D1 and then move it onto the stack. If you put the above code into the ToolBox shell, you should still see "@" printed in the output window.

MOVE can also be used to transfer data between registers. The third line of the following code will move the contents of data register D1 to data register D2.

```
Data EQU $0040
MOVE #Data,D1
MOVE D1,D2
MOVE D2, -(SP)
__DrawChar
```

The source address in a MOVE statement can be specified using any of the 68000's addressing modes. The destination address, however, cannot be specified with immediate addressing nor can it use either of the program counter addressing modes.

The reason immediate addressing cannot be used should be obvious. The destination must be a location, a place to put something. It's simply not possible to store something in a piece of data.

Why the program counter modes can't be used may not be so clear. But consider this: if you store a piece of data in the program counter, you will destroy the previous contents of the program counter. Since the program counter keeps track of which instruction is to be executed next, erasing that address will completely disrupt program execution.

The MOVE instruction, like most other instructions, affects the flags in the status register. The extend bit is unaffected. The carry and overflow bits always get a value of 0. (We say that they are cleared.)

What happens to the negative and zero bits depends on the value being moved. If the value is equal to zero, the zero flag will be set (given a value of 1) and the negative bit will be cleared. If the value is negative, the negative bit will be set and the zero bit cleared. If the value is positive, both bits will be cleared.

**PEA**

The letters PEA stand for Push Effective Address. This instruction is not commonly used in many 68000 machines, but because it pushes addresses onto
the stack and then automatically decrements the stack pointer, it is extremely useful for setting up parameters for ToolBox routines.

Take a look at the two statements you removed from the Sample program to create the shell:

```
PEA 'HOORAY!!! You did it'
      DrawString
```

This use of the ToolBox routine `DrawString` displays the string that you see as the operand for the `PEA` instruction. Like `DrawChar`, `DrawString` looks for its operand on the stack. The string itself, though, is not placed on the stack; during assembly and linking it is placed at the end of the program code. Therefore, when you want to display a string, push a `pointer` to the start of the string.

What's a pointer? A pointer is an address that corresponds to the starting address of a series of storage locations. Usually, a pointer will be the starting address of a string or a data structure in main memory.

The general form of the instruction is:

```
PEA effective address of source data
```

`PEA` can use any addressing mode except immediate, simply because immediate data isn't an address. This instruction does not affect the codes in the status register.

**LEA**

LEA stands for Load Effective Address. It moves an address into an address register. The general form of the instruction is:

```
LEA source address, destination address register
```

`LEA` is most useful for retrieving the absolute address assigned to a symbolic address. To see how it works, let's look at the data structure used to pass parameters to the operating system routines that provide access to disk files.

```
paramBlock
    Link DC.L 0
    Type DC 5
    Trap DC 0
    CmdAddr DC.L 0
    Complete DC.L 0
    Result DC 0
    NamePtr DC.L 0
    VRefNum DC 2
```
These eight parameters are common to all file manager routines. (The complete parameter block contains 8 to 16 additional fields, depending on the specific routine.) The first four fields are used by the File Manager. The other four, though, are of concern to the programmer.

For example, NamePtr must contain the address of the location where the name of a file is stored. The pointer must be loaded into NamePtr before calling the File Manager routine. Assume that the file name is stored under a symbolic address:

\texttt{Fname DC 'SampleFile.Text'}

The instruction:

\texttt{LEA Fname,A1}

will store the starting address of the string \texttt{SampleFile.Text} in A1. This is an example of absolute addressing, since \texttt{Fname} represents a specific RAM location.

To put that address into NamePtr, the address of NamePtr must also be available in an address register:

\texttt{LEA NamePtr,A2}

Then, a program can execute:

\texttt{MOVE.L A1,(A2)}

This statement takes whatever is stored in A1 (the address of \texttt{Fname}) and stores it at the address stored in A2 (the address of \texttt{NamePtr}).

There are some things that are important to remember about \texttt{LEA}. The destination of the instruction is always an address register. The mnemonic does not take any extensions; \texttt{LEA} always transfers a full longword (even though the addresses are only 24 bits).

The source address can be either in an address register, the program counter, or can be an absolute address. Three address register indirect addressing modes are acceptable: Address Register Indirect, Address Register Indirect with Displacement, and Address Register Indirect with Displacement and Index. Both of the program counter modes can also be used for the source address.

\texttt{LEA} does not affect any of the flags in the status register.

---

**LOOPING**

Executing a series of statements repeatedly is rather easy in a high-level language. In Pascal for example, you can use \texttt{WHILE/DO,REPEAT/UNTIL}, and
FOR to implement iteration. With 68000 assembly language, though, there are no built-in looping instructions. To understand the sequence of instructions necessary to create a loop, consider the steps required to repeat a set of instructions a fixed number of times.

1. Initialize the counter to 1.
2. Compare the counter with the quantity that represents the number of times the loop is to be executed.
3. If the counter equals the ending value, then terminate the loop.
4. Otherwise, execute the instructions that form the body of the loop.
5. Increment the counter.
6. Return to step 2.

To program a loop in assembly language, you must execute each step above. To see a loop in action, insert the following instructions into your I/O shell:

```
MOVE    #1,D1   ;counter
MOVE    #5,D2   ;number of times to execute loop
Again   CMP     D1,D2   ;check the counter
        BMI     Done   ;end the loop
        MOVE    #$0040, - (SP)  
        _DrawChar
        Add     #1,D1   ;increment the counter
        BRA     Again   ;continue the loop
```

In order to get this code to work (it should print a series of six "@"s as in Figure 4.2), place the symbolic address **Done** in the label field of the statement:

```
MOVE.L  everyEvent,D0
```

so that the statement appears as:

```
Done    MOVE.L  everyEvent,D0
```

This sequence introduces four new instructions: **CMP** (used to make decisions), **BMI** (one way to check the flags in the status register), **ADD** (integer addition), and **BRA** (one way to do an unconditional branch). Once you are familiar with these instructions and their variations you will, believe it or not, know most of the instructions used in Macintosh assembly language programs.
Making Comparisons

The 68000 instruction set has one generalized instruction for making comparisons — **CMP**. (There are others, but they are more specialized and less commonly used.) The general form of the instruction is:

```
CMP address of source operand, destination data register
```

**CMP** subtracts the source operand from the quantity in the destination data register. The result of the subtraction isn't stored anywhere. The instruction does, though, set the codes in the status register according to that result.

For example, consider this series of instructions:

```
MOVE   #6,D1
MOVE   #10,D2
CMP    D1,D2
```

The **CMP** instruction will perform the subtraction “10 – 6.” The result (4) is not stored anywhere. The negative bit in the status register is cleared (the result was positive). The zero bit is also cleared (the result was non-zero). Since no overflow occurred and no borrow was required, both the overflow and carry bits are cleared. **CMP** does not affect the extend bit.
Now, look at these instructions:

MOVE  #12,D1  
MOVE  #10,D2  
CMP    D1,D2

The result of the subtraction is – 2. Therefore, the negative and carry bits will be set and the others cleared.

After executing:

MOVE  #5,D1  
MOVE  #5,D2  
CMP    D1,D2

only the zero bit will be set; all the others will be cleared.

CMP will work with characters as well as quantities. If you think about ASCII codes for a moment, you'll notice that letters that come alphabetically first have numerically lower codes than those that come later (e.g., A = $41, B = $42, C = $43, etc.). Therefore, when CMP performs a subtraction using ASCII codes, a program is actually testing for alphabetical order.

For example:

MOVE  #$0043,D1  
MOVE  #$0046,D2  
CMP    D1,D2

tests whether C comes before F in an alphabetical sequence. Remember that lower-case letters have different codes from upper-case letters so, for example, h will be greater than H.

You can specify the source operand using any addressing mode.

Testing the Condition Codes

CMP is conceptually only part of an IF/THEN statement. It compares the operands in question and sets the status register so you can actually test the condition. Testing the condition requires a separate instruction.

In Pascal, any executable statement, including a compound statement, can follow THEN for execution if the condition is true. In assembly language, you are much more limited. Though you can test for a variety of relationships between the quantities being compared (e.g., equal to, not equal to, greater than, less than,
plus, etc.), there are only two possible actions: you can branch to another instruction (the branch will take place if the condition is true; otherwise program execution continues with the next statement); or you can set or clear a destination byte (the byte will be set if the tested condition is true, cleared if the condition is false).

Using Pascal, you would write:

**IF condition is true THEN GOTO symbolic address**

or

**IF condition is true THEN destination byte = $FF**
**ELSE destination byte = $00;**

Regardless of whether you decide to branch or set a byte, you will still be testing the condition codes that were set during a previous operation.

**Bcc**

Bcc stands for Branch on Condition Code. It is a *conditional* branch, since the branch occurs only if the condition being tested is true. The cc is replaced by two letters which stand for the specific condition you want to test. The most commonly used forms are:

- **BEQ** Branch if Equal (true if the zero bit is set)
- **BNE** Branch if Not Equal (true if the zero bit is clear)
- **BMI** Branch if Minus (true if the negative bit is set)
- **BPL** Branch if Plus (true if the negative bit is clear)

The following conditions, also often used, are tested using logical combinations of the bits in the status register:

- **BGE** Branch if Greater Than or Equal To
- **BGT** Branch if Greater Than
- **BLE** Branch if Less Than or Equal To
- **BLT** Branch if Less Than

The full set of condition codes can be found in the 68000 Programmer's Reference Manual that came with your Macintosh 68000 Development System.

To use a **Bcc**, code a statement like:

**Bcc  symbolic address of destination**

How does a branch work? During assembly, the assembler computes the number of bytes between the **Bcc** instruction and the destination statement. This quantity, know as an offset, becomes a part of the instruction in the object code.
When the statement is executed, the appropriate condition codes are tested. If the condition is true, the offset is added to the contents of the program counter. The program continues at the program counter’s new contents. The offset is limited to the quantity that will fit in one word.

To explore how Bcc works, insert the following code into your ToolBox shell:

```
MOVE #0,D1
MOVE #5,D2
CMP D1,D2
BMI LessThan
PEA "The source operand is smaller than the destination operand"
_DrawString
JMP Ending
LessThan PEA "The source operand is larger than the destination operand"
_DrawString
Ending PEA "!!!"
_DrawString
```

Vary the mnemonic in the fourth line (BMI) to see how the different conditions work. You can also put different values in D1 and D2. Try, for example, using equal quantities.

**Unconditional Branching without Tests**

There is one instruction in the above example that we haven’t discussed — **JMP**. **JMP** is one of two instructions that does an unconditional branch. ("Jump directly to a symbolic address; Do not pass Go, Do not collect $200...") The general form is:

```
JMP symbolic address of destination
```

During assembly, the symbolic address of the destination is replaced by the actual address of the destination instruction.

The other instruction that causes an unconditional branch is **BRA**. Like **Bcc**, during assembly the assembler computes the number of bytes between the **BRA** instruction and the destination instruction and turns that quantity into an offset. The offset is limited to the quantity that will fit one word.

The general form is:

```
BRA symbolic address of destination
```
When a program executes an unconditional branch, the contents of the program counter are changed. If branch is initiated by a JMP instruction, the address in the operand field will replace whatever was in the program counter. With BRA, the offset in the operand is added to the current contents of the program counter. In either case, program execution continues at the new address indicated by the modified program counter.

In most cases, choosing whether to use JMP or BRA is a toss up. If, though, you have a very long program and are concerned about space, BRA can save at least one word of space over JMP. In cases where the offset will fit within a byte, it is assembled in the same word as the BRA instruction. If the offset is more than 255, it will be assembled into the word after the instruction. A JMP always occupies two or three bytes, one for the instruction and one or two for the address, depending on its size. Therefore, if an unconditional branch spans less than 255 bytes, you will save one or two words of space in your object code every time you use BRA rather than JMP. On the other hand, if you want to shift program control more than 32,767 bytes (the maximum offset), you must use the JMP.

More on Testing Condition Codes (Scc)

The second option for action after testing a condition code—setting or clearing a byte—is specified by the various forms of Scc (Set on Condition Code). The cc is replaced by two characters representing the condition to be tested. A Scc statement is written:

```
Scc   address of byte to be set or cleared
```

The destination byte can be specified by any addressing mode except: 1) Address Register Direct; 2) Program Counter with Displacement; 3) Program Counter with Index; 4) Immediate; and 5) Quick Immediate.

For example, consider these instructions:

```
MOVE   #2,D1
MOVE   #6,D2
CMP    D1,D2
SEQ    D3
```

The SEQ (Set if Equal) instruction checks the zero bit in the status register. In this case, the zero bit has been cleared because the result of the comparison was non-zero. Therefore, the SEQ instruction will clear D3 (fill it will all zeros).

If, though, you execute:

```
MOVE   #2,D1
MOVE   #6,D2
CMP    D1,D2
SNE    D3
```
D3 will be set (filled with all 1's [$FF]). **SNE** (Set if Not Equal) is true if the zero bit in the status register has been cleared.

As with **Bcc**, you can test a wide variety of conditions:

- **SGE** Set if Greater Than or Equal To
- **SGT** Set if Greater Than
- **SLE** Set if Less Than or Equal To
- **SLT** Set if Less Than
- **SMI** Set if Minus
- **SPL** Set if Plus

(Other, less frequently used codes are in the *Programmer's Reference Manual*).

It is also possible to set or clear a byte without testing the condition codes.

**ST** destination address

will fill the byte specified by the destination address with all 1's. By the same token,

**SF** destination address

clears the byte at the destination address.

To clear either a word or longword, use the **CLR** instruction:

**CLR** destination address

with no extension or a .W extension will clear two bytes beginning at the destination address; an extension of .L will clear four bytes. **CLR.B** will clear one byte. The address to be cleared can be specified using any addressing mode except: 1) Address Register Direct; 2) Program Counter with Displacement; 3) Program Counter with Index; 4) Immediate; and 5) Quick Immediate.

There is no instruction to simply set all the bits in a word or longword.

---

**Questions and Problems**

For problems 1 through 5, assume the contents of the following selected 68000 registers and memory locations (the latter are identified by symbolic and absolute addresses):

- **D0** .. [0000AB12]
- **D1** .. [00000002]
- **D2** .. [FF00FFAA]
- **D3** .. [FF000000]
- **A0** .. [00000001]
- **A1** .. [00000002]
100 MACINTOSH ASSEMBLY LANGUAGE: AN INTRODUCTION

$0001 .. [003A]  $0002 .. [0001]  $0003 .. [0010]
LOC1       LOC2       LOC3

$0004 .. [0002]  $0005 .. [0020]  $0006 .. [0011]
LOC4       LOC5       LOC6

1. What will be stored in register D3 after each of the instructions below are executed?

   a. MOVE.B D0,D3
   b. MOVE D0,D3
   c. MOVE.L D0,D3
   d. MOVE.B D2,D3
   e. MOVE D2,D3
   f. MOVE.L D2,D3

2. A. What will be stored in the destination register after each of the instructions below are executed?

   B. Identify the addressing mode used in each case.

   a. MOVE LOC1,D0
   b. LEA LOC1,A0
   c. MOVE.L A0,D0
   d. MOVE (A0),D0
   e. MOVE 4(A0,D1),D0
   f. MOVE 2(A0),D0
   g. CMP D1,D0
   h. ST D0
   i. CLR.B D0
   j. SF D0
   k. CLR D2
   l. CLR.L D2
   m. MOVE #'C',D0
   n. MOVE #'AB',D0
   o. MOVE.B #'AB',D0
   p. MOVE.L #'ABCD',D0
   q. MOVE.L #'ABC',D0
   r. MOVE #'ABC',D0

3. Identify the contents of each register and memory location that changes when the following blocks of code are executed:

   a. MOVE LOC2,D0
      MOVE #0006,A0
      MOVE D0,A0

   b. LEA LOC2,A0
      MOVE #0006,(A0)

   c. MOVE #0004,A0
      MOVE (A0)+,D0

   d. Offset EQU 3
      LEA LOC2,A0
      MOVE Offset(A0),D0
4. Problems (f) and (g) in 3 above are essentially the same; their difference lies only in the quantities being compared. Looking at both blocks of code, what do they do?

5. For each block of code below, indicate the state of the flags in the status register after the code has been executed.

<table>
<thead>
<tr>
<th>a.</th>
<th>MOV LOC1,D0</th>
<th>c.</th>
<th>MOV LOC3,D1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>LOC2,D0</td>
<td>CMP</td>
<td>#10,D1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b.</th>
<th>MOV LOC6,D0</th>
<th>d.</th>
<th>LEA LOC5,A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>LOC1,D0</td>
<td>MOV LOC4,D0</td>
<td>CMP (A0),D0</td>
</tr>
</tbody>
</table>
6. Write a block of code that will load an operand into a data register from a main memory location called Spot and then push that same operand onto the stack:
   a. Write the code assuming that Spot has been defined as:
      ```assembly
      Spot DC 0
      ```
   b. Write the code assuming that Spot has been defined as:
      ```assembly
      Spot DS 1
      ```

7. Write a block of code that pulls a longword from the stack and stores it in a main memory location called NextPlace:
   a. Write the code assuming that NextPlace has been defined as:
      ```assembly
      NextPlace DC.L 0
      ```
   b. Write the code assuming that NextPlace has been defined as:
      ```assembly
      NextPlace DS.L 1
      ```

8. Write a block of code that:
   a. loads an operand from Place1 into a data register.
   b. loads a second operand from Place2 into another data register.
   c. compares the operands.
   d. If the operands are equal, branches to a statement labeled Done.
      i. if Place 1 > Place 2, writes the first operand in Largest and the second in Smallest
      ii. if Place 2 <= Place 2, writes the first operand in Smallest and the second in Largest

All operands are word-sized. Be sure to set aside storage space for Place1, Place2, Smallest, and Largest with the DC or DS directive. (Remember: storage locations defined by DS must be referenced relative to register A5.) Use additional statement labels as necessary.

9. Write two versions of a block of code that creates a string — 'Some silly text' — and pushes its addresses onto the stack:
   a. Allocate the string as a literal within the code itself.
   b. Allocate the string with a DC directive.

10. Write a block of code that loads an operand from main memory into a data register. If the operand is positive, set D7; if it is negative, clear D7. Allocate any necessary storage locations. Something to think about: is a CMP instruction required as part of your code?
Chapter Objectives

1. To understand the two's complement system of integer representation
2. To review the process of constructing an assembly language loop
3. To learn the integer arithmetic instructions
4. To learn the logical instructions
5. To understand the use of assembly language subroutines

Integer Arithmetic

Microprocessor instruction sets contain instructions for doing arithmetic with integers. To manipulate them, you need to know how integers are stored.

Two's Complement

The Macintosh stores integers using a two's complement system. Complementing is easier to understand if we look first at base 10 (decimal). The complement of a base 10 number is the quantity which, when added to the original number, produces a sum of 10. For example, the 10's complement of 6 is 4; the 10's...
complement of 4 is 6. By the same token, the two's complement of a binary number is the quantity which, when added to that number, will produce a result of 2.

There is a simple procedure for obtaining a binary number's two's complement:

1. Invert the bits in a number (for every 0 write a 1, for every 1 write a 0)
2. Add 1 to the least significant bit

   For example, to convert %100011 to its two's complement:

   1. Invert: %100011 becomes %011100
   2. Add 1: %011100 + 1 becomes %011101

To convert a number back to its true-magnitude binary form, take the two's complement of the two's complement.

The Macintosh has two sizes of integer — 16 bits and 32 bits (integer and longinteger). In each case, the high-order bit is used as a sign bit — bit 15 of an integer and bit 31 of a longinteger. (Remember that the bits are numbered beginning with 0.) If the high-order bit is clear, the number is positive; if it is set, the number is negative. That means that the high-order bit does not participate in the magnitude of the number. An integer therefore has only 15 bits available for the number itself, producing a range of −32,768 to +32,767. A longinteger has 31 bits available for the number, giving it a range of −2,147,483,648 to +2,147,483,647.

Negative numbers are stored in their two's complement form. Positive numbers are stored in their true magnitude form (i.e., they are not translated to two's complement). This means that if you look at positive numbers in registers or in main memory, they can be directly converted to decimal. Negative numbers, on the other hand, must first be converted back to their true magnitude form before you can determine their value.

As an example, consider the number −5. In binary, 5 is %0101. If the number were positive, it would be stored in a word-sized location as %0000 0000 0000 0101 ($0005). To store a −5, however, two things must happen: the binary must be converted to two's complement form, and a 1 must be placed in bit 15 as the sign bit:

1. Convert to two's complement
   a. Invert the digits (%000 0000 0000 0101) to get %111 1111 1111 1010. Note that we are working with only 15 bits; the 16th will be added later to serve as a sign bit.
   b. Add 1 to get %111 1111 1111 1011. This is the two's complement form.
2. Insert a sign bit to get the final number %1111 1111 1111 1011
In hexadecimal, \(-5\) appears as $\text{FFFFFB}$. That is the quantity you will see displayed by the debugger if you examine a register or memory location that contains \(-5\). In hex, \(-1\) is $\text{FFFF}$, \(-2\) is $\text{FFFE}$, \(-3\) is $\text{FFFD}$, \(-4\) is $\text{FFFC}$, and so on.

### The Integer Arithmetic Instructions

Before going on, let's recapitulate the code that creates a loop like the one first introduced in Chapter 4:

```
TopOfLoop   MOVE   #1,D1   Step 1
           MOVE   #TargetValue,D2 2
           CMP    D1,D2 3
           BMI    Outside Loop 4
           {body of the loop goes here} 5
           ADD    #1,D1 6
           BRA    TopOfLoop 7

Outside Loop  . . . . .
```

The steps in this loop are:

1. Initialize a counter
2. Set a location equal to the target value
3. Compare the counter to the target value
4. If the counter equals the target value, end the loop
5. Execute the body of the loop (any executable statements go here)
6. Increment the counter
7. Transfer control to the top of the loop

The instruction in statement 6 is an example of integer arithmetic. Integer arithmetic adds and subtracts numbers up to 32 bits in length. The highest-order bit (regardless of the size of the operands) is maintained as a sign bit ($0 = \text{a positive number, } 1 = \text{a negative number}$).

Integer arithmetic also multiplies and divides signed or unsigned whole numbers up to 16 bits in length: the result can fill up to 32 bits. If you indicate that you want to do a signed operation, the highest-order bit in each operand will be used as a sign bit. Otherwise, all bits participate in the magnitude of the number.

To do arithmetic with larger or smaller numbers or numbers that have a fractional portion, you must use the Macintosh floating point arithmetic package (see Chapter 12).
ADD

The **ADD** instruction adds a source operand and a destination operand and stores the result in the destination location. One of the two operands must be a data register; therefore, the instruction can take two forms:

ADD  effective address of source operand, destination data register

or

ADD  source data register, effective address of destination operand

The statement:

ADD  #16,D1

will add the quantity 16 to the contents of D1 and store the result in D1. The second operand, which had previously been in D1, will be lost when the result is stored. The **ADD** statement has the same effect as the Pascal statement:

D1 := D1 + 16

In this form of **ADD**, you may use any addressing mode to specify the source operand.

When you specify the destination operand as a RAM address (the source operand will be in a data register), you may only use "alterable" addressing modes: all the address register direct modes and absolute addressing. For example, assume the symbolic address **Label1** has been assigned to a location in the applications globals area. Then:

MOVE  #22,D1
ADD    D1,Label1

will 1) put the quantity 22 into D1, 2) add the quantity in D1 to the quantity in the RAM location associated with the symbolic address **Label1**, and 3) store the result in **Label1**. (The Pascal equivalent is **Label1 := Label1 + D1**.)

If you want to use an address register as the destination for the result of an addition, you need to use a variation of the **ADD** instruction: **ADDA**. The instruction is written as:

ADDA  effective address of source operand, destination address register

For example:

ADDA  #22,A1
will add the quantity 22 to the contents of A1 and store the result back in A1. (In
Pascal, \( A1 := A1 + 22 \).) You may use any addressing mode to specify the source
operand.

It is also possible to add immediate data to an operand stored in RAM without
moving the operand into a data register. ADDI (Add Immediate) and ADDQ (Add
Quick) will both do the job. ADDI has the form:

\[ \text{ADDI} \ #\text{quantity, effective address of destination operand} \]

An ADDQ instruction is written exactly like an ADDI, but the immediate data is
restricted to the range 1 to 8. The quantity is assembled as a part of the instruction
and therefore can save space in your source code.

Any variation of the ADD instruction can be specified as operating on a byte
(ADD.B), a word (ADD or ADD.W), or a longword (ADD.L).

This family of instructions affects all the condition codes. As you might expect,
the negative bit will be set if the result is negative, cleared if positive. The zero bit will
be set if the result is zero, cleared if non-zero. The carry and extend bits are both set
if a carry occurs and cleared if there has been no carry. An overflow will set the
overflow bit; otherwise, it will be cleared.

\section*{SUB}

SUB (Subtract) is exactly analogous to ADD. The instruction subtracts the quantity
in a source location from the quantity in a destination location and stores the result
in the destination location. As with ADD, there are two forms:

\[ \text{SUB} \ \text{effective address of source operand, destination data register} \]

or

\[ \text{SUB} \ \text{source data register, effective address of destination operand} \]

The instruction:

\[ \text{SUB} \ #12,D1 \]

has the same effect as the Pascal statement:

\[ D1 := D1 - 12 \]

The instructions:

\[ \text{MOVE} \ #12,D1 \]
\[ \text{SUB} \ D1,\text{Label1} \]
perform the same actions as:

\[ \text{Label1} = \text{Label1} - D1 \]

(assuming that \text{Label1} has been previously assigned as a symbolic address).

\text{SUB} has the same restrictions on addressing modes as \text{ADD}. If the source operand is specified by its effective address (as opposed to being in a data register), then you may use any addressing mode. But if the destination address is identified by an effective address, only the register indirect and absolute modes are acceptable.

\text{SUB}, as does \text{ADD}, has three variations:

1. \text{SUBA} — the destination address is an address register, as in:

   \text{SUB} \ #333,A1

2. \text{SUBI} — the source address is an immediate quantity and the destination is identified by its effective address; i.e.:

   \text{SUB} \ #54,\text{Label3}

   where \text{Label3} has been previously defined as a symbolic address.

3. \text{SUBQ} — the source address is an immediate quantity in the range 1 through 8 and the destination location is specified with any addressing mode but the program counter modes. For example:

   \text{SUBQ} \ #2,(A2)

   With each variation you may specify the size of the operands as byte (e.g., \text{SUB.B}), word (e.g., \text{SUB} or \text{SUB.W}), or longword (e.g., \text{SUB.L}).

   \text{SUB} also affects the condition codes in the same way as \text{ADD}: the negative bit is set if the result is negative (cleared if positive); the zero bit is set if the result is zero (cleared if non-zero); carry and extend bits are set if a borrow occurred (cleared if no borrow occurred); and the overflow bit is set if an overflow occurred (cleared if no overflow).

\textbf{Integer Multiplication}

Integer multiplication comes in two forms — \text{MULS} and \text{MULU}. \text{MULS} (for Signed Multiply) computes the product of two 16-bit signed numbers and returns a signed 32-bit result. \text{MULU} (Unsigned Multiply) does the same but returns an unsigned result.

The general form for these instructions is:
**MULS**  effective address of source operand, destination data register

or

**MULU**  effective address of source operand, destination data register

You may use any addressing mode except Address Register Direct to specify the effective address of the source operand. For example:

\[ \text{MULU} \ #62,\text{D1} \]

will multiply whatever quantity is stored in D1 by the quantity 62 and store the result in D1. When using **MULU** or **MULS**, remember that you must use a data register as the destination in a multiplication operation.

The source operand can only be a word in length; therefore, **MULU** and **MULS** do not take extensions.

With either instruction, the overflow and carry bits of the status register are always cleared; the extend bit is not affected. **MULS** will cause the negative bit to be set if the result is negative; **MULU** will set the negative bit if the most significant bit of the result is set. (In both cases the negative bit will be cleared if the condition for setting the bit has not occurred.) The zero bit will be set when either **MULU** or **MULS** produces a zero result and cleared for a non-zero result.

## Integer Division

**DIVS** and **DIVU** perform division on signed and unsigned numbers, respectively. The general form of the instructions is:

\[ \text{DIVS} \ \text{effective address of source operand, destination data register} \]

or

\[ \text{DIVU} \ \text{effective address of source operand, destination data register} \]

The destination operand (up to 32 bits in length and contained in a data register) is divided by the source operand. The source operand can be specified using any addressing mode except Address Register Indirect and is 16 bits in length.

The result is stored in the destination data register. The lower-order half of the longword (bits 0–15) will contain the quotient. The upper half (bits 16–31) will contain the remainder. For example:

\[ \text{MOVE} \ \#33,\text{D1} \]
\[ \text{DIVS} \ \#4,\text{D1} \]
will cause D1 to receive the following contents:

%0000 0000 0000 0001 0000 0000 0000 1000

The quotient (8) is, as explained above, in the lower-order half of the 32-bit longword; the remainder (1) begins in bit 16, the first bit of the higher-order half of the longword.

Since the source operand is restricted to 16 bits, neither DIVU or DIVS take an extension. The size of the operation is always a word.

Overflows can be nasty when doing integer division. An overflow condition arises when the quotient is larger than 16 bits. If the condition is detected before the operation finishes, it is possible that the overflow bit in the status register will be set and the destination data register left unchanged. Therefore, if a division could generate an overflow, a program should check the overflow bit before assuming that the operation was completed successfully. If no overflow has occurred, the bit will be cleared.

The carry bit will always be cleared by a division. The extend bit is unaffected. The zero and negative bits are set or cleared, depending on the result of the operation.

Logical Operations

The way the integer division instructions return their results presents an interesting problem — what can you do if you are interested in the quotient, but not in the remainder? Conversely, what if you need just the remainder? To isolate the quotient, you will need to fill the high-order half of the destination data register with zeros. To isolate the remainder, you'll need to first fill the low-order half of the destination register with zeros and then swap the two halves so that your remainder is in the low-order half.

There is more than one way to selectively set the bits in a byte, word, or longword, but a commonly employed strategy is to use a logical operator and an immediate operand called a “mask.” The logical operations available in the 68000 instruction set are AND, OR, EOR, and NOT.

AND

Logical instructions work differently than any other kind of instruction — they operate separately on each bit in the operands. AND compares the state of the pair of bits that occupy the same location in each operand. If the two bits are both 1, then that bit will have a result of 1. If either or both are 0, the result will be zero. Table 5.1 summarizes the effect of ANDing two bits together. In essence, AND has the same effect as multiplying the two bits.
The **AND** instruction takes two forms:

\[
\text{AND} \quad \text{effective address of source operand, destination data register}
\]

or

\[
\text{AND} \quad \text{source data register, effective address of destination}
\]

If the effective address field is the source operand (the first form above), then you may use any addressing mode but Address Register Direct. If the effective address field is the destination of the operation, there are further restrictions; Data Register Direct, both program counter modes, and Immediate are also not allowed. The size of an **AND** operation can be specified as byte (**.B**), word (no extension or **.W**), or longword (**.L**).

\[
\begin{array}{c|ccc}
\text{AND} & 1 & 0 \\
\hline
1 & 1 & 0 \\
0 & 0 & 0 \\
\end{array}
\]

**Table 5.1** AND Truth Table

To see how **AND** works, consider the following example:

\[
\begin{align*}
\text{MOVE.B} & \quad \text{#\%00110101,D0} \\
\text{AND.B} & \quad \text{#\%11110000,D0}
\end{align*}
\]

Can you predict what the result (stored in D0) will be? Remember that **AND**ing two 1's produces a 1, but **AND**ing anything else produces a 0. The result will therefore be %00110000.

How then, can this help us when we want to isolate one part of the result of a division operation? There’s another way to look at how an **AND** works — **AND**ing something with a 1 preserves the value of the source bit. **AND**ing something with a 0 will always return a 0. To retrieve the quotient of a division, we need to zero out the high-order bits and leave the low-order bits alone. To get only the remainder, we need to first zero out the low-order bits, leaving the high-order bits untouched, and
then swap the low- and high-order halves of the register. The strategy, then, is to create a "mask" so that when we AND the mask with the data register holding the result of the division, the part we want to retain will be unaltered, but the half we don't want will be filled with Os. Let's look at an example:

\[
\begin{align*}
\text{MOVE} & \ #88,D1 \quad (D1) = \$00000058 \\
\text{DIVU} & \ #3,D1 \quad (D1) = \$0001001D \quad (\text{quotient} = 29; \text{remainder} = 1) \\
\text{AND} & \ #\$0000FFFF,D1
\end{align*}
\]

The mask in the example is $0000FFFF$ (in binary: %0000 0000 0000 0000 1111 1111 1111 1111). Therefore, the contents of D1 after the AND instruction is executed will be $0000001D$. The remainder has been "masked" off and we can now use the quotient in D1 as a quantity somewhere else in the program.

To isolate the remainder, we'll need to reverse the mask:

\[
\begin{align*}
\text{AND} & \ #\$FFFF0000,D1
\end{align*}
\]

The contents of D1 will be $00010000$. The problem with this result is that though the remainder is actually 1, the quantity D1 is 65,536. What we need now is some way to make the high-order bits the low-order bits, and make the low-order bits the high-order bits. The instruction SWAP does exactly that for the contents of any data register:

\[
\begin{align*}
\text{SWAP} & \ D1
\end{align*}
\]

will leave us with $00000001$ in D1, which is exactly what we need to work with the remainder as a quantity. Note that SWAP only works with data registers. It sets the negative flag if the most significant bit of the result is set and sets the zero flag if the entire result is zero (otherwise, both are cleared). The overflow and carry bits are always cleared; the extend bit is not affected.

\[
\text{OR}
\]

Like, AND, OR has two general forms:

\[
\begin{align*}
\text{OR} & \quad \text{effective address of source operand, destination data register}
\end{align*}
\]

and

\[
\begin{align*}
\text{OR} & \quad \text{source data register, effective address of destination operand}
\end{align*}
\]

OR is also exactly like AND with regard to restrictions on addressing modes, operand size specification, and effect of the condition codes.

OR is not like AND, though, when it comes to producing results. If you OR together two bits, the result will be 1 if either of the two bits is 1; the result will be 0 only if both input bits are 0 (see Table 5.2).
Table 5.2 OR Truth Table

What do you think the final contents of D1 will be after we execute these instructions?

```
MOVE.B  #%00001111,D1
OR.B    #%10101010,D1
```

The final contents of D1 will be %10101111. The only places where the result will have 0s are those bit positions in which there were 0s in the OR instruction operands.

**NOT**

The general form of the **NOT** instruction is:

```
NOT  effective address of destination operand
```

The instruction *inverts* the bits in the destination operand, which can be specified using any addressing mode but: 1) Address Register Direct; 2) Program Counter with Displacement; 3) Program Counter with Index; 4) Immediate; and 5) Quick Immediate. In other words, it replaces each 0 with a 1, and each 1 with a 0. For example, if D1 contains %0000 0001 1111 1010 0000 1111 1000 1011, then:

```
NOT.L  D1
```

will place %1111 1110 0000 0101 1111 0000 0111 0100 in D1. Note that **NOT** takes an extension (.B, .W, or .L) to specify the size of the operand.

**NOT**, like the other logical operators, does affect the condition codes in the status register. The negative and zero bits are set or cleared according to the result of the operations, overflow and carry are always cleared, and extend is not affected.
EOR

You may remember that AND does a multiplication of two bits and returns the result. EOR (Exclusive OR) adds two bits and returns the results. (In the case of 1 + 1, it returns 0 and throws away the carry.) Therefore, EOR will produce a result of 0 when both input bits are the same (either two 1s or two 0s); the result will be 1 when the two inputs are different (0 and 1). See Table 5.3 for the truth table.

For example:

```
MOVE.B  #00001111,D2
EOR.B   #11001100,D2
```

will place %11000011 in D2.

EOR functions exactly like AND and OR in terms of operand size specification, address mode restrictions, and effect on the condition codes.

<table>
<thead>
<tr>
<th>EOR</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3 EOR Truth Table

Subroutines

Assembly language programs are never famous for their elegant structure, but you can achieve some semblance of order if you break your program into modules by placing blocks of code that perform a single function into subroutines.

Assembly language subroutines are much like Pascal procedures used in a program where all variables are defined in the program's var block (i.e., all variables are global). All storage locations defined by data allocation directives are available to all subroutines (i.e., they are also global). You must take care that you place your subroutines so that the main program will not drop into them, since
assembly language programs execute sequentially unless they encounter a branch instruction. Generally, that means that subroutines will be placed toward the end of the source code, after the statement that forms the logical end to the main program.

To call a subroutine, you may JSR (jump to subroutine) or BSR (branch to subroutine):

\[
\text{JSR} \quad \text{symbolic address of first statement of subroutine}
\]

or

\[
\text{BSR} \quad \text{symbolic address of first statement of subroutine}
\]

The difference between the two is the same as the difference between JMP and BRA. During assembly, the symbolic address following JSR is turned into an absolute address. When the JSR is encountered during program execution, the absolute address replaces the contents of the program counter, and program execution continues with the instruction at that address.

On the other hand, assembling a BSR instruction creates an offset equal to the number of bytes between the BSR instruction and the start of the subroutine. The Macintosh will add the offset to the contents of the program counter to determine the absolute address for the first instruction of the subroutine.

Even before changing the contents of the program counter, both instructions cause the address of the following instruction to be pushed onto the stack. This is the address where program execution will continue when the subroutine is finished. Subroutines end with RTS (return from subroutine). RTS pulls the address that JSR or BSR pushed onto it from the top of the stack and puts it in the program counter.

Remember that because the stack is a last in, first out device, nested subroutines will always return to whatever routine called them. There may be situations, though, where you don't need to thread your way up through nested subroutines but would like to return directly to, for example, a main program. You can do this by pulling one longword off the stack for each level of subroutine nesting you want to skip. An example of this technique appears in the following section.

**NOTE:** RTS has a special use in Macintosh assembly language. While most assembly language programs signal the logical end of the program with something akin to an END statement, a Macintosh assembly language program does not. (Remember that END is an assembler directive that signals the physical end of your source code.) Whenever your program should return to the Finder (i.e., a logical ending place), use RTS. It has the same effect as END in a Pascal program. It is an executable statement that stops the assembly language program and returns control of the system to the Finder. Obviously, this will only work if there are no subroutine return addresses on the stack. (If there are, the RTS will simply return you to the statement below the last encountered JSR or BSR.)
Putting the Instruction Set to Work — Sorting and Searching Arrays

Among the things that assembly language does exceptionally well are sorting and searching. When the data to be sorted and searched are kept in an array in main memory, the processes execute at breakneck speed. Let's look, then, at how the video tape index program maintains its master file as a sorted RAM array which can be searched using an efficient binary search technique. You will see that understanding the individual instructions is really a very minor part of the task; it's figuring out which instructions to use when that's the challenge.

Introduction to the Video Tape Index Program

The video tape index program actually uses two files. The first — TAPE.MASTER — is a sequential file that is read into a main memory array at the start of program execution. All changes to TAPE.MASTER are made while it is in main memory. It is rewritten to disk just before the program ends (when the user selects QUIT from the OPTIONS menu). The second file — ANNOTATIONS — is a direct access file that is kept on disk. Since the annotations can be rather long (up to 256 characters each), they are brought into memory only as needed.

Though TAPE.MASTER is a sequential file, it nonetheless has fixed field lengths, and therefore fixed record lengths. Why? To locate a particular field in a particular record in main memory, you must know exactly where each piece of data will begin relative to the starting address of the array. This is not possible if the ends of fields depend only on the number of characters in each individual piece of data.

NOTE TO PURISTS: There is a way to manage this data with variable field and record lengths by preceding the records with a look-up table that gives the relative starting location of each record; the programming required to maintain and especially to search such a structure is far more complex than that required by the video tape index program.

The structure of TAPE.MASTER is:

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TapeName</td>
<td>30 characters</td>
</tr>
<tr>
<td>Producer</td>
<td>20 characters</td>
</tr>
<tr>
<td>ReleaseDate</td>
<td>4 characters</td>
</tr>
<tr>
<td>Rating</td>
<td>4 characters</td>
</tr>
<tr>
<td>TapeNumber</td>
<td>4 characters</td>
</tr>
<tr>
<td>AnnotNum</td>
<td>1 word (2 bytes)</td>
</tr>
</tbody>
</table>

Total record length = 64 bytes
To allocate space in main memory to hold the array, we need a very large block of storage set aside in the applications globals area:

**TapeArray** DS.B 6400

This statement allocates enough storage for 100 records (one byte for each of the 64 characters in each record). Note that because this statement uses a DS directive, there is no way to automatically assign a starting value to each byte in the storage block when the program is assembled.

New records are entered into a temporary area called **NewRecord**. **NewRecord** is a data structure (we'll talk more about data structures in Chapter 6) defined as follows:

**NewRecord** DS.B 64

Offsets into the record are defined as equates at the top of the program:

- oTapeName EQU 0
- oProducer EQU 30
- oReleaseDate EQU 50
- oRating EQU 54
- oTapeNumber EQU 58
- oAnnotNum EQU 62

The symbolic address **NewRecord** refers to the starting address of this data structure in memory. A program can, however, get to any field within the structure by using Address Register Indirect With Offset Addressing. For example, to specify the starting location of the ReleaseDate field, use:

**NewRecord + oReleaseDate(A5)**

Remember that because space for **NewRecord** is allocated with DS, its address is relative to (A5), the start of the applications globals area.

### Inserting New Records into a Sorted Array

In order to do a binary search, the array you are searching must be in some order. TAPE.MASTER is kept in alphabetical order by the name of the tape. Though there are many ways of sorting an array, one simple technique for inserting a new record into an array that is already in order is the straight-insertion method. To understand the process, take a look at Listing 5.1, pseudocode that describes the video tape index's straight-insertion sort.
Listing 5.1 Pseudocode for Straight Insertion Sort

Get number of records in TapeArray.
Subtract 1 from number to records to get record number of last record (the record pointer).
Initialize two character pointers to the first character, one for TapeArray and one for NewRecord.
If record to be inserted is not the first record then

Repeat
    Get next character from TapeArray record indicated by record pointer;
    Get next character from NewRecord;
    If character from TapeArray is greater than character from NewRecord then
        Move entire TapeArray down one record position;
        Decrement record pointer;
        Reset character pointers to first character
    Until entire name field has been compared OR character from NewRecord is greater than character from TapeArray OR record pointer is -1.

Add 1 to record pointer to obtain record number where NewRecord will be inserted into TapeArray.

Move characters from NewRecord to TapeArray.
Increment the total number of records in TapeArray.

The strategy involves comparing the data to be inserted with the bottom record in the array. If the new record should be placed “above” the last record, the last record is moved down, in effect creating a hole in the array. The new record is then compared to the last record but one. If the new record should be placed above the last record but one, the last record but one is moved into the hole created when the last record was moved. This process is repeated, making comparisons between the new record and records already in the array from the bottom up, until such time as the new record is equal to or less than a record in the array. Once that condition is encountered, the new record is inserted into the hole in the array (thus the name, straight-insertion sort).

Locating Data Stored in Arrays

Before examining the subroutine that performs the straight-insertion sort in detail, let’s look at accessing data stored in arrays. How do we do it? We use Address
Register Indirect with Index addressing.

The starting address of the TAPE.MASTER array in main memory is given by the symbolic address TapeArray(A5). (There is no need for us to know its absolute address.) The starting address of any given record will therefore be equal to:

\[
[(\text{Record Number } \times 64) + \text{TapeArray(A5)}]
\]

where 64 is the number of bytes in a record. (In this case the characters are packed into adjoining bytes.) The expression in brackets above is an offset into the TAPE.MASTER array. If this expression seems a bit confusing at first, remember that in a computer, numbering systems generally begin with 0 rather than 1 (i.e., the second record will have a record number of 1). We might use that quantity as the displacement in Address Register Indirect with Index addressing.

Using the displacement locates the start of one particular record. It does not, though, locate a particular field or character that is a part of the record. (The displacement is an offset with the array.) To do that, we need an additional offset within the record. The index register portion of the effective address could be used for that purpose. The same equates that hold offsets into NewRecord can be used as offset into a TapeArray record since both have the same structure. To locate, for example, the first character in the Rating field, oRating might first be placed in a data register:

\[
\text{MOVE } \#\text{oRating},D0
\]

Then, an effective address for the first character in the Rating field would appear as:

\[
\text{Offset(A0,D0)}
\]

where Offset is computed by the method described above, and the address of TapeArray(A5) has previously been stored in A0.

There is a major problem with using the offset as a displacement, however, in an array the size of the one used by the video tape index program. With Address Register Indirect with Index addressing, the displacement is limited to a range of \(-128\) to \(+127\). As soon as there are more than three records in the array, the offset will exceed that range. It is therefore easier to manually compute an address into TapeArray.

For example, to locate the beginning of the Rating field:

1. Get the starting address of TapeArray:

\[
\text{LEA } \text{TapeArray(A5),A0}
\]
2. Compute the record offset (assume the record number is in D0):

   \textbf{MULU} \#64,D0

3. Add the record offset to the start of the array:

   \textbf{ADD} D0,A0

4. Add the field offset:

   \textbf{ADD} \#oRating,A0

It may also be necessary to step through a record, character by character. In that case, the strategy is to initialize the index register with the offset into the array and then increment it by 1 to move to each successive character.

Assuming that D0 is used as the index register and that A0 contains the starting address of TapeArray, an entire 64-character record could be handled using the following code:

\textbf{MOVE} RecordNumber,D0
\textbf{MULU} \#64,D0 ;compute offset
\textbf{MOVE} \#0,D7 ;initialize character counter
\textbf{Loop}
\textbf{MOVED.B} (A0,D0),D1 ;get one character
\{process the character in some way\}
\textbf{ADDQ} \#1,D0 ;increment index register
\textbf{ADDQ} \#1,D7 ;increment character counter
\textbf{CMP} \#64,D7 ;have 64 char. been handled?
\textbf{BNE} Loop ;return to get another
\textbf{RTS}

This technique is used to compare the name of the tape in the record to be inserted with tape names already in the file.

\textbf{The Straight-Insertion Sort}

The assembly language version of the straight-insertion sort appears in Listing 5.2. This is part of the subroutine that handles the entry of new records. It assumes that the data to be inserted into TapeArray has been collected in \textit{NewRecord}.

Since the sort starts by looking at the last record in the array, the record number of the first record to be considered will be equal to the total number of records in the file. Therefore, the sort first loads that quantity into D1 [(a) in Listing 5.2]. When computing offsets, though, the quantity should be one less than the number of records. (Remember that the records are numbered beginning with 0.) The routine therefore subtracts 1 from the number of records. The program statement at (b) initializes D2 as a character counter and index register.
Listing 5.2 Straight-Insertion Sort (Version 1)

(a) **Sort**

MOVE TotalRecords,D1
SUBQ #1,D1
MOVE #0,D2

(b) **LEA**

TapeArray(A5),A1

(c) **LEA**

NewRecord(A5),A2

(d) **CMP**

#0,D1

(e) **BEQ**

InsertNew

(f) **JSR**

ComputeOffset1

(g) **MOVE.B**

(A1,D6),D3

(h) **MOVE.B**

(A2,D2),D4

(i) **CMP**

D3,D4

(j) **BGT**

InsertNew

**BT**

MoveOld

(k) **ADDQ**

#1,D2

(l) **ADDQ**

#1,D6

(m) **CMP**

D3,D4

(n) **BEQ**

InsertNew

(o) **BRA**

NextChar

(p) **MOVE.B**

(A2,D2),(A1,D7)

(q) **ADDQ**

#1,D7

(r) **CMP**

#64,D6

(s) **BNE**

Again

(t) **SUBQ**

#1,D1

(u) **CMP**

-D1,D1

(v) **BEQ**

InsertNew

(w) **BRA**

Checking

(x) **LEA**

Tota1Records,A0

(y) **ADDO**

#1,(A0)

(z) **JSR**

ComputeOffset2

**BRA**

AllDone

;return - sort is complete (continued)
Listing 5.2 (continued)

```
ComputeOffset1
(r)
    MOVE  D1,D6 ;offset = record # * 124
    MULU  #64,D6
    RTS

ComputeOffset2
    MOVE  D5,D7
    MULU  #64,D7
    RTS
```

We also need to place the starting addresses of `TapeArray` and `NewRecord` in address registers so they can be used as part of Address Register Indirect with Index addressing. That happens at (c); A1 will hold the address for `TapeArray` and A2 will hold `NewRecord`'s address. If the record being inserted is the first record (e.g., the array is empty), the record is simply moved to `TapeArray` without further processing (d). If `TapeArray` already has some records, the sort needs to begin comparing characters.

The first task is to compute the offset for the record indicated by D1. If you look at (e), you'll see that offsets are computed in subroutines. The subroutine `ComputeOffset1` (r) uses the contents of D1 as the record number; `ComputeOffset2` bases its computations on D5. To compute an offset, the program:

1. Moves the record number into a temporary storage register (D6) and then
2. Multiplies the record number by the record length (64 bytes).

Once the offset is computed, one character from the tape array can be loaded into D3 (f). A character from the new record is loaded into D4 (g). The characters are compared to each other (h). If the character from the new record is greater (comes later in an alphabetical sequence) than the character from the array, then the place to insert the new record has been found. The program branches to do the insertion (n). If the character from the new record is less than the character from the array, then the record in the array must be moved "down" one position (j). On the other hand, if the two characters are equal, there are two possibilities.

If 30 bytes (the total length of the field) have been examined, then the names of the two tapes are equal. The program checks for this condition by incrementing the index register/character counter in D2 and then comparing the new value with 30 (i). The video tape index program inserts records with duplicate tape names without further ordering. Therefore, if the two fields are equal, the new record can be inserted directly after the old one.

If all 30 characters have not been checked, then it is not possible to make a judgment about whether to insert a record or move an existing one. The only
recourse is to check the next character in the field. Therefore, the program branches back to (f) to begin the comparison process again.

Moving an existing record down (j) is done character by character. The first task is to compute two offsets: one to the beginning of the record (Offset1 in D6) which will be moved; and the other to the beginning of the location to which it will be moved (Offset2 in D7). Then, statement (k) moves a single character. The index registers are incremented and then checked against the number of bytes in the record (64) to determine if all of the characters have been moved. If not, the program branches to move another character.

Once an entire record has been moved, the contents of D1 are decremented (m). Why decrement the record number? Simply because a straight insertion sort starts at the bottom of the array and moves toward the beginning. If, after the decrement, D1 contains –1, then the new record comes before all others already in the file and should be inserted. In that case, the program branches to insert the new record (n). Otherwise, the program branches to begin a new comparison (e).

Inserting a new record (n) involves moving characters one by one from NewRecord into the array. The insertion position is one record beyond the one pointed to by D1. Therefore, D1 is incremented. The offset into TapeArray is computed (o) and a single character is moved (p). The index registers (D2 and D7) must then be incremented to count the characters just transferred. Just as with the procedure for moving an existing record down, the number of characters moved is compared against the total number of characters which must be moved (64) to determine if the process is complete. If not, the program branches to move another (p).

Once the new record is inserted into the array, only one task remains — incrementing the total number of records. The absolute address associated with the symbolic address NumRecords is loaded into address register A1 (q). The quantity stored at the address in A0 can then be incremented with a single instruction. Now NumRecords reflects the number of records in the array after the new record has been inserted. This action completes the straight-insertion sort.

**Locating Records in a Sorted Array**

One of the fastest ways to search an ordered list is to use a binary search. The binary search strategy involves looking at the middle record in the list, deciding whether the record you want is above or below the middle, and then looking only in the half of the list where the record could occur. The file is repeatedly cut in half, always looking at the middle record, until either the desired record is located or it is apparent that the record wanted isn't present in the array.

Pseudocode for a binary search appears in Listing 5.3. TapeArray refers to the array in RAM while searchString contains the tape name to be found. A pointer to the top record being considered is initialized to 0. (When we talk about the top and bottom of the array, we think of the array as if it were written on a sheet a paper, with record number 0 at the top.) The pointer to the bottom of the array is
initialized to the total number of records minus one. Then we compute the number of the middle record by adding top to bottom and dividing by 2.

If the record we are looking for is above the middle record, then we move the bottom pointer up to the middle; if it is below the middle record, we move to top pointer down to the middle. We know that a search has been unsuccessful (the record we want isn't present in the array) if the two pointers cross (i.e., bottom becomes greater than top).

We must handle the top two records and the two bottom records in the array separately. Therefore, if the computation of the middle record number generates a result equal to 1 or 2, the program does a sequential search of the first two records; if the computation generates a record number equal to the total number of records – 1 or the total number of records, the program searches the last two records sequentially. Note that these two “special case” searches occupy more than 1/2 of the pseudocode listing.

A binary search also falls apart if there are less than four records in the array. If you wish to handle such a possibility in a program, check the number of records in the array before beginning the search. If there are less than four records, search the array sequentially. The video tape index program assumes that there will never be less than four records—a fairly realistic assumption considering the nature of the application—and therefore does not handle that situation.

Listing 5.3 Pseudocode for Binary Search

Set bottom record pointer equal to total number of records - 1.

Set top record pointer equal to 0.

Initialize character pointers for TapeArray and SearchString.

Repeat

Compute number of middle record;

If middle record is not one of the first two records or last two records then

Get next character from TapeArray;

Get next character from SearchString;

Compare the characters;

If character from name field of TapeArray record is greater than character from SearchString then

Make bottom pointer equal to the middle record number;

Reset character pointers to beginning of records

(continued)
Else

Make top pointer equal to the middle record number;

Reset character pointers to beginning of records

Until all characters in name field have been compared and are equal OR top pointer is greater than bottom pointer OR middle record is one of first two records or last two records.

If top pointer is greater than bottom pointer then

Report "No Find"

Else

If all characters in name field have been compared and are equal then

Location of record with SearchString is middle record number.

Else {must be first two or last two records}

If middle record is one of first two records then

Set middle record to Ø for first record;

While character from name field of TapeArray record is equal to character from SearchString do

Get next character from TapeArray;

Get next character from SearchString;

If all characters in name field of TapeArray record are equal to all characters in SearchString then

Location of SearchString is record Ø

Else

Set middle record to 1 for second record;

While character from name field of TapeArray record is equal to character from SearchString do

Get character from TapeArray;

Get character from SearchString;

If all characters in name field of TapeArray record are equal to all characters in Search String then

Location of SearchString is record 1

(continued)
Listing 5.3 (continued)

Else

Report "No Find";

Else

Set middle record number equal to last record - 1;

While character from name field of TapeArray record is equal to character from SearchString do

Get next character from TapeArray;

Get next character from SearchString;

If all characters in name field of TapeArray record are equal to all characters in SearchString then

Location of search string is last record -1

Else

Set middle record number equal to last record;

While character from name field TapeArray record is equal to characters in SearchString do

Get next character from TapeArray record;

Get next character from SearchString;

If all characters in name field of TapeArray record are equal to all characters in SearchString then

Location of SearchString is last record

Else

Report "NoFind".

The assembly language version of the binary search is a subroutine called by three modules in the video tape index program (Select, Change, and Delete). The code appears in Listing 5.4.

The name of the tape for which the routine is searching is stored in the first field of NewRecord. When a search is successful, the record number of the record in the array whose TapeName matches the tape name in NewRecord is returned in D5 (this assumes that the records are numbered beginning with 0). If a search is unsuccessful, then the routine returns a -1 in D5.
Listing 5.4 Binary Search

NameSearch ;result appears in D5 (-1 = no find)
(a) LEA TapeArray(A5),A2
LEA NewRecord(A5),A2
MOVE TotalRecords,D1
(b) SUBQ #1,D1 ;bottom pointer
MOVE D1,D3
(c) SUB #1,D3 ;save total number of records-1 for later reference
(d) MOVE #0,D2 ;top pointer

MidPoint
(e) MOVE D2,D5 ;find middle record #
ADD D1,D5
DIVU #2,D5
(f) AND.L #$0000FFFF,D5;mask off remainder
(g) CMP #1,D5
(h) BLE TopRec ;handle first two records
CMP D5,D3
(i) BLE BottomRec ;handle last two records
MOVE #0,D4 ;initialize index
JSR ComputeOffset2
CheckChar
MOVE.B (A2,D7),D0 ;character from array
MOVE.B (A1,D4),D6 ;character from search string
(j) CMP D0,D6
(k) BPL BottomHalf ;are two fields exactly alike?
(l) BMI TopHalf
(m) ADDQ #1,D4
ADDQ #1,D7
BNE CheckChar
RTS

(n) BottomHalf
MOVE D5,D2 ;move top pointer down
BRA NoFindCheck
(o) TopHalf
MOVE D5,D1 ;move bottom pointer up
(p) NoFindCheck
CMP D2,D1
BMI NoFind ;pointers have crossed
MOVE #0,D4 ;reset index
BRA MidPoint ;find new middle record and go again
(q) NoFind
MOVE #-1,D5 ; -1 = no find
RTS
(r) TopRec
MOVE #$0,D5
JSR OneCheck
MOVE #1,D5
JSR OneCheck
MOVE #-1,D5 ;no find
RTS

(continued)
Listing 5.4 (continued)

(s) BottomRec
    MOVE D3,D5
    JSR OneCheck
    ADDQ #1,D3
    MOVE D3,D5
    JSR OneCheck
    MOVE #-1,D5 ;no find
    RTS

(t) OneCheck
    MOVE #0,D4
    JSR ComputeOffset2

OneMore
    MOVE.B (A1,D7),D0 ;character from array
    MOVE.B (A2,D4),D6 ;character from search string
    CMP D6,D0
    BNE WrongOne
    ADDO #1,D4
    ADDO #1,D7
    CMP #30,D4
    BNE OneMore

(u) MOVE.L (SP)+,D7 ;pop two subroutine return addresses off stack
    RTS ;return directly to "Select" routine

(v) WrongOne
    RTS ;return to Top or Bottom

Conducting a Binary Search

To begin the binary search, we load the addresses of the two data structures and
the value of the one constant that the search will need to reference (TapeArray,
NewRecord, and TotalRecords) into registers [(a) in Listing 5.4]. The record
number of the last record in the array (equal to the total number of records minus
one, since the records are numbered beginning with zero) is saved in D1 (b) as the
bottom pointer. The number of the last record but one is moved to
D3 for future
reference (c), and the top pointer — held in D2 — is initialized to 0 (d).

To compute the middle record (e), we sum the contents of the top and bottom
pointers and then divide by 2. Then the remainder portion of the result is removed
by ANDing the destination location (D5) with the appropriate mask (f). If you can't
remember how this works, refer back to the section in this chapter that deals with
AND.

The next step in the search is to determine whether the middle record is either
the first or second record (h) or one of the last two records (i). If it is, then the
program must branch to examine those records separately.

Otherwise, the routine must compare the name of the tape in the middle record
with the name of the tape for which we are searching. The comparison (j) is
performed in the same way as the comparisons in the straight-insertion sort.
There are three possible results of the comparison. The character from the name of the tape for which we are searching may be greater than the name of the tape in the middle record \((k)\). If so, the top pointer is moved down to equal the middle record \((n)\). If the character from the name of the tape for which we are searching is less than the character from the name of the tape in the middle record \((l)\), the bottom pointer must be replaced by the number of the middle record \((o)\). In either case, before proceeding to compute another midpoint, the program needs to determine if the two pointers have crossed \((p)\).

A top pointer greater than a bottom pointer indicates that the record for which we are searching is not in the array \((q)\). The search routine loads the "no find" flag \((a = 1)\) into D5 and returns to the calling program. If the pointers have not crossed, then the search must continue by computing another midpoint \((e)\) and repeating the entire comparison procedure.

On the other hand, if the two characters being compared are equal, then it is not possible to decide immediately whether the correct record has been found or whether the character checking must continue. The deciding factor is the total number of characters that have been checked. If all 30 characters are alike \((m)\), the name of the tape for which we are searching is exactly the same as the name of the tape in the middle record. The search therefore ends successfully \((\text{the number of the middle record remains in D5})\) and the subroutine returns to the calling program.

The top and bottom two records are searched sequentially \((r,s)\). For example, if the middle record was computed to equal either 0 or 1, then 0 is loaded into D5. The comparison between the name of the tape for which we are searching and the name of the tape in record 0 is performed by the subroutine OneCheck \((t)\). The procedure is exactly the same as that used earlier in the program beginning at the symbolic address CheckChar.

Assuming that the search of record 0 is successful, there is no need for the routine to return to Top \((r)\), where it was called; it can return directly to the part of the program that called the entire search. To "skip" one subroutine level, we need to pull one subroutine return address off the stack. At \((u)\) the top of the stack is moved into D7 and, since Postincrement addressing is used, the stack pointer is also incremented. Remember that incrementing the stack pointer has the effect of removing the top item from the stack. The \texttt{RTS} that follows will therefore transfer program control back to the original calling program.

If the search of record 0 is unsuccessful \((u)\), then the search continues with record 1. An unsuccessful search of record 1 indicates a "no find." The two bottom records are handled in exactly the same manner.

This binary search technique can be used with any ordered file or array that contains more than three records. Since TAPE.MASTER is ordered by tape name, that is the only field on that will support a binary search. If we need to retrieve tapes by something other than the name of the tape, there are two alternatives: reorder the array on the field to be searched, or do a sequential search. The video tape index program uses the latter approach.
Questions and Problems

1. Show how the decimal numbers below would be stored as 16-bit binary integers in a 2's complement system.
   a. 12  
   b. -12  
   c. 84
   d. -84  
   e. 603  
   f. -603  
   g. 2006  
   h. -2006

2. Convert your answers from problem 1 to their hexadecimal representation.

3. Convert the 2's complement integers below from hexadecimal to binary and then to their true magnitude in decimal. Remember to consider the high-order bit as a sign bit.
   a. 0016  d. FF00  g. 88BC
   b. EA14  e. 010A  h. 0333
   c. 1183  f. 4100

4. Indicate whether each of the following represent legal 68000 instructions. For each illegal instruction, describe what is wrong with it.
   a. ADD SomePlace, D0  
   b. ADD.L D0, SomePlace  
   c. ADD D0, Locate(A5)  
   d. SUB D0,#8  
   e. SUB #8,D0  
   f. SUB #10,A0  
   g. SUB (SP)+,D0  
   h. MULU D1,D7
   i. MULU.L SomePlace, D0  
   j. MULU Locate(A5), D6  
   k. DIVS #12, D3  
   l. DIVU # -6, D2  
   m. DIVS.L #$FF00, D6  
   n. AND #6,D0  
   o. AND D6,A1  
   p. OR D2,#%11110000

5. For the following blocks of code:
   A. indicate the contents of the destination register after the code has been executed
   B. indicate the state of each of the flags in the user byte of the status register after the code has been executed.
   a. MOVE #44,D0  
   MOVE #86,D1  
   ADD D0,D1
   b. MOVE #186,D0  
   MOVE # -99,D1  
   ADD D0,D1
   c. MOVE # -186,D0  
   MOVES #99,D1  
   ADD D0,D1
   d. MOVE #99,D0  
   MOVE #106,D1  
   ADD.B D0,D1
e. MOVE #12,D0
   MOVE #10,D1
   MULU D0,D1

j. MOVE #%11110000,D0
   AND.B #%00110011,D0

k. MOVE #%11110000,D0
   OR.B #%00010011,D0

l. MOVE #$00AB,D0
   AND #$FFFF,D0

m. MOVE #$00AB,D0
   OR #$FFFF,D0

n. MOVE #$00AB,D0
   EOR #$F0F0,D0

o. MOVE #$124A,D0

i. MOVE #80,D0
   MOVE #-8,D1
   DIVS D0,D1

6. Indicate the contents of registers D0 and D1 when the block of code below finishes executing.

   MOVE #6,D0
   MOVE #0,D1

   Top
   ADD #4,D1
   SUB #1,D0
   CMP #0,D0
   BNE Top

   ......

7. Consider the following block of code:

   MOVE #0,D0
   MOVE #0,D1
   LEA Start(A5),A0

   Top
   MOVE (A0,D0),D2
   BEQ Done
   ADD D2,D1
   ADD #2,D0
   BRA Top

   Done

   Start DS 20
A. What does this code do?  
B. Why is the index register, D0, incremented by 2 rather than 1? Hint: consider the size of the addition instruction's operands.

8. A. What Pascal operation does this block of code simulate?
   
   DIVS  Operand2,D0  
   AND.L  #$FFFF0000,D0  
   SWAP  D0  

B. What Pascal operation does this block of code simulate?

   DIVS  Operand2,D0  
   AND.L  #$0000FFFF, D0  

9. Write an assembly language subroutine that will take an operand from register D0 and compute its square.

10. Write an assembly language subroutine that computes the factorial of a word-sized operand which is passed to the subroutine in D0 (n! = 1 * 2 * 3 * .... n – 1 * n).

11. Write an assembly language subroutine that checks an array of ten characters (stored in consecutive main memory locations) and returns the array position of the character which is alphabetically last. Place the result in register D7.

12. Write an assembly language subroutine that checks a character string stored in main memory and counts the occurrences of a given character within that string. The address of the first character in the string is passed to the subroutine in A0; the ASCII code of the character being counted is in D0. Though the length of the string is unknown, its last character is a double quote (").

13. One assembly language instruction you have not seen is a "shift." A left shift moves the bits in the operand one position to the left and puts a zero in bit 0. A right shift moves the bits one position right and fills the high-order bit with a zero.

   As an example, let's consider a byte-sized shift. The byte at the address $1A2B contains the quantity 01100110. The instruction ASL $1A2B (ASL = arithmetic shift left) produces the result 11001100. The instruction ASR $1A2B (ASR = arithmetic shift right) produces the result 00110011.

   What does a left shift do? What does a right shift do? Hint: the answer is closely tied to the fact that the contents of the byte is a quantity rather than an address or an ASCII code.
Chapter Objectives

1. To review Pascal elementary data types
2. To review Pascal user-defined data types
3. To review Pascal data structures (arrays and records)
4. To review Pascal syntax for procedure and function calls
5. To take a first look at translating the Pascal syntax of the ToolBox and operating system routines into assembly language
6. To understand the general organization of the ToolBox and operating system routines
7. To learn more details about the trap mechanism that provides access to the ToolBox and operating system routines

Yes, this is an assembly language book, not a Pascal book. Nevertheless, the Macintosh's internal routines were created with the Pascal programmer in mind. It will therefore not only simplify the process of mastering the ToolBox and operating system routines, but make it possible for you to read Macintosh documentation if you are comfortable with Pascal data types, their assembly language equivalents and how additional data types and data structures are constructed from elementary data types.
Pascal Elementary Data Types

There are six elementary data types from which all other data types are developed — three numeric, two character, and one logical.

Numeric Data Types

Integers

Integers are stored as either INTEGER or LONGINT (longinteger). An INTEGER occupies two bytes. Whenever the specifications for a ROM routine require an INTEGER, you must set aside two bytes of storage somewhere. A LONGINT occupies 4 bytes, which means you must allocate the full four bytes anywhere a LONGINT is required.

The most significant bit in an INTEGER is used as a sign bit. If bit 15 is clear, then the number is positive; if it is set, the number is negative. The remaining 15 bits hold the number. Therefore, the maximum value that can be stored in an INTEGER location is 32,767; the minimum is –32,768. A Pascal INTEGER is therefore exactly the same as the Macintosh’s 16-bit word.

To set aside space for an INTEGER you must:

SymbolicAddress DC.W initial value

A LONGINT also retains the most significant bit as a sign bit. Bit 31 will hold a 0 for a positive number and a 1 for a negative number. The maximum quantity that you can store in a LONGINT is 2,147,483,647; the minimum is –2,147,483,648. A Pascal LONGINT is therefore exactly the same as the Macintosh’s 32-bit longword.

To declare space for a LONGINT, use:

SymbolicAddress DC.L initial value

Integer and longintegers are stored using the two’s complement system described in Chapter 5. Why use the two’s complement form? The answer lies in how arithmetic operations are done. With a two’s complement system, you can perform a subtraction using addition. In other words, to do a subtraction you take the two’s complement of the subtrahend (the number on the bottom in a subtraction operation) and add it to the minuend (the number on the top). The result is the same as if you did a standard subtraction. A computer designed to use two’s complement arithmetic, therefore, only requires hardware which can do addition; it doesn’t need special subtraction circuitry.
There are times, when you’re in the midst of developing a Macintosh assembly language application, that being able to handle two’s complement numbers is very handy. For example, the File Manager (the part of the ToolBox that provides for file I/O) returns a result code in D0 after each call to one of its routines. A successful file operation has a result code of 0, but all the other result codes are negative.

If you are monitoring the progress of the program with the debugger, then you can use that result code as a clue to why an attempted file operation failed. Suppose that the program attempted to write something to the disk, but the disk was full. The result code for a disk full error is −34, but the contents of D0 appear as $FFDE. Believe it or not, $FFDE is the two’s complement representation of −34. To prove it, let’s convert $FFDE back to its true magnitude form:

Step 1: Convert the hexadecimal digits to binary

$FFDE = \%1111\ 1111\ 1101\ 1110$

Step 2: Invert the digits

\%
1000\ 0000\ 0010\ 0001
(Note that the highest order bit does not participate in the magnitude of the number; it is a sign bit)

Step 3: Add 1

\%
1000\ 0000\ 0010\ 0010

Step 4: Convert the binary to hexadecimal

\%
1000\ 0000\ 0010\ 0010 = −22_{16}

Step 5: Convert the hexadecimal to decimal

$−22_{16} = −(16 \times 2) + 2 = −34$

### Real Numbers

Real numbers, stored as the Pascal data type REAL, occupy 4 bytes. The number is broken into three parts: the mantissa (the fractional part of the number), the exponent (the power to which 2 is raised and then multiplied by the mantissa), and the sign of the mantissa. All quantities are binary.

The Macintosh makes no use of the Pascal data type REAL. Arithmetic operations on numbers that contain fractional portions are handled by FP68K, the floating point arithmetic package. FP68K is discussed in detail in Chapter 12.
Character Data Types

The data type CHAR occupies two bytes. The ASCII code of the character is stored in the low-order byte (bits 0-7); the high-order byte is unused. Though it may seem like a waste of space to use 16 bits to store an eight-bit code, it is nonetheless the way the Macintosh was designed to handle single characters. Fortunately, whenever Macintosh needs to deal with more than one character at a time, the ASCII codes are packed into adjoining bytes.

The Toolbox routine __DrawChar requires data stored as CHAR, which is why we've been moving an entire word (e.g., $0040) onto the stack rather than just the eight bits occupied by an ASCII code. To allocate space for a CHAR, use:

\[
\text{SymbolicAddress DC.W initial value}
\]

Pascal also has a data type to handle strings — STRING[n]. The overall length of the string is \( n + 1 \) bytes. The first byte contains the length of the string. The rest of the bytes contain the ASCII codes of the characters. For example, STRING[255] (also written Str255) requires 256 bytes of storage and will accommodate a string of up to 255 characters. Note that even though the definition allows 255 characters, you need not use them all.

Since a STRING requires more space than a longword, it can be specified by using a constant block:

\[
\text{SymbolicAddress DCB.B length, initial value}
\]

For example, a Str255 data item could be accommodated by:

\[
\text{Label DCB.B 256, " "}
\]

Strings that are defined in assembly language programs are not automatically assembled with length bytes. By default, strings that are defined by LEA or PEA instructions are placed immediately after program code and are given a length byte. On the other hand, strings defined by any form of DC directive are allocated space in the place where they occur in the application source code. They do not have length bytes. This distinction can be important, since a number of Toolbox routines have parameters of type Str255 and therefore expect the first byte to be a length byte.

The default allocations can be overridden with the STRING_FORMAT assembler directive. The format of the directive is:

\[
\text{STRING_FORMAT value}
\]

STRING_FORMAT's value is two bits. The first bit determines how LEA and PEA strings will be handled. If it has a value of 1 (the default), these strings will be assembled with a length byte. A value of 0 assembles the text without a preceding length byte but with a trailing 0.
The second bit affects the format of DC strings. A value of 0 (the default) produces strings with no length byte and no trailing 0. A value of 1 will assemble the strings with a length byte. If you wish both types of strings to be assembled with length bytes, use:

\textbf{STRING\_FORMAT 3}

The 3 is the decimal equivalent of a two-bit number with a 1 in each bit.

\textbf{The Logical Data Type}

Pascal's final elementary data type is BOOLEAN. Though a BOOLEAN occupies two bytes, only one bit is important. Bit 8 holds a 1 if the word has the value of true, a 0 if the value is false; all other bits are cleared. If you define a BOOLEAN as:

\textbf{SymbolicAddress DC.W initial value}

then you can compare the symbolic address against 0 to test for a value of false, but you must test against 256 to check for a value of true.

\textbf{User-Defined Data Types}

As you probably remember, Pascal allows a programmer to combine the elementary data types to create new data types known as "user-defined data types." There are four user-defined types commonly used in the definitions of Macintosh ToolBox and operating system routines:

1. SignedByte — occupies one byte with its contents stored in two's complement form. A SignedByte can therefore hold integers in the range $-128$ to $+127$.

2. Byte — occupies two bytes with the value stored in the low-order byte.

3. Ptr (a "pointer") — occupies four bytes and contains an address which indicates the starting location of a data structure. The Macintosh uses many different pointers; they can be identified by the presence of the characters \texttt{Ptr} in the data type name.

4. Handle — occupies four bytes and contains the address of a master pointer. (A Handle is a pointer to a Ptr.) As with pointers, the Macintosh uses many different handles. Handles have the characters \texttt{Handle} as part of their data type name.
While a number of ToolBox routines provide handles to data structures, it sometimes becomes necessary to use a pointer to that same data structure. In that case, an application must "de-reference" the handle. The code to do so appears as:

\texttt{MOVE.L \textit{SomeHandle},A0}  
\texttt{MOVE.L (A0),A0}

The first line loads the handle itself into an address register. The second line says: take whatever you find at the address specified by the contents of A0, and put it back in A0. This will place the pointer in A0, since the contents of a handle storage location is a pointer.

We'll look at other user-defined types as we need them to work with ToolBox and operating system calls.

---

**Pascal Data Structures**

Pascal data structures come in two varieties — arrays and records. Arrays can be built from any previously defined data type, though all values in an array must be of the same type. Records, as well, can be created from any previously defined data type, but different data types are permitted within the record; each item in a record is termed a field.

**Arrays**

The Pascal syntax:

\texttt{ArrayName = ARRAY [1..20] of INTEGER}

creates a new data type called \textit{ArrayName} that contains space for 20 values, each of which is an INTEGER. Therefore, the total length of this data structure is 20 words (40 bytes). To allocate space for it in an assembly language program, you might use:

\texttt{SymbolicAddress DCB.W 20,initial value}

where the 20 refers to the length of the array.

When an array is PACKED, the computer will store the data as efficiently as possible, without regard to how that storage might affect access. For example:

\texttt{NewArray = ARRAY [1..24] of BOOLEAN}
will require 24 words of storage, since each BOOLEAN occupies an entire word. To allocate space for it in an assembly language program, you must write:

```
SymbolicAddress  DCB.W  24,0
```

But:

```
NewArray = PACKED ARRAY [1..24] of BOOLEAN
```

will require only 24 bits (one and a half words), since the boolean values will be crammed one next to the other. Defining the packed array for assembly language use requires only:

```
SymbolicAddress  DCB.B  3,0
```

As a further example, consider the Pascal data type Str255, which is defined as:

```
Str255 = PACKED ARRAY [1..256] of CHAR
```

Instead of occupying one word per character as in the CHAR data type, each eight-bit ASCII code is packed in a single byte, and the entire string will occupy up to 256 bytes. (Don't forget that the first byte contains a number indicating how many characters there are in the string.) On the other hand:

```
StringArray = ARRAY [1..256] of CHAR
```

would occupy 512 bytes, since each non-packed character requires an entire word.

**Records**

In terms of dealing with ToolBox and operating system routines, you will encounter records more frequently than arrays. Records are commonly used to group information about various entities within the Macintosh. For example, whenever you create a menu, the Mac stores data about that menu in a menu record. That record is defined as:

```
MenuInfo = RECORD
    menuID :INTEGER;
    menuWIDTH :INTEGER;
    menuHeight :INTEGER;
    menuProc :Handle;
    enableFlags :PACKED ARRAY [0 ..31] of BOOLEAN;
    menuData :Str255;
END;
```
This definition creates a new data type called MenuInfo which represents a record consisting of six fields of data. Whenever you create a menu, the Macintosh will return a pointer to a pointer (the Menu Handle) that will tell you where this information is stored.

How much storage will this menu record use? You can determine the length of any record by adding up the length required by each of its fields. In the menu record, each of the first three fields requires one word, the Handle data type requires two words, the packed array is 32 bits long and therefore requires 2 words, and the string is up to 256 bytes (128 words) long. Therefore, each menu record will take up a maximum of 135 words, or 270 bytes.

The menu record is an example of a record that will be generated for you by the Macintosh; you gain access to it by the handle that is returned by the system when you create the menu. At times, though, you will need to define records within your programs so that you can either access fields within the records after the Macintosh creates them, or pass data to ToolBox and operating system routines in a record.

For example, every time an "event" happens to the system (an event could be a keypress, a click of the mouse, a disk insertion, or a signal from an I/O device, etc.), the Macintosh generates an event record, recording data about the event. An event record has the structure:

```
EventRecord = RECORD
    what :INTEGER;
    message :LONGINT;
    when :LONGINT;
    where :Point;
    modifiers :INTEGER;
END;
```

If you examine the contents of what, then you can determine what kind of event occurred. (What will contain a code identifying the type of event.) Where lets you know where the mouse pointer was when the event occurred. The data type Point (a user-defined data type) consists of two numbers which give the coordinates of the mouse pointer in a Cartesian coordinate system which is superimposed on the screen. (See Chapter 7 and the discussion of windows for more information.)

Since the Sample program in Chapter 3 is designed to respond to mouse and keyboard events, the program must set aside space for the event record at the end of the program code. (If the storage had been allocated with DS, the space would be in the applications globals area.) The definition appears as:

```
EventRecord
    What DC 0
    Message DC.L 0
    When DC.L 0
    Point DC.L 0
    Modify DC 0
```
Just as with the structure of TAPE.MASTER that was defined in Chapter 5, we can access the starting address of the structure by referencing the symbolic address `EventRecord`, or we can access a single field by using its individual symbolic address. For example, `What` will reference the address of the word that contains the code representing the type of event that the system recorded.

This works only because the assembler allocates storage in the order in which it encounters `DC` and `DS` directives. If the allocation directives for a record are placed physically one after the other in the source code, they will be allocated physically contiguous storage locations.

Interacting with the File Manager (the group of operating system routines that control file I/O) is probably the most complex task we must tackle when writing Macintosh assembly language programs, at least in terms of the associated data structures. File Manager routines require some data as input and will return additional data when the routines are finished, using extremely large records known as parameter blocks. An example of this usage appears in Chapter 4 in the discussion of the instruction `LEA`. (Complete discussion of the File Manager appears in Chapter 11.)

---

**Procedure and Function Calls**

Procedures and functions are two types of Pascal subprograms. When used in a Pascal program, the data used by these subprograms may be declared globally in the program's `var` block. In that case, the programmer has the option of merely letting the subprogram use whatever data it needs without bothering to explicitly transfer the data into and out of the subprogram. However, the ToolBox and operating system routines, all of which are defined as Pascal functions and procedures, cannot use global data because they are `external` to the program which calls them: that is, the code for the ToolBox and operating system routines is never a part of the source code of the application in which they are being used. Use of the Macintosh's built-in routine therefore requires careful attention to the process of moving data to and from procedures and functions.

The data passed to a subprogram are called `parameters`. Parameters that are only used as input to a subprogram are known as `value parameters`. Parameters that are modified within the subprogram and then passed back to the main program are called `variable parameters`. Each call to a procedure or function involves not only the name of the subprogram but a list of the parameters that will be passed in and out of the subprogram.

Procedures and functions differ primarily in how they return information to the main program. A procedure returns data only through variable parameters specified in the call's parameter list. A function, though, returns an additional result. This result might be a handle to a data structure or a boolean indicating whether or not the function successfully completed the required operation.
Access to the ToolBox and operating system routines is through either a Pascal procedure or function call. Macintosh technical documentation presents them in their Pascal syntax and generally leaves it up to the assembly language programmer to simulate the calling sequence.

The __DrawChar routine, which you have already seen, is written in Pascal as:

**PROCEDURE DrawChar (ch: CHAR);**

The parameter list (ch: CHAR) appears in parentheses after the name of the procedure (DrawChar). The ch is the variable name given to the parameter; CHAR refers to its data type. As an assembly language programmer, you will not necessarily be concerned with variable names, but with the data types, since they specify the size and format of the data you must prepare before calling the procedure. ch is a value parameter; it is used only as input to the procedure.

Parameter lists are not limited to a single parameter. For example:

**PROCEDURE insertMenu (menu: MenuHandle; beforeID: INTEGER)**

requires two parameters, the handle to a menu record and an integer indicating the relative position of this menu in the menu list (i.e., when this menu is placed in the menu bar, between which of the other menus should it be placed?). Parameter names are separated from their data types by colons. If more than one parameter has the same data type, the parameter names will be separated by commas. (See the discussion on BlockMove below for an example.) Parameters with different data types are separated by semicolons. Variable parameters are preceded by VAR; value parameters have nothing to distinguish them. Both of InsertMenu's parameters are therefore value parameters; they serve only as input to the procedure.

To call a ToolBox procedure from an assembly language program, you must first push the parameters, in order from left to right as they appear in the parameter list, onto the stack. Then you call the procedure. To draw a character using DrawChar for example:

```
MOVE $0040, -(SP)
```

first places the ASCII code of one character onto the stack. Because the Pascal data type is CHAR, an entire word is moved. Once the character is on the stack, then:

```
__DrawChar
```

initiates the call to the ToolBox routine. The procedure takes the parameters off the stack while it is executing, so that when it terminates, none remain on the stack.

ToolBox functions are handled in approximately the same way. The main difference is that before beginning to push the function's parameters onto the stack, a program must push an empty space for the function's result. The space for
the result is always deepest in the stack. When the function is finished, all of the parameters will have been removed from the stack; the result will be on top so that it can be easily recovered.

For example, the Toolbox function MenuSelect identifies which menu item received a click from the mouse. The Pascal definition of the function is:

\[
\text{FUNCTION MenuSelect (startPt: Point) : LongInt}
\]

The parameter \text{startPt} is the coordinates where the mouse was clicked. The data type \text{Point} refers to a user-defined data structure that is four bytes long and contains the coordinates of where the mouse was when the mouse button was clicked. The result of this function is the number of the menu item that was chosen. Since that data type of the result is \text{LongInt}, four bytes must be set aside to hold it. Therefore, the first step in the set-up sequence is to clear space on the stack for the result:

\[
\text{CLR.L} \quad -(SP)
\]

Then, the point can be moved onto the stack:

\[
\text{MOVE.L} \quad \text{Point} \quad \text{(Point comes from the event record described above)}
\]

Finally, all that remains is to call the function:

\[
\_\_\_\_\text{MenuItem}
\]

When a function finishes, you must always recover the result:

\[
\text{MOVE.L} \quad (SP) + , D0
\]

\textbf{NOTE:} Regardless of whether your program will use the result in any way, be sure to remove it from the stack. If the result is not removed, its presence will disrupt the operation of further procedure and function calls and will probably cause \text{RTS} instructions to fail in unexpected ways.

Probably the hardest thing about simulating the Pascal syntax for assembly language calls to Toolbox routines is deciding whether to put the parameter itself on the stack or to merely push a pointer to the parameter. Here are a few guidelines that should help:

1. Push pointers to variable parameters. For example, the procedure GlobalToLocal converts a point from the screen's coordinate system to the coordinate system of whatever window that point is within. In Pascal, the procedure is defined as:

\[
\text{PROCEDURE GlobalToLocal (VAR pt: Point)}
\]
The point will be passed into the procedure, converted, and then passed out. Since the procedure needs an address to store the converted coordinates, the address of the point is placed on the stack rather than the value of the point itself. Therefore, to call `GlobalToLocal`, you would:

```
PEA Point ___GlobalToLocal
```

2. Push pointers to records. In other words, when the data type of a parameter is a record or an array rather than a single value, only a pointer to the beginning of the data structure is necessary.

3. Push pointers to parameters that occupy more than 4 bytes of space.

4. Otherwise, move the parameter onto the stack using the `MOVE` instruction. When moving parameters, pay particular attention to the size of the parameter. Note that if you move a byte, the Macintosh will automatically push another unused byte onto the stack to keep the stack pointer on an even address.

Operating System procedures and functions are described with Pascal syntax just like ToolBox routines, but they do not get their parameters from the stack. Instead, Operating System routines take their parameters from registers. Operating System functions also return their results in registers. Unfortunately, the only way to know which parameters should be placed in which registers is to consult `Inside Macintosh`; merely examining the procedure or function definition will not give you that information. An example of the use of one Operating System routine follows.

### An Overview of the Toolbox and Operating System Routines

One of the things that makes the Macintosh both a pleasure and a pain to program in assembly language is the presence of so many prewritten routines. Most are in ROM, though some are present only on disk. They fall into two major groups: those known as the ToolBox and those that are part of the operating system. In either case, they are organized into “Managers,” each of which relates to one general function.

### The ToolBox

The ToolBox consists of 13 ROM managers and three sets of routines on disk:

1. The **Resource Manager** provides tools that manage resources. Resources are constructs such as windows and menus that an application will use. Most
applications will store resources in a file that is separate from the source code during the development process and will need to use at least the Resource Manager routine that opens the appropriate resource file.

2. **QuickDraw** contains all of Macintosh's graphics routines. Even applications that contain no graphics must make use of QuickDraw routines, since they control the location of all screen display operations and provide for the manipulation of text display characteristics.

3. The **Font Manager** is a small set of routines that are rarely accessed directly by a programmer. Instead, they are called by QuickDraw when a program requests font manipulations.

4. The **ToolBox Event Manager** contains routines that monitor things that happen to the system. Events (discussed in detail in Chapter 8) include occurrences such as a click of the mouse button, the insertion of a disk, or the press of a key on the keyboard. Interaction with the Event Manager forms the central control structure of any Macintosh application.

5. The **Window Manager** handles the definition, disposition, and manipulation of windows. Any application that adheres to the standard Macintosh user interface will make significant use of these routines.

6. The **Control Manager** does for controls what the Window Manager does for windows. Controls include scroll bars in windows and buttons (those hot-dog shaped balloons that appear in alert and dialog boxes). Control Manager routines may be called directly by a program or may be called by the Dialog Manager (see below).

7. The **Menu Manager** provides routines that create and manipulate menus. Most applications use the Menu Manager extensively.

8. **TextEdit** is a powerful set of routines that provide for the entry, display, and editing of text. Even a totally graphics-based application cannot avoid TextEdit, since some of the standard desk accessories (which all Macintosh applications should support) allow text editing.

9. The **Dialog Manager** allows an application to create, manipulate, dispose, and monitor events in dialog and alert boxes. Virtually every Macintosh program will use dialogs and alerts in some way.

10. The **Desk Manager** contains the routines that support desk accessories. They allow an application to invoke a specific desk accessory and to then turn management of that desk accessory over to the system.

11. The **Scrap Manager** provides the capability to transfer text and graphics between applications via the Clipboard. Whether or not an application will interact with the Scrap Manager is determined by the characteristics of the specific application.

12. The **ToolBox Utilities** are a diverse set of routines that cover some logical operations and bit manipulations.
13. The **Package Manager** is a gateway to the non-ROM ToolBox routines. The non-ROM routines are grouped into three packages which are loaded into RAM the first time they are called by an application. The packages handled by the Package Manager are:
   a. The **Binary-Decimal Conversion Package** converts ASCII strings of decimal characters into binary numbers that can then be used in arithmetic operations.
   b. The **International Utilities Package** contains a group of routines that make it possible to write non-English applications; also has some useful string comparison routines.
   c. The **Standard File Package** contains the standard dialog boxes that gather information about opening, closing, and saving files.

The Operating System Routines

Like the ToolBox, the operating system's routines are divided into managers. Eight are in ROM; two managers and three packages are on disk.

1. The **Memory Manager** handles the allocation of main memory while an application is running. Most Memory Manager routines affect the application heap.

2. The **Segment Loader** is the part of the operating system that actually loads a program into memory so it can be executed. For small applications, the Segment Loader is transparent to the programmer. It is invoked when a user double-clicks on a program icon. However, large applications that will not fit all at once into main memory can be broken up into chunks known as segments. In that case, the programmer must explicitly use Segment Loader routines to manage the swapping of segments between the disk and main memory.

3. The **Operating System Event Manager** contains the routines that actually detect hardware events such as mouse button and key presses. The events are passed directly to the ToolBox Event Manager, which can then be tapped by a programmer. An application rarely accesses the Operating System Event Manager directly.

4. The **File Manager** provides routines that create, open, close, read to, and write from files. They provide an unprecedented amount of flexibility in file I/O.

5. The **Device Manager**, like the File Manager, deals with I/O, but on the device rather than the file level. There are three device drivers in ROM:
   a. The **Disk Driver** (takes care of the disk drives)
   b. The **Sound Driver** (handles the Macintosh's speaker)
   c. The **Serial Drivers** (manages the two serial communications ports)
6. The **Vertical Retrace Manager** handles system actions which must be repeated at regular intervals while an application is running. These include incrementing the system clock, checking to see if the stack and heap have run into each other, and looking for hardware events such as a disk insertion or a change in the status of the mouse button. The only time an application will use the Vertical Retrace Manager is if it wishes to insert an activity of its own among those that the operating system is performing automatically.

7. The **System Error Handler** is that part of the operating system that provides the alert box with the little bomb in the upper left-hand corner. It is invoked whenever the system detects an error from which the system cannot recover, such as a binary instruction code which has no meaning to the 68000, or an address which is larger than the Macintosh's address range. This is another manager which is rarely tapped directly by an application program.

8. The **Operating System Utilities** are another miscellaneous set of "nifty" routines. They provide some string comparison (the string comparisons in the International Utilities Package are better), provide block move capabilities, and give access to the system's date and time.

9. The **Printing Manager** is not in ROM but rather is kept on disk. Along with the appropriate **Printer Driver**, the Macintosh can then support a theoretically infinite number of different printers. Any application that supports printing will make extensive use of the Printing Manager's routines.

10. The **AppleTalk Manager** is the Macintosh's gateway to the AppleTalk telecommunications network. It contains a number of disk-based routines to manage AppleTalk access as well as a pair of RAM-based device drivers.

11. The **Disk Initialization Package** is also on disk. It is called by the Standard File Package whenever a disk needs to be initialized. It is rarely called directly by an application program.

12. The **Floating-Point Arithmetic** and **Transcendental Functions Packages**, both of which are kept on disk, provide for arithmetic operations which cannot be handled within a single 32-bit register.

**A Couple of Things to Be Aware Of**

There is a conceptual problem with the way the ToolBox and Operating System routines are grouped. The Managers themselves tell you nothing about the sequence of calls necessary to perform a specific program action. For example, the routine that detects and identifies what sort of event has occurred is a part of the Event Manager. If the event was a mouse down event (the mouse button was clicked), then you must use a Window Manager routine to discover where the mouse button was pressed, even if it was pressed in the menu bar. Assuming that the mouse down event was in the menu bar, then Menu Manager routines can determine which menu and what item within that menu was selected.
Figuring out which routine to call when is one of the most baffling tasks in creating any Macintosh application, regardless of the language in which an application is written. Therefore, as you read on in this book, you will generally find descriptions of ToolBox and operating system calls grouped by function rather than by manager to aid you in understanding the sequencing of activities within an application.

Though the ToolBox and operating system routines are mostly in ROM, they are nonetheless programs. That means that they make use of the 68000's internal registers. If an application has placed information that must be retained in address and/or data registers, that information may be lost during a call to one of the Mac's routines. There are two ways to get around the problem.

The first is to put information that the application requires in some other form of storage by assigning it to storage locations defined by DC or DS directives. The second is to temporarily save the contents of the registers on the stack.

The instruction MOVEM (move multiple registers) simplifies the task of placing the contents of a series of registers on the stack. The general form of the instruction is:

```
MOVEM.L register list, -(SP)
```

To retrieve the contents of the registers:

```
MOVEM.L (SP)+, register list
```

The register list accepts either a series of individual registers separated by / or a range of registers indicated by a starting and ending register number. For example:

```
MOVEM.L D1/D2/A0 - A4, -(SP)
```

will place the contents of D1, D2, A0, A1, A2, A3, and A4 on the stack in that order.

When retrieving information stored on the stack, the register list must be in the same order as when the information was stored. The system will correctly pull the information from the stack and place it in the appropriate registers. To retrieve the information stored in the example above, use:

```
MOVEM.L (SP)+, D1/D2/A0 - A4
```

Be very aware of what is happening to the stack when attempting to use it for temporary storage of CPU registers. Consider the situation when it becomes necessary to save register contents before jumping or branching to a subroutine. The subroutine instruction pushes a return address onto the stack. That return address is "on top" of the register contents. Therefore, the application must not attempt to restore the contents of the registers until after the program has returned from the subroutine: that is, the return address must be pulled from the stack before the registers can be properly restored. If it is necessary to have the contents
of the registers within the subroutine, then the instruction to save them should occur after the jump or branch to subroutine instruction.

When should an application save the contents of its registers? There are two approaches you can take. The conservative approach says save all registers every time an application makes a call to a ToolBox or operating system routine. The second method is initially to not save any registers and then monitor program activity with the debugger to determine specifically which registers are altered and must therefore be saved. In general, the ToolBox and operating system routines use D0–D2 and A0–A4, though there are many exceptions.

---

**Calling Toolbox and Operating System Routines — The Trap Mechanism**

All the ToolBox and operating system routines you have seen so far are invoked in assembly language source code with a name that begins with an underbar. The Assembler translates those routine names into machine language instructions that the Macintosh can understand.

When assembled, all calls to ToolBox and operating system routines — except those of the Printing Manager — begin with $A, or %1010; the rest of the instruction word contains information that identifies the particular routine being called. The 68000 microprocessor has no instructions with codes that begin with %1010. Therefore, it "traps" those instructions. Under most circumstances, the microprocessor would return a system error indicating that it encountered an unrecognizable instruction. The Macintosh operating system, however, intercepts the microprocessor's detection of the trap. It interprets the trap as a reference to the ToolBox Dispatch Table discussed in Chapter 2. Because instructions that begin with %1010 are not part of the microprocessor's hardware instruction set, they are known as "unimplemented instructions" or "line 1010 unimplemented instructions." They allow a computer manufacturer to enhance the 68000 instruction set by adding custom instructions that are implemented in software.

Trap words are associated with names by using the assembler directive `.TRAP`. For example, the routine that draws a single character has a trap word of $A883. To give it a name, the following could be included in program code:

```
.TRAP __DrawChar  $A883
```

For the programmer's convenience, trap words for all ROM routines are assigned names in the file MacTraps.D (found on MDS2). It should be included at the beginning of each application developed on a 512K machine. There may not
be enough memory in a 128K machine to assemble a program that contains the entire MacTraps.D file. In that case, you will need to define explicitly any traps the program uses with the .TRAP directive.

Using Toolbox and Operating System Routines — Simplifying the Sort and Search

The straight-insertion sort and the binary search have one basic process in common — they compare strings. The sort also moves large blocks of code. It would simplify the code for these two utilities considerably if they could use prewritten routines to accomplish the comparison and move activities.

There are actually three different routines that do string comparisons. One is an Operating System routine — EqualString. The problem is that this function only returns a boolean value indicating whether the two strings being compared are equal or unequal. That is not enough information for either the sort or the search; both need to know direction (i.e., is the SearchString greater than or less than the string in the array?).

Tucked within the International Utilities Package are two string comparison functions. One is exactly like EqualString (IUIDString); but the other, IUMagString, returns the kind of result the sort and search require — a 0 if the strings are equal, a −1 if the first string is less than the second, and a +1 if the first string is greater than the second one. Depending on how you look at it, there is one drawback to using IUMagString: upper-case and lower-case letters are evaluated as different characters, with lower-case coming after upper-case. (EqualString and IUIDString ignore the upper- and lower-case distinction.)

IUMagString is specified as:

FUNCTION IUMagString (aPtr, bPtr: Ptr; aLen, bLen:INTEGER): INTEGER;

The first two parameters are of the same data type — Ptr. They are pointers to the start of the two strings which are to be compared. The third and fourth parameters are both integers — the number of bytes in each string. The result is an integer as described above.

In terms of the sort, one of the strings is contained in the data structure identified by NewRecord. The second is somewhere within TapeArray. We'll use the string in the array as the "a" string. Therefore, to find its starting address, we need to compute its offset from the beginning of the array, just as we did before. (Take the record number and multiply by 64, the length of a record.) If the offset is
in D6 (as it is after a call to Compute Offset) and the address of TapeArray is in A3, then a pointer to the start of the beginning of the "a" string is equal to:

```
ADDA D6,A3
```

The address of NewRecord goes into A2. We will need to compare 30 characters, since the tape name field is 30 characters long.

The set-up sequence therefore involves first pushing an empty word onto the stack to contain the result and then each of the parameters in order:

```
CLR.W – (SP) ; space for integer result
MOVE.L A3, – (SP) ; "a" pointer
MOVE.L A2, – (SP) ; "b" pointer
MOVE.W #30, – (SP) ; characters in "a" string
MOVE.W #30, – (SP) ; characters in "b" string
```

At this point it might seem that we're ready to call the function. Using:

```
—IUMagString
```

though, it will not work in this case. IUMagString is part of a package and therefore doesn't exist as a separate call. Instead, whenever you need a routine that is part of a package, first push a number that identifies the routine onto the stack and then call the package as a whole. The International Utilities Package is Package #6; IUMagString is routine #10. Therefore, to initiate IUMagString:

```
MOVE.W #10, – (SP)
—IPack6
```

(For further information on using Macintosh's packages, see Chapter 12).

The result of IUMagString is recovered by the instruction:

```
MOVE.W (SP)+,D0
```

Since a MOVE instruction sets the condition codes, the value of the result can be checked using one or more of the Bcc variations without any further manipulation.

To see how IUMagString simplifies the sort and search routines, take a look at Listings 6.1 (the sort) and 6.2 (the search).

A prewritten routine that moves blocks of main memory would simplify considerably one of the major tasks of the straight-insertion sort. BlockMove is an operating system procedure that does just that. It is defined as:

```
PROCEDURE BlockMove (sourcePtr, DestPtr: Ptr; byteCount: Size);
```
This definition alone does not contain enough information to call the routine. *Inside Macintosh*, though, indicates that:

1. A pointer to the starting location of the bytes to be moved (the source pointer) should be placed in A0;
2. A pointer to the starting location of where the bytes should be moved to (the destination pointer) should be placed in A1;
3. The total number of bytes to be moved should be placed in D0 and that the size of this operand is longinteger.

### Listing 6.1 Straight-Insertion Sort with Block Moves

```
MOVE   TotalRecords,D1
LEA    TapeArray(A5),A2
CMP    #0,D1
BEQ    InsertNew    ;if first record, insert immediately
SUBQ   #1,D1         ;otherwise, adjust for record #’s beginning with 0

Checking
JSR    ComputeAddress1 ;Address returned in A3
MOVE.L D1,-(SP)        ;save D1 on stack
CLR.W -(SP)             ;space for result
MOVE.L A3,-(SP)         ;pointer to record in array
PEA    NewRecord(A5)    ;pointer to new record
MOVE.W #30,-(SP)        ;characters to look at in first string
MOVE.W #30,-(SP)        ;characters to look at in second string
MOVE.W #10,-(SP)        ;ID for IUMagString
_Pack6

MOVE.W (SP)+,D0         ;recover result
MOVE.L (SP)+,D1         ;recover former contents of D1

CMP    #0,D0
BLE    JustBeforeInsert ;found place to insert record
BGT    MoveOld          ;move existing record down

MoveOld
MOVE   D1,D5            ;record # to move to
ADDQ   #1,D5
JSR    ComputeAddress1  ;offset returned in A3
JSR    ComputeAddress2  ;offset returned in A4

MOVE.L A3,A0            ;source pointer for block move
MOVE.L A4,A1            ;destination pointer for block move
MOVE.L #64,D0           ;64 bytes will be moved
_BlockMove

SUBQ   #1,D1            ;move back a record
CMP    #-1,D1           ;does new record go in first position?
BEQ    JustBeforeInsert
BRA    Checking
```

(continued)
JustBeforeInsert
   ADDQ #1,D1 ;insert just below where comparing

InsertNew
   MOVE D1,D5
   JSR ComputeAddress2
   LEA NewRecord(A5),A0 ;pointer to source (the new record)
   MOVE.L A4,A1 ;pointer to destination
   MOVE.L #64,D0 ;number of bytes to move
   _BlockMove ;move a record
   LEA TotalRecords,A0
   ADDQ #1,(A0) ;increment number of records

ComputeAddress1
   MOVE.L D1,D6 ;offset = record # * 64 bytes
   MULU #64,D6
   MOVE.L A2,A3
   ADDA.L D6,A3
   RTS

ComputeAddress2
   MOVE.L D5,D7
   MULU #64,D7
   MOVE.L A2,A4
   ADDA.L D7,A4
   RTS

Listing 6.2 Sequential Search with String Comparison Routine from the International Utilities Package

LEA TapeArray(A5),A2 ;start of tape array
MOVE TotalRecords,D1
SUBQ #1,D1 ;bottom pointer
MOVE D1,D3
SUBQ #1,D3 ;save last record-1 # for future reference
MOVE #Ø,D2 ;top pointer

MidPoint
   MOVE D2,D5 ;find middle record #
   ADD D1,D5
   DIVU #2,D5
   AND.L #$0000FFFF,D5 ;mask off remainder
   CMP #1,D5
   BLE TopRec ;handle first two records
   CMP D5,D3
   BLE BottomRec ;handle last two records
   JSR ComputeAddress2
   MOVEM.L D1-D5/A1-A2,-(SP) ;save registers

(continued)
Listing 6.2 (continued)

CLR.W -(SP) ;space for result
MOVE.L A4,-(SP) ;pointer to record in tape array
PEA NewRecord(AS) ;pointer to search string
MOVE.W #30,-(SP) ;number of characters to compare
MOVE.W #30,-(SP) ;number of characters to compare
MOVE.W #10,-(SP) ;number of characters to compare
PACK6 ;invoke the package
MOVE.W (SP)+,D0 ;recover result
MOVEM.L (SP)+,D1-D5/A1-A2 ;restore registers

CMP #0,D0 ;check result of string compare
BGT TopHalf ;array greater than search string
BLT BottomHalf ;array less than search string

LEA RecordCounter,A0
MOVE D5,(A0)
JSR DisplayOneRecord ;must be equal - record has been found
MOVE ReturnFlag(A5),D0
CMP #0,D0 ;which module called this routine?
BEQ KeepGoing ;call was from Select
RTS ;call was from Change or Delete

KeepGoing
JSR DisplayDialog3 ;display find & wait dialog box
JSR DisplayWindows ;clear text edit windows
RTS ;return to Select menu

BottomHalf
MOVE D5,D2 ;move top pointer down
BRA NoFindCheck

TopHalf
MOVE D5,D1 ;move bottom pointer up

NoFindCheck
CMP D2,D1 ;pointers have crossed
BMI NoFind ;find new middle record and go again

NoFind
JSR DisplayDialog1 ;displays "none found" dialog box
JSR DisplayWindows ;clear screen and text edit records
RTS ;return to Select menu

TopRec
MOVE #0,D5
JSR OneCheck
MOVE #1,D5
JSR OneCheck
BRA NoFind

BottomRec
MOVE D3,D5
JSR OneCheck
ADDQ #1,D3
MOVE D3,D5
JSR OneCheck
BRA NoFind

(continued)
When **BlockMove** terminates, a result code will be placed in D0, indicating whether or not an error occurred.

One situation in which the sort moves data is to move an existing record down in the array. Therefore, the source of the data to be moved is the current record and the destination is one record below it. This requires two addresses in **TapeArray** that are computed by the subroutines ComputeAddress1 and ComputeAddress2. A pointer to the current record is returned in A3, and a pointer to the record below is returned in A4. To set-up for **BlockMove**, then:

\[
\begin{align*}
\text{MOVE.L} &\quad A3,A0 \quad ;\text{pointer to current record} \\
\text{MOVE.L} &\quad A4,A1 \quad ;\text{pointer to record just below} \\
\text{MOVE.L} &\quad \#64,D0 \quad ;\text{record is 64 bytes long}
\end{align*}
\]
Operating System routines are called just like ToolBox routines:

__BlockMove

In many cases, a program will not bother to check the result of an operation such as a block move. Since the result is in a register, though, and not on the stack like the results of ToolBox functions, it can be safely ignored.

To see where BlockMove fits into the flow of the straight-insertion sort, see Listing 6.1.

---

Questions and Problems

1. Write an assembler directive that will set aside storage space in the applications globals area for each of the following Pascal data structures:

   a. TYPE Point = RECORD
      v: INTEGER;
      h: INTEGER
      END;

   b. TYPE Rect = RECORD
      top: INTEGER;
      left: INTEGER;
      bottom: INTEGER;
      right: INTEGER
      END;

   c. TYPE Rect = RECORD
      topLeft: Point;          {assume that data type Point as defined in a above}
      bottomRight: Point
      END;

   d. TYPE Region = RECORD
      rgnSize: INTEGER;       {assume the data type Rect as defined in b or c above}
      rgnBBox: Rect
      END;

   e. TYPE Cursor = RECORD
      data: ARRAY [0 ..15] of INTEGER;
      mask: ARRAY [0 ..15] of INTEGER;
      hotSpot: Point          {assume Point as in a above}
      END;
THE PASCAL CONNECTION TO THE TOOLBOX AND OPERATING SYSTEM ROUTINES 157

f. TYPE
   Style = INTEGER;
   FMInput = PACKED RECORD
      family: INTEGER;
      size: INTEGER;
      face: Style;
      needBits: BOOLEAN;
      device: INTEGER;
      number: Point; {assume Point as in a }
      denom: Point
   END;

g. TYPE ScrapStuff = RECORD
   ScrapSize: LONGINT;
   ScrapHandle: Handle;
   ScrapCount: INTEGER;
   ScrapState: INTEGER;
   ScrapName: StringPtr
END;

2. Listed below are some Pascal data type statements for data structures used
as ToolBox routine parameters. For each:

A. decide whether the parameter itself or a pointer to the parameter should
be placed on the stack and

B. write the assembly language statements that will place the parameter or
its pointer on the stack.

For this exercise only, assume that space has been allocated in the
applications globals area for the data structures and that each has the
symbolic address DataType.

Example: TYPE Pointer = Ptr; Answer: When used as a value parameter,
push the pointer itself: MOVE.L DataType(A5),(SP)+. When used as a
variable parameter, push the address of the pointer: PEA DataType(A5).

a. TYPE TEHandle = Handle; {used as a value parameter}
b. TYPE TEHandle = Handle; {used as a variable parameter}
c. TYPE Point = RECORD {used as a value parameter}
   v: INTEGER;
   h: INTEGER
END;
d. TYPE Point = RECORD {used as a variable parameter}
   v: INTEGER;
   h: INTEGER
END;
e. TYPE Rect = RECORD {used as a value parameter}
   topLeft: Point;
   bottomRight: Point
END;

f. TYPE Rect = RECORD {used as a variable parameter}
   topLeft: Point;
   bottomRight: Point
END;

g. TYPE Str03 = PACKED ARRAY [0 ..3] of CHAR;

h. TYPE Str255 = PACKED ARRAY [0 ..255] of CHAR;

3. Consider the program skeleton below:

   ::
   ::
   ::
   MOVEM.L D0/D1/A0 – A4,(SP) +
   JSR StartOfSubroutine
   ::
   ::
   {end of main program}

StartOfSubroutine

   ::
   ::
   {body of subroutine goes here}
   ::
   MOVEM.L – (SP), D0/D1/A0 – A4
   RTS

A. What problem can you see with the statements in this program skeleton? Hint: think about the order in which operands and addresses are placed on the stack.
B. What simple re-arrangement of the statements will solve the problem?
Chapter Objectives

1. To learn the steps necessary to create a Macintosh window
2. To understand the purpose of resource files and know how to prepare one for use by a Macintosh application
3. To explore the ToolBox routines that manipulate windows
4. To learn the steps necessary to create a Macintosh menu
5. To explore the ToolBox routines that manage the menu bar

As we discussed in Chapter 1, a major element in a successful Macintosh application is adherence to the standard Macintosh user interface. Two of the distinguishing characteristics of that interface are windows and pull-down menus.

Creating Windows

The ToolBox routines that manipulate windows are grouped together under the heading of the Window Manager. These routines provide facilities for not only creating windows, but for changing their size and position on the screen.
Before using any Window Manager routines, first initialize QuickDraw; the Window Manager relies on many QuickDraw routines. Then call the routine which initializes the Window Manager. The calling sequence is:

```
PEA - 4(A5)
    _lnitGraf ;initializes QuickDraw
    _lnitWindows ;initializes the Window Manager
```

Usually, this is done at the very beginning of a program, immediately after the statements that INCLUDE equates files. In fact, it is important to perform these and the other initialization routines that we will encounter in a specific order. Macintosh’s ROM routines are deeply interconnected and some of the initialization routines rely on others in order to function properly. Failure to initialize in the correct order will cause a system error when you attempt to execute your program.

There are two ways to define windows. The first is to place all of the window specifications within the application program itself. The second is to create a source file (defined below) which contains a template for the window and to access that template from within the application. In either case, a number of parameters must be present to completely describe the window. These include its boundaries (how big it should be), its type, its title, whether it is visible or invisible, whether it should have a GoAway box, and where it should be placed relative to other windows on the screen (e.g., in front or in back).

### Window Boundaries

Windows are specialized graphics ports (known as grafports) in which the Macintosh can draw. A grafport (the concept originates with QuickDraw, the set of graphics routines that underlie nearly everything Macintosh does) is basically an area in which the Macintosh can execute graphics procedures. Grafports can overlap and move from front to back on the screen, providing the basis for overlapping windows.

Grafports have many characteristics, but most important for working with windows is the coordinate system that defines them. Superimposed on the Macintosh screen is a coordinate grid. If we assume that the origin (0,0) is in the upper left-hand corner (just below the menu bar), then the screen is 512 pixels wide and 342 pixels tall. The term pixel is short for “picture element” and refers to one dot on the screen. The Macintosh screen coordinate system appears in Figure 7.1.

Windows are rectangles that have corners defined in that coordinate system. Note that this is not necessarily the only coordinate system that can be superimposed on the screen, but it is the one that is used when defining windows. The 512 x 342 coordinate grid is often referred to as the screen’s global coordinates.
The coordinate system imposed on the Macintosh's screen has a top left coordinate of 0,0 and a bottom right coordinate of 512, 342.

This coordinate system does not include the menu bar.

Though many coordinate systems could be imposed on the screen, this is the one that is used to define windows and to place graphics images.

Each pair of coordinates (a point) refers to one pixel.

![Figure 7.1 The Macintosh Screen's Coordinate System](image)

The rectangles that define window boundaries are contained in the user-defined data type Rect. In the Pascal syntax:

```
TYPE Rect = RECORD CASE INTEGER OF
  0: (top: INTEGER;
      left: INTEGER;
      bottom: INTEGER;
      right: INTEGER);
  1: (topLeft: Point;
      botRight: Point)
END;
```

What this means is that there are two choices for defining the corners of a rectangle, though for all intents and purposes, they work out the same. You can either provide four separate positions that indicate the top, left, bottom, and right positions of the rectangle; or provide two points, one for the top left corner of the rectangle and the other for the bottom right. A point is another user-defined data type that puts together an X and Y coordinate to locate a specific pixel.
As an example, consider the window used in the Sample program. Its top left corner is at 40,20 and its bottom right corner at 300,350. Figure 7.2a shows those coordinates. If a window is defined within an application program (rather than in a resource file), then the rectangle which describes the window boundaries is usually assigned to a symbolic address. In the Sample program, the "boundary rectangle" is:

**BoundsRect 40,20,300,350**

Any use of the symbolic address **BoundsRect** will refer to all four integers. The coordinates are expressed in the screen's global coordinate system. These are the window's initial coordinates, which will change if the window is sized.

![Figure 7.2(a) Using Global Coordinates to Define a Window](image)

Windows have a second coordinate system called a *local* coordinate system. In a local coordinate system the point 0,0 is assigned to the upper left-hand corner of a window, regardless of the size of the window or where it is currently placed on the screen. For example, if window has global screen coordinates of 40, 20, 300, 350, the top left point of 40, 20 is translated to 0,0 for the window's local coordinate system.

The bottom local coordinate for a window is equal to the bottom global coordinate minus the top global coordinate plus 1 (e.g., 300 – 40 + 1 = 261). The right local coordinate is computed in a similar way; subtract the left global coordinate from the right global coordinate and add 1 (e.g., 350 – 20 + 1 = 331). The boundaries of this window's initial local coordinate system are therefore 0, 0, 261, 331, as shown in Figure 7.2b.
The bottom right local coordinates of a window will change as the window is sized. Though the top left local coordinates will remain at 0, 0, the bottom right coordinates will increase and decrease with the size of the window.

![Figure 7.2(b) A Window's Local Coordinate System](image)

**Window Types**

The Macintosh provides six pre-defined window types. These will be adequate for the majority of applications. Each type has an identifying number (see Table 7.1). If ToolEqu.D is INCLUDED in your source code, you can use the symbolic address assigned to the number rather than using the number itself.

<table>
<thead>
<tr>
<th>Symbolic Address</th>
<th>ID #</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentProc</td>
<td>0</td>
<td>Standard document window</td>
</tr>
<tr>
<td>dBoxProc</td>
<td>1</td>
<td>Alert or modal dialog box (heavy inner border)</td>
</tr>
<tr>
<td>plainDBox</td>
<td>2</td>
<td>Plain window with single outline border</td>
</tr>
<tr>
<td>altDBoxProc</td>
<td>3</td>
<td>Plain window with a shadow on the right and bottom</td>
</tr>
<tr>
<td>noGrowDocProc</td>
<td>4</td>
<td>Standard document window that cannot contain grow icon</td>
</tr>
<tr>
<td>rDocProc</td>
<td>16</td>
<td>Round cornered window for desk accessories</td>
</tr>
</tbody>
</table>

*Table 7.1 Pre-defined Window Types and Their ID Numbers*
The window type **documentProc** is a standard document window (see Figure 7.3a). It has a title bar, square corners, and may contain a size box and scroll bars. **noGrowDocProc** (Figure 7.3b) is the same as a **documentProc** box but cannot contain a size box and scroll bars.

![Figure 7.3(a) Standard Document Window with Scroll Bars](image)

![Figure 7.3(b) Standard Document Window without Grow Icon](image)
plainDBox (Figure 7.3c) is simply a rectangle with a solid border. It has no title or scroll bars. If you use altDBoxProc (Figure 7.3d), you'll get a plain box with a shadow along the right and bottom borders. dBoxProc (Figure 7.3e) will produce a plain window with an inner border. This type of window is generally used as an alert box.
This is an alert or dialog box

Figure 7.3(e) Alert or Dialog Box

**rDocProc** (Figure 7.3f) is a round-cornered window. It has a title, but no scroll bars. It is most often used to hold desk accessories and therefore will generally not appear in an application program unless that program is defining its own desk accessories.

This is a round-cornered window

Figure 7.3(f) Round-cornered Window
The Window Record

Information about the windows an application uses are kept in window records, one for each window. The structure of a window record is as follows:

\[
\text{WindowRecord} = \text{RECORD} \\
\text{port:} \quad \text{GrafPort;} \quad \text{the window's grafport} \\
\text{windowKind:} \quad \text{INTEGER;} \quad \text{the window's type} \\
\text{visible:} \quad \text{BOOLEAN;} \quad \text{TRUE if visible} \\
\text{hilited:} \quad \text{BOOLEAN;} \quad \text{TRUE if highlighted} \\
\text{goAwayFlag:} \quad \text{BOOLEAN;} \quad \text{TRUE if goAway region} \\
\text{spareFlag:} \quad \text{BOOLEAN;} \quad \text{currently unused} \\
\text{strucRgn:} \quad \text{RgnHandle;} \quad \text{structure region} \\
\text{contRgn:} \quad \text{RgnHandle;} \quad \text{content region} \\
\text{updateRgn:} \quad \text{RgnHandle;} \quad \text{update region} \\
\text{windowDefProc:} \quad \text{Handle;} \quad \text{window definition function} \\
\text{dataHandle:} \quad \text{Handle;} \quad \text{used by windowDefProc} \\
\text{titleHandle:} \quad \text{StringHandle;} \quad \text{window's title} \\
\text{titleWidth:} \quad \text{INTEGER;} \quad \text{width of title in pixels} \\
\text{controllist:} \quad \text{Handle;} \quad \text{handle to first control} \\
\text{nextWindow:} \quad \text{WindowPeek;} \quad \text{next window in list} \\
\text{windowPic:} \quad \text{PicHandle;} \quad \text{pic. for drawing window} \\
\text{refCon:} \quad \text{LONGINT;} \quad \text{reference value} \\
\text{END;}\
\]

An application can ignore many of the fields in a window record, but some do require further mention. In particular, an application may need to get to the three "region" parameters: the structure, content, and update regions. The term region comes from QuickDraw. It refers to some area that can be bounded by a rectangle but is not necessarily rectangular in shape. In other words, a region can be described by the rectangle that most closely encloses its contents. A region is defined by a simple record:

\[
\text{Region} = \text{RECORD} \\
\text{rgnSize:} \quad \text{INTEGER;} \\
\text{rgnBox:} \quad \text{Rect} \\
\text{END;}\
\]

rgnSize contains the number of bytes in the region. rgnBox is the rectangle that encloses it.

Windows have three regions. The structure region includes the window's outside outline and its title bar, if it has one. The content region is everything inside the window, including scroll bars. The update region contains those parts of a window that have been changed by the actions of an application and therefore need to be redrawn. All three regions can change while an application is executing.
It may be necessary for an application to retrieve the rectangles that describe any of these three regions. To do so, the application must:

1. Get the pointer to the window record.
2. Use an offset into the window record to retrieve the region's handle. Offsets into a window record are defined in the ToolBox equates file.
3. De-reference the handle to get a pointer to the region record.
4. Add 2 to the pointer to the region record to skip over the region size parameter. The result will be the starting address of the region rectangle.

As an example, let's look at finding the structure rectangle for a window:

MOVE.L WindowPtr, A0 ;get pointer to window record
MOVE.L strucRgn(A0), A0 ;get handle to structure region
MOVE.L (A0), A0 ;get pointer to region record
ADDA #2, A0 ;adjust address to skip over ;region size

Other parameters from the window record that an application might need will be discussed with the program activities that require them.

Defining Windows within an Application Program

The Window Manager routine NewWindow will set up and draw a window whose parameters are specified wholly within the application program. In Pascal, the routine appears as:

FUNCTION NewWindow (wStorage: Ptr; boundsRect: Rect; title: Str255; visible: BOOLEAN; procID: INTEGER; behind: WindowPtr; goAwayFlag: BOOLEAN; refCon: LongInt) : WindowPtr;

Note first of all that NewWindow is a function; it returns something called WindowPtr (the window pointer). The window pointer is the address of the location in the applications globals area of the window record. Many Window Manager routines need this window pointer as a parameter so they can operate on the correct window.

Since a window pointer contains an address, it will require a longword (4 bytes) of space. Therefore, the first step before calling NewWindow is to reserve space on the stack for the WindowPtr result:

CLR.L – (SP)

Then all the remaining parameters must be placed, in order, on the stack.
wStorage refers to a pointer to where the window record will be stored. It must reserve enough space for the entire window record. Therefore, wStorage should be defined as:

\[
\begin{array}{ccc}
\text{wStorage} & \text{DCB.W} & \text{windowsize},0 \\
\text{wStorage} & \text{DS} & \text{windowsize}
\end{array}
\]

where windowsize is defined in the ToolEqu.D file as the number of words in a window record. As long as ToolEqu.D has been INCLUDED in your source code, it isn't necessary to know the actual size of the window record. Using symbolic addresses rather than actual quantities is always preferable. For example, if the size of a window record changes at some later date, you will only need to use the updated equates file rather than changing your application program.

Since wStorage is a pointer, push it onto the stack using MOVE.L. Whenever a parameter is 4 bytes or less in length, put the parameter itself on the stack.

It is possible to allocate space for the window record on the application heap rather than in a program's code (using DC) or the applications globals area (using DS). To do so, use a value of 0 for wStorage.

boundsRect is the coordinates of the boundaries of the window's rectangle. As discussed above, the boundary rectangle should be assigned to a symbolic address. That address is placed on the stack with PEA, since the coordinates themselves occupy 8 bytes and are therefore too long to be placed on the stack themselves.

The title of the window can be simply included as a string in quotes. However, the string itself is not pushed on the stack. Like the boundary rectangle, it occupies more than 4 bytes.

PEA  'Text of the Title'

will push a pointer to the string Text of the Title onto the stack and place the string at the end of the program code.

visible is a boolean that indicates whether the window should initially be visible or invisible. If it has a value of TRUE, the window will be visible; otherwise, the window will be defined by NewWindow but not drawn. A boolean occupies a word of space. Therefore, to create a visible window, you would:

\[
\text{ST} \quad -(\text{SP})
\]

Though ST only affects one byte, the system will automatically push an unused byte onto the stack to keep the contents of the stack pointer even.

The procID is one of the six pre-defined window types mentioned above. Since procID is an integer, simply MOVE the appropriate constant onto the stack. For example:

\[
\text{MOVE} \quad \#\text{documentProc}, -(\text{SP})
\]

will indicate that this window should be a standard document window.
behind indicates where this new window should be placed relative to other windows on the screen. If behind contains a pointer to the window record of another window, the new window will be placed directly behind that window. On the other hand, if behind is 0, the new window will be placed behind all the other windows. A value of −1 for behind will place the new window in front. Since behind is a pointer, it requires a longword of space:

MOVE.L #−1,−(SP)

The goAwayFlag is a boolean that determines whether or not a GoAway box will appear in the title bar of the window. A value of TRUE draws a GoAway box; FALSE leaves it out.

The final parameter, refCon, sets up space for the window's reference value. A reference value is anything a programmer wishes to assign. It can be used in any way an application desires. In most cases, an application will rarely use it and should therefore give it a value of 0.

The complete NewWindow calling sequence appears as:

CLR.L −(SP) ;space for window pointer result
PEA wStorage ;pointer to storage for window record
PEA boundsRect ;coordinates of window corners
PEA 'Text of Title' ;title of the window
ST −(SP) ;visible window
MOVE documentProc ;window's resource ID
MOVE.L #−1,−(SP) ;window goes in front of all others
SF −(SP) ;no GoAway box
CLR.L −(SP) ;room for reference value
__NewWindow ;calls the routine

When NewWindow finishes, the window pointer will be left on top of the stack. Since the window pointer is essential to so many other Window Manager routines, it is vital that a program retrieve that window pointer before doing anything else. Space for the window pointer should be prepared by defining:

WindowPointer DC.L 0

or

WindowPointer DS.L 1

Then, immediately after defining the window, the pointer can be moved to WindowPointer with:

LEA WindowPointer,A0
MOVE.L (SP)+,(A0)
Using Resource Files to Create Windows

Using NewWindow is really the hard way to create a window. It is far more efficient to place the window definition parameters in a resource file which an application program can then tap. Changes to parameters can then be made in the resource file without requiring modification of the source code.

A resource file is a text file that has been compiled by the Resource Compiler, RMaker. It may contain not only window definitions, but definitions for things like menus and dialog boxes. To create a resource file, enter the Editor and type the resource definitions. Resource source files should be named with an extension of .R. (For example, the resource source file for the video tape index is called Tapes.R.)

The format of a resource file is very rigid. The first line contains the name of the file to which RMaker should write the compiled file. While the Video Tape Index program was being developed, the first line of its resource file read:

tape.index: Tapes.Rsrc

For each resource you wish to define, first identify the type of resource to which the definition applies. For example, to define a window:

TYPE WIND

The word TYPE is a signal that a new resource definition is beginning. WIND refers to one of 12 predefined resource types — in particular, a window.

The remainder of a window definition might appear as:

,1
A Sample Window
40 20 300 350
Visible GoAway
0
0

The second line contains a space, followed by a comma and then a sequence number for the window. Since it is possible to have many window definitions in the same resource file, each must be assigned a unique sequence number. By referring to that sequence number, the Macintosh can access the window definitions in any order. The space preceding the comma is required.

The text of the window title appears directly below the sequence number. It should not be in quotes. Even if you are defining a window type that doesn't have a title, it is useful to include one anyway simply for documentation.

The coordinates of the window's boundary rectangle follow immediately on the next line, separated by spaces. Their order is top, left, bottom, right.
The fifth line of a window definition indicates whether the window is visible or invisible and whether or not it should have a GoAway box. Use the appropriate word (Visible, Invisible, GoAway, noGoAway), though only the first character is actually used by the Macintosh.

A resource file will not accept the symbolic addresses assigned to window resource ID's in the ToolEqu.D file. Therefore, on the sixth line of a window definition you must use the numeric values to indicate what type of window should be drawn. The 0 in the example above refers to a standard document window (documentProc).

The final line of the window definition contains the window's reference value. If no reference value is needed, use 0 as a placeholder.

Once a resource file has been created by the Editor, it must be compiled using RMaker. Enter RMaker either by transferring to it from the Editor or by double-clicking on its icon from the Finder. Once you "open" a resource source file, the compilation proceeds automatically. A successful compilation produces a binary file with the name specified on the first line of the resource file's source code (e.g., the original compiled resource file for the video tape index was Tapes.Rsrc).

Before an application program can use the information in a separate resource file, that file must be opened. Therefore, immediately after initializing all the managers, open the resource file the program will be using. The routine that does so, OpenResFile, is part of the Resource Manager. The calling sequence for OpenResFile is:

```assembly
FUNCTION OpenResFile (fileName: Str255) : INTEGER;

OpenResFile returns an integer which contains a reference number for the file. It is rarely used. Nevertheless, since the reference number is left on the stack, you must be sure to remove it after calling the routine, since an extra parameter left on the stack will disrupt stack operations.

The sequence to open the separate resource file for the video tape index appears as:

CLR -(SP) ;space for result
PEA 'Tape.index:Tapes.Rsrc' ;name of resource file
__OpenResFile
MOVE (SP)+,D0 ;discard unused result

Once a window has been defined in a resource file, creating it from an applications program is very straightforward. The routine to use is GetNewWindow:
FUNCTION GetNewWindow (windowID: INTEGER; wStorage: Ptr;
behind: WindowPtr) : WindowPtr;

windowID refers to the sequence number assigned to the window definition in the resource file. The other parameters are exactly the same as those for NewWindow: wStorage is a pointer to where the window record will be stored, behind determines the window’s placement on the screen, and WindowPtr is the window pointer result.

To create the window defined by the sample window definition above (assuming it has a windowID of 1), you would code:

CLR.L - (SP) ;space for window pointer result
MOVE #1, - (SP) ;window ID
PEA wStorage ;pointer to storage for window record
MOVE.L # -1, - (SP) ;put this window in front
__GetNewWindow
LEA WindowPointer, A0 ;get address for window pointer
MOVE.L (SP) +, A1 ;retrieve window pointer from stack
MOVE.L A1, (A0) ;store window pointer

The video tape index program uses seven different windows. The portion of Tapes.R that contains the window definitions appears in Listing 7.1. Note that TYPE WIND is not repeated. Once RMaker has encountered a single TYPE statement, it assumes that all resource definitions that follow are of the same type until another TYPE appears.

Each window has its unique sequence number. While sequence numbers may not repeat within the same type of resource, they may be duplicated within another type (e.g., the eight menus that the program uses are numbered 1–8 even though the windows are numbered 1–7).

The main window is a standard document window (see Figure 7.4) with the title Video Tape Index. It acts more or less like a placemat for the remaining windows, which hold text as it is entered or displayed. Windows 2–6 are plain document windows (the window resource ID is 2). Though these windows have no titles when drawn, the resource file contains titles so the windows can be easily identified. The seventh window is another standard document window.

How do you figure out the coordinates for the window boundaries? Unfortunately, there is no easy way. Trial and error generally works best. The boundaries of the video tape index text windows changed six or seven times before they were properly placed. Making such changes with the definitions in a resource file is quick and easy; doing it with window definitions in an application is tedious and time consuming.
Listing 7.1 Resource Templates for Video Tape Index Windows

```
TYPE WIND
,1 Video Tape Index
40 10 300 500
visible NoGoAway
ØØ

,2 Tape Name
50 240 70 490
visible NoGoAway
2Ø

,3 Producer
75 240 95 415
visible NoGoAway
2Ø

,4 Date
100 240 120 283
visible NoGoAway
2Ø

,5 Rating
125 240 145 269
visible NoGoAway
2Ø

,6 Tape Number
150 240 170 276
visible NoGoAway
2Ø

,7 Annotation
205 20 280 490
visible NoGoAway
ØØ

;; window templates follow
;; sequence number
;; title
;; boundary rectangle
;; visible but no GoAway Box
;; window type (documentProc)
;; reference value

;; sequence number
;; title for documentation only

;; window type (plainDBox)
```
Programming Technique — Making a Resource File Part of Program Code

While an application is being developed it is convenient to keep the resource file separate from the program code; such an arrangement facilitates changes in the resource definitions. Once an application is completely debugged and its resource definitions no longer changing, the resources can be linked into the application itself. To make resource definitions parts of an application, do the following:
1. Rename the RMaker output file so that it has a .REL extension (e.g., the video tape index resource file Tapes.Rsrc was recompiled as TapesRs. REL).

2. Remove the call to OpenResFile from the application’s source code and reassemble the application.

3. Add the following to the application’s Linker control file after the names of all program modules:

   /Resources
   ResourceFileName. REL

   When modified to include its resource file in program code, the video tape index’s Linker control file appears as:

   Tapes.REL ; assembled version of program code
   PrLink.REL ; needed to do printing (see Chapter 10)

   /Resources
   TapesRs. REL ; compiled version of resource file

   $ 

4. Re-link the application

   Once the resource file is linked to the program code, the separate resource file no longer needs to be present on the same disk as the application. Note also that this procedure significantly lengthens the linking process and therefore should really be the last step in preparing an application.

---

**Manipulating Windows**

If you have run the videotape index program, you will have noticed that as you select an option from the main Options menu, the title of the main, background window changes to match the option selected. The text windows — hidden when the program begins — appear. Whenever you select Quit from within one of the program functions, the text windows disappear and the main window’s title reverts to Video Tape Index. These functions are accomplished with a few of the many routines that permit the manipulation of windows once they have been created.

---

**Changing a Window’s Title**

Changing a window’s title is accomplished with the SetWTite routine:

PROCEDURE SetWTite (theWindow: WindowPtr; title: Str255);
To use it, move the appropriate window pointer to the stack and then push a pointer to a string for the title. For example, changing the video tape index's main window's title from Video Tape Index to Enter New Titles and Annotations requires:

```
MOVE.L MainWindowPtr, -(SP)
PEA 'Enter New Titles and Annotations'
__SetWTitle
```

### Making Windows Appear and Disappear

It is possible, at any time, to make any window visible or invisible. This does not change the position of the windows relative to one another; it merely affects whether or not they can be seen.

To make a previously invisible window visible, use:

```
PROCEDURE ShowWindow (theWindow: Ptr);
```

Move the window pointer onto the stack and then call the routine. For example:

```
MOVE.L SomeWindowPtr, -(SP)
__ShowWindow
```

Using ShowWindow on a window that is already visible will have no effect.

The routine to make a previously visible window invisible is HideWindow:

```
PROCEDURE HideWindow (theWindow: Ptr);
```

### Changing a Window's Position in the Plane

How much of a window is visible also depends on which other windows are in front of it. Two routines, BringToFront and SendBehind directly affect window position.

BringToFront will make the window in the procedure call the front-most window on the screen:

```
PROCEDURE BringToFront (theWindow: Ptr);
```

SendBehind can place a particular window behind all other windows or behind any other window on the desktop:

```
PROCEDURE SendBehind (theWindow: Ptr;
behindWindow: Ptr);
```
The parameter `theWindow` is a pointer to the window that should be moved. `behindWindow` is a pointer to the window behind which `theWindow` should be placed. If `behindWindow` is 0, then `theWindow` will be sent to the very back.

`BringToFront` and `SendBehind` do not, however, make a window active. As stated in Chapter 1, regardless of how many windows occupy the screen at any given time, only one can be active. An active window is highlighted, though the specifics of the highlighting depend on the type of window. For example, for standard document windows, highlighting means that the title will appear in the title bar surrounded by horizontal lines. When a standard document window is inactive, the title bar still contains the title but the horizontal lines disappear. Drawing can only occur in active windows.

The routine `SelectWindow` is the best way to activate a window:

**PROCEDURE SelectWindow (theWindow: WindowPtr);**

A call to `SelectWindow` will do the following:

1. Unhighlight whatever window was most recently active;
2. Bring the window being activated to the front (i.e., does the same thing as `BringToFront`);
3. Highlight the window; and
4. Let the program know that one window is deactivated and another activated (this generates two “events,” which are discussed in Chapter 8).

Whenever possible, it is better to use `SelectWindow` rather than `BringToFront`. You should also not use `SendBehind` to deactivate a window, since using `SelectWindow` takes care of it for you.

The Video Tape Index program uses repeated calls to `SelectWindow` to manage its windows. If the main window is brought to the front by `SelectWindow`, it effectively hides the text entry windows since it is so much bigger. Therefore, each time the program returns from a subroutine that performs one of the `Options`, it executes:

```
MOVE.L MainWindowPtr, -(SP)
    _SelectWindow
```

Selecting each of the text entry windows in turn brings them in front of the main window. The actions which follow involve set-up for text entry and will therefore be discussed in detail in Chapter 9.

**Preparing Windows That Will Change Size**

If an application needs to give the user the ability to size a window, that window should contain a grow icon (two overlapping squares). In document windows, the
grow icon always appears in the lower right-hand corner of a window. A grow icon is displayed by the routine `DrawGrowlcon`:

**PROCEDURE DrawGrowlcon (theWindow: WindowPtr);**

If the window indicated by `theWindow` (a pointer to the appropriate window record) is active, `DrawGrowlcon` will draw the outline of the grow icon area, the icon itself, and the outline of the area that should contain scroll bars for that window. If the window is inactive, only the grow icon area and the scroll bar areas will be drawn.

For details on how to use the grow icon to size windows, see the section in Chapter 8 on handling mouse down events in grow regions.

### Setting Up Scroll Bars

One of the things that the Macintosh does very well is scrolling through large documents. The scroll bars that provide that facility are grouped with buttons and check boxes under the heading of controls. Controls are graphics images that allow the user to control program action in some way.

Most controls, like buttons and check boxes, only appear in dialog and alert boxes. They are handled by Dialog Manager routines (see Chapter 9). Generally, the only controls an application will deal with directly are scroll bars.

Information about a control is stored in a control record that is located by a handle:

```
ControlRecord = RECORD
    nextControl: ControlRecord;
    contrlOwner: WindowPtr;
    contrlRect: Rect;
    contrlVis: BOOLEAN;
    contrlHilite: BOOLEAN;
    contrlValue: INTEGER;
    contrlMin: INTEGER;
    contrlMax: INTEGER;
    contrlDefProc: Handle;
    contrlData: Handle;
    contrlAction: ProcPtr;
    contrlRefCon: LONGINT;
    contrlTitle: Str255;
END;
```

While an application will not need to retrieve data from most of these fields, there are two that are of some importance. Like windows, controls can be assigned arbitrary reference values (`contrlRefCon`) by an application. Since it may be necessary to identify what type of control a control record describes, the reference value can be used to hold that information. For example, some of the sample code
in Chapter 8 must distinguish between vertical and horizontal scroll bars. Therefore, each was assigned a unique reference value. Remember that reference values are assigned arbitrarily by an application and have no meaning to the system other than what an application gives them.

The first parameter in the record, nextControl, is also of some importance. Controls belong to windows. The handle to the control record of a window's first control will be stored in the wControlList parameter of the window record. The rest of a window's controls are linked together in a chain through the nextControl field of the control record. In other words, the handle to the next control in the list is found in the nextControl field. A window's last control will have a nextControl value of 0. A window without any controls will have a wControlList value of 0. This type of organization is known as a linked list. An application can find all of a window's controls by threading its way down the list, from one nextControl field to the next.

Scroll bars, like windows, can be defined either within an application or from a template in a resource file. Using a resource file is the simpler of the two procedures.

The scroll bars in the program that created Figure 7.3b were defined with the following entries in a resource file:

```
TYPE CNTL
    ,1 ;unique resource ID#
    horizontal ;title for documentation only
    245 0 261 316 ;boundary rectangle
    Visible ;visible or invisible?
    16 ;procedure ID that stands for scroll bar
    0 ;application-defined reference value
    0 100 1 ;minimum maximum value
    ,2 ;unique resource ID#
    vertical ;title for documentation only
    0 316 245 331 ;boundary rectangle
    Visible ;visible or invisible?
    16 ;procedure ID that stands for scroll bar
    0 ;application-defined reference value
    0 100 1 ;minimum maximum value
```

Control templates have a type of CNTL. As with windows, each control in the resource file must be assigned a unique number, its resource ID, which is the first line in the template. The space preceding the comma is required.

The third line may contain the title of the control. Since scroll bars do not have titles, that line is ignored by the system and can therefore be used just as documentation to identify the control. The third line in the control template is the boundary rectangle that defines where the control should be drawn. Coordinates are expressed in the local coordinate system of the window in which the control will appear. For scroll bars, the boundary rectangle should be 16 pixels wide. A
horizontal scroll bar will begin at the right edge of the window and end 15 pixels before its left edge, leaving room for the grow icon (a 15 x 15 pixel square). Vertical scroll bars will begin at the top of the window and end 15 pixels above the bottom, again to leave room for the grow icon.

As discussed earlier in this chapter, the window in Figure 7.3b has global coordinates of 40, 20, 300, 350. It is therefore 261 pixels high (top − bottom + 1) and 331 pixels wide (right − left + 1), giving it local coordinates of 0, 0, 261, 331. These latter coordinates were used to determine the boundary rectangles of the scroll bars. For example, the horizontal scroll bar has a top coordinate of 245 (261 − 16) to accommodate the width of the scroll bar, a left coordinate of 0 so the scroll bar will begin at the right edge of the window, a bottom coordinate of 261, and a left coordinate of 316 (331 − 15) to accommodate the grow icon.

Line four in the control template indicates whether the control is initially visible or invisible. Visible controls will be drawn when the control is created. The fifth line indicates what type of control the definition is for. Scroll bars have a procedure ID of 16. As with windows, the procedure ID's must be used as integers; the symbolic addresses assigned to them in the ToolBox equates file cannot be substituted.

Line six contains the optional reference value. This longinteger can be assigned any value in the resource file and accessed and changed while the application is running. If you will not use a reference value, simply assign it a value of 0.

The three parameters in the final line of the control template are the minimum value the control can take, the maximum value the control can take, and its initial value. The initial value for scroll bars should always be 1; this will ensure that the scroll bars are drawn and active when the control is created. A scroll bar is an analog scale; each movement within it represents movement of a certain percentage of a document. Therefore, its minimum value should be set to 0 or 1. The maximum value is rather arbitrary, but the larger the maximum value, the greater the sensitivity of the scale. For example, if a scroll bar has a range of 0 to 10, then the document will have, in effect, 10 positions to which it can be scrolled, each presenting a move of 10% through the document. On the other hand, a maximum value of 100 divides the scale into 100 pieces, permitting far smaller movements within the document.

Once a control template has been defined in a resource file and the resource file successfully compiled with RMaker, the control is created by GetNewControl:

FUNCTION GetNewControl (controlID: INTEGER; theWindow: WindowPtr) : ControlHandle;

GetNewControl returns a longinteger result which is a handle to the control record. All the other routines which affect controls need the handle to locate the record. Therefore, the control handle must be saved after it is pulled from the stack.

The parameter controlID is the resource ID number from the first line of the control template in the resource file. The second parameter is a pointer to the window in which the control will be drawn.
The program that drew Figure 7.3b created scroll bars with the following code:

```assembly
CLR.L - (SP) ; space for control handle result
MOVE #1, - (SP) ; the horizontal scroll bar
MOVE.L WindowPtr, - (SP) ; window pointer
__GetNewControl
LEA BottomControlHandle,A0
MOVE.L (SP) +,(A0) ; retrieve handle
CLR.L - (SP) ; space for control handle result
MOVE #2, - (SP) ; the vertical scroll bar
MOVE.L WindowPtr, - (SP) ; window pointer
__GetNewControl
LEA SideControlHandle,A0
MOVE.L (SP) +,(A0)
```

The above sequence will only display the control bars. It does not take care of moving them or moving the text in the window. For intercepting mouse down events in scroll bars, see Chapter 8. Chapter 9 includes a discussion of scrolling text within a window.

### Closing and Disposing of Windows

If an application needs to remove a window from the screen (rather than making it invisible or hiding it behind another window), there are two routines that will do so. **CloseWindow** is used when an application allocated its own storage for the window record:

```
PROCEDURE CloseWindow (theWindow: WindowPtr);
```

This routine removes the window from the screen and deletes it from the application's window list. Since storage for the window record was allocated by the application, that block of storage is unaffected when the window is closed. Any other data structures associated with the window are deleted from memory.

On the other hand, if an application did not give the window creation routine a storage area for the window record, but rather indicated that the window record should be placed on the heap (a **wStorage** value of 0), **DisposWindow** is used to remove it:

```
PROCEDURE DisposWindow (theWindow: WindowPtr);
```

**DisposWindow** will not only remove the window from the screen and the window list, but will release the memory used to store the window record.
Once a window has been closed with either `CloseWindow` or `DisposWindow`, it cannot be used again unless it is redefined by another call to `NewWindow` or `GetNewWindow`.

---

**Creating Menus**

Like windows, menus can be created either completely within an application program, or they can be retrieved from a template in a resource file. Creating menus within an application is far more cumbersome than creating a window within an application. It is far easier to always use a resource file for menu definitions.

**Defining Menus**

Resource file menu definitions begin with:

```
TYPE MENU
```

and, like window definitions, are followed by a second line containing a sequence number unique to that menu. (As mentioned earlier, sequence numbers need only be unique within resource type.)

The complete definition for the Video Tape Index's `Options` menu appears as:

```
TYPE MENU
  ,3
Options
Enter
Change
Delete
Select
Print
Quit/Q
```

The third line of the definition is the window’s title. The remaining lines are the options that will appear when the window is pulled down. The `/` after `Quit` indicates that `Quit` has a keyboard equivalent. When the menu is pulled down, `Quit` will appear with a cloverleaf- Q to its right and the Macintosh will interpret that key sequence as equivalent to selecting `Quit` from the menu with the mouse. Use as many keyboard equivalents for as many menu items as you wish, but the equivalents should be unique.
All Macintosh applications support at least two, and often three, standard menus. The Apple menu (its title appears as an apple symbol) provides access to the Macintosh’s built-in desk accessories; these should be available in all applications. Some of the desk accessories also require the ability to edit text. Therefore, applications should have an Edit menu, even if the remainder of the program does no text editing at all. Finally, most applications will have a File menu that handles the opening, saving, printing, and closing of files.

To define the Apple menu, use:

```
TYPE MENU
   ,1
\ 14
```

The \ indicates that the title of the menu is not a character string, but an ASCII code. In the Macintosh’s extended ASCII character set, 14 represents the solid apple symbol. No menu items are part of this definition; they are added later.

An Edit menu also has a standard format:

```
TYPE MENU
   ,2
Undo/Z
(-
Cut/X
Copy/C
Paste/V
Clear
```

The fourth line of this definition (\(-\)) prints a line across the width of the menu. Note then when numbering the items in a menu, this line counts as an item, even though it’s not an option. The line will be printed unhighlighted (dimmed, or light-grey). A left parenthesis preceding any menu item indicates that the item should be dimmed. The order of the items in an Edit menu and their keyboard equivalents are standard and should be used as shown if your application is to conform to the standard Macintosh user interface.

The remainder of the video tape index’s menu templates appear in Listing 7.2. Note that the keyboard equivalents have been selected to be as mnemonic as possible (e.g., cloverleaf- A stands for “Add a new record”). Also notice that while cloverleaf- Q stands for Quit in all of these menus, no more than one of them is present in the menu bar at any given time.
Listing 7.2 Templates for Application-Specific Menus Used in the Video Tape Index

```
TYPE MENU
,3 Options
Enter
Change
Delete
Select
Print
Quit/Q

,4 Enter
Add/A
Quit/Q

,5 Change
Find Record/F
Save Change/S
Abandon Change/A
Quit/Q

,6 Delete
Find Record/F
Delete/D
Cancel/C
Quit/Q

,7 Select
Display All
Display All Titles
Select One Title
Select by Producer
Select by Date
Select by Rating
Select by Tape Number
Quit/Q

,8 Print
Print All
Print All Titles
Quit/Q

;; menu templates follow
;; sequence number
;; menu title
;; menu item #1
;; menu item #2
;; menu item #3
;; menu item #4
;; menu item #5
;; menu item #6 (has keyboard equivalent - cloverleaf-Q)
```
Defining the Menu Record

Just as information about windows is stored in window records, information about menus is stored in menu records. Before attempting to create a menu record, though, you must first initialize the Menu Manager with:

__InitMenus

(The routine has no parameters.) This initialization should be placed directly after InitWindows.

The application must also allocate storage space for a handle to each menu record that the program will create. Since a menu handle contains a pointer to the menu record, it requires a longword of space. For example:

AppleHandle DC.L 0

will set aside space for the handle to the apple menu (the one that gives access to the desk accessories).

Assuming that space has been allocated for the menu handle, a menu record is created by the GetRMenu routine:

FUNCTION GetRMenu (resourceID: INTEGER) : MenuHandle;

resourceID refers to the sequence number you assigned to a particular menu. The function call returns the handle to the menu record. It also automatically adds menu items where menu items are specified in the resource definition. (If you were defining a menu within an application, a call to another routine would be required to add menu items to the menu record.)

To create the Video Tape Index's Apple menu, the strategy is:

CLR.L -(SP) ;space for menu handle result
MOVE #1, -(SP) ;menu number 1
__GetRMenu
LEA AppleHandle,A0 ;address to store menu handle
MOVE.L (SP)+, (A0) ;pull handle off stack and store

The menu handle is required by most Manager Routines and therefore must be recovered for subsequent use.

While menu items are automatically added to all menus that have them listed in the resource file, the Apple menu is a special case. The desk accessories must be added in a separate step. To understand what is happening, consider that, to the Macintosh, desk accessories are resources, just like windows and menus. They are stored in the system's resource file, which is opened by the system whenever any application is executed.
Adding the desk accessories to the Apple menu is accomplished by identifying a type of resource (in this case DRVR) and instructing the Macintosh to find all resources of that type and add them to the menu in question. The ToolBox routine that does this is AddResMenu:

PROCEDURE AddResMenu (theMenu: MenuHandle; theType: ResType);

ResType refers to a four-character string that identifies the resource type (e.g., WIND identifies a window resource type and MENU a menu resource type). Locating and appending the desk accessories requires:

MOVE.L AppleHandle, -(SP); menu handle on stack
MOVE.L #‘DRVR’, -(SP); 4 characters take 4 bytes
_AddAddResMenu

It is important to remember that while the menu records have been created, their handles saved, and menu items added where appropriate, no menu bar has been drawn. Getting the menu bar to appear with just the menus you want, and in the order you want, is a two-step process.

Managing the Menu Bar

Issuing a call to the routine that draws the menu bar will display only those menus that are part of the menu list. In fact, every menu for which a menu record has been created does not have to be part of the menu list; in fact, only those menus which should be displayed at any given time are members of the list. Inserting into and removing from the menu list is the way an application controls the menus available to the user.

Menu list insertion is done with the InsertMenu routine:

PROCEDURE InsertMenu (theMenu: MenuHandle; beforeID: INTEGER);

The parameter beforeID refers to the position in the menu bar where the menu referenced by theMenu (the menu handle of the menu to be inserted) should be placed relative to other menus currently in the list. If beforeID is 0, then the new menu will appear to the right of all others. On the other hand, if beforeID contains the sequence number of a menu already in the menu list, the new menu will be inserted to the left of the menu indicated by beforeID.

To delete a menu from the menu list use:

PROCEDURE DeleteMenu (menuID: INTEGER);

where menuID is the sequence number of the menu to be removed.
**InsertMenu** and **DeleteMenu** do not affect the appearance of the menu bar. Therefore, any time a change is made to the menu list, the menu bar must be redrawn. The ToolBox routine:

```assembly
PROCEDURE DrawMenuBar;
```

will take care of it. **DrawMenuBar**, which has no parameters, is simply called by:

```assembly
__DrawMenuBar
```

The Video Tape Index has eight different menus (templates for six of which appear in Listing 7.2), though no more than three are in use at any one time. The Apple and Edit menus are always present. The third menu varies with which section of the program is currently being executed. For example, when the program is launched, the three menus are **Apple**, **Edit**, and **Options**. The **Options** menu has one item for each of the program's five functions and a Quit option.

If one of the five program functions is selected, the **Options** menu is removed from the menu list. A menu corresponding to the selected function is inserted into the list and the menu bar redrawn. For example, the following code prepares the menu bar for adding new titles:

```assembly
MOVE     #3, -(SP) ;the Options menu is #3
__DeleteMenu
MOVE.L  EnterHandle, -(SP) ;put handle of Enter menu on stack
CLR      -(SP) ;new menu will go at end of menu list
__InsertMenu
__DrawMenuBar ;this makes the changes visible
```

When the user exits the function (by selecting Quit from the function's menu), the function menu is removed, the **Options** menu re-inserted, and the menu bar redrawn. To return to the main program after entering new titles, the code is:

```assembly
MOVE     #4, -(SP) ;the Enter menu is #4
__DeleteMenu
MOVE.L  OptionsHandle, -(SP) ;appropriate handle goes on stack
CLR      -(SP) ;put menu at end of menu list
__InsertMenu
__DrawMenuBar ;changes only visible after this call
```

The Video Tape Index has no File menu, since this particular application does not provide the user with file handling options. (See Chapter 11 for details on Macintosh file management.)
Controlling the Appearance of Menu Items

In some instances, you may wish to have a menu present in the menu bar while some of its menu items are not available to be selected. For example, the desk accessories which allow you to enter text support the UnDo operation. The Video Tape Index, though, supports all the text editing functions except UnDo. Therefore, when a desk accessory is being used, the UnDo item of the Edit menu should be highlighted (displayed in dark type), but when the user is entering text into the application's text windows, the UnDo item should be dimmed to indicate that it is not available.

The procedures DisableItem and EnableItem take care of dimming and highlighting menu items, respectively. To do so, they need two pieces of information: which menu and what item within that menu. Therefore, they appear as:

PROCEDURE DisableItem (theMenu: MenuHandle; item: INTEGER);

and

PROCEDURE EnableItem (theMenu: MenuHandle; item: INTEGER);

When counting the menu items to decide what number to use for item, remember to include lines as items. For example, UnDo is item 1 in the Edit menu, but Cut is item 3. To disable Clear, the assembly language statements would be:

MOVE.L EditHandle, − (SP) ; put menu handle on stack
MOVE #6, − (SP)
__DisableItem

On occasion, it is appropriate to disable an entire menu without removing it from the menu list. For example, the video tape index program disables the Edit menu when the user is printing. Since no editing is possible during print operations, it makes little sense to have an active Edit menu. A disabled menu will appear with its title dimmed. To disable an entire menu, call DisableItem with an item number of 0. Then call DrawMenuBar to redraw the entire menu bar. To re-enable the menu, call EnableItem with an item number of 0 followed again by a call to DrawMenuBar.

If you were to write a program that went this far with its menus — getting them from the resource file, inserting the desk accessory items, forming the menu list, and drawing the menu bar — you would discover that the menus did not pull down to display the menu items. There is yet another Menu Manager routine that handles pulling down the menus and registering a menu selection from the mouse. This routine is called in response to something that happens within an application — an event. Managing events and the actions that result from them is covered in Chapter 8.
Menus have a further function which may not be instantly obvious — they can help to establish a structure for an application program. Figure 7.5 presents a hierarchical block diagram of the Video Tape Index program. Note that each major program block, or module, corresponds to a separate menu. The function menus are all subordinate to the main Options menu. The code that handles each function is therefore written as a subroutine that can be called from the main program which is controlled by the Options menus.

Figure 7.5 Gross Block Diagram of Video Tape Index Program Structure
Questions and Problems

1. For each global window boundary rectangle below, indicate the top left and bottom right points of its local coordinate system.
   a. 10, 10, 200, 200
   b. 40, 40, 500, 500
   c. 200, 10, 250, 100

2. Assume that you want to define a standard document window with a boundary rectangle of 50, 20, 275, 120. The window should be visible, have a GoAway box, and be placed in front of any other windows already on the screen. It can have a title of your own choosing.
   A. Write the assembly language code that will define this window within a program. Be sure to set aside storage for any data structures your code will use. Retrieve the window pointer from the stack.
   B. Write a resource file template for the same window.
   C. Write the assembly language code that will create the window defined by the template in B.

   Be sure to allocate space for any necessary data structures and retrieve the window pointer from the stack.

3. For the window defined in problem 2 write blocks of code that will perform the following operations. (Assume that each operation is independent of any of the others.)
   A. change the title to something other than the original title
   B. make the window invisible
   C. make the window active
   D. close the window

4. Write code to prepare the window defined in problem 2 for scroll bars:
   A. write code to draw a grow icon
   B. write the control templates to define vertical and horizontal scroll bars in a resource file
   C. write code to draw the scroll bars defined by the templates you wrote for B.
5. Write a resource file template to define a standard File menu. Include options to open a new file, open an existing file, close a file, print a file, save a file, and quit the program. Include keyboard equivalents as appropriate. (For a sample of how a standard File menu might appear, see Figure 1.2.)

6. Write assembly language code to create the menu defined by the template of problem 5. Define data structures as needed. Retrieve and store the menu handle.

7. Write assembly language code to insert the menu from problem 6 into the menu list. Finish the process by redrawing the menu bar.

8. Write assembly language code to:
   A. disable the entire File menu from problem 6
   B. disable only the options which open files

   In which case must you re-draw the menu bar?
Chapter Objectives

1. To understand how events are used to control program actions
2. To be able to handle mouse down events in a variety of locations
3. To be able to process key down events as equivalents for menu selections
4. To understand the sequence of steps required to update a window

The System Event Mechanism

Macintosh applications are controlled by events. An event is anything that happens to the computer. A click on the mouse button is an event; pressing or releasing a key on the keyboard is an event. Most events that an application handles are those generated by users, though some are generated by the Macintosh itself. The most common types of events that a program will process are:

1. Null events — the system reports that nothing has happened since the last time you checked.
2. Mouse down events — the mouse button was pushed.
3. Mouse up events — the mouse button was released.
4. Key down events — a key was pressed.
5. Key up events — a key was released.
6. Update events — something has disrupted the contents on a window and it needs to be redrawn in some way. This type of event is posted by the system when, for example, a window that was previously obscured by another window is brought to the front.
7. Activate events — a text window needs to be activated or deactivated. This type of event is posted by the system whenever you call `SelectWindow`.

There is a point of potential confusion with regard to activate events. While the event is called "activate," it is generated by two distinct situations. In the first instance, a window must be deactivated; in the other, a window must be activated. Calls to `SelectWindow` produce two activate events. The first one posted to the event queue is for the window being deactivated; the second is for the window being activated.

8. Disk insertion events — a disk was inserted into a disk drive.
9. Abort events — cloverleaf-. was typed to abort an activity.

Other types of events include:

10. Auto-key events — generated by continuing to hold down a key.
11. Network events — relevant to an application that interacts with the AppleTalk network.
12. I/O driver events (currently not used).
13-16. Four events that can be defined by an application.

These constitute the maximum of 16 possible types of events.

As events occur they are posted to the event queue in first-in, first-out order. The nature of the events also determines to some extent the order in which they will be detected. Activate events have the highest priority (deactivate is first, followed by activate) and are not actually posted to the event queue. Keyboard, mouse, disk, and abort events have the next priority. Update events come after those just mentioned, and null events are of the lowest priority.

When an event is detected, the Macintosh generates an event record for it. An event record has five fields:

what: INTEGER;
message: LONGINTEGER;
when: LONGINTEGER;
where: Point;
modify: INTEGER;
The **what** field identifies what type of event the event record represents. Event types are represented by numeric codes. To identify what type of event has occurred, an application must compare the contents of the **what** field of the event record with the type codes for whatever events the program needs to trap. If SysEqu.D is INCLUDEd in an application, you can avoid using the numeric codes and reference the event types by their symbolic addresses. For example:

```
nullEvt   represents the code for a null event;
mButDwnEvt represents the code for a mouse down event;
updatEvt  represents the code for an update event;
activateEvt represents the code for an activate event.
```

Consult Table 8.1 to see the remainder of the symbolic addresses associated with event types. Using the symbolic addresses rather than the type codes makes a program more readable and easier to debug.

The meaning of the **message** field depends on the type of event being posted:

1. For keyboard events — the key that was pressed. The low-order byte contains the ASCII code for the key; the high-order byte indicates whether any modifier keys, such as the shift, cloverleaf, or option keys, were also held down.
2. For update and activate events — a pointer to the window where the event occurred
3. For disk insert events — the drive number where the event occurred
4. For abort events — the key that was pressed. The low-order byte contains the ASCII code for the key; the high-order byte identifies any modifier keys that were also held down.
5. For mouse and null events — the field has no meaning

An application commonly compares the **message** field to its own window pointers to determine which windows need updating and activating. **message** is used less frequently to directly read the keyboard for text entry; that function is handled by the TextEdit routines discussed in Chapter 9.

**when** indicates the time when an event was posted. For most applications, this field is of less importance than any of the others.

**where** gives the coordinates of the mouse when the event was posted. These coordinates are **global** (i.e., expressed in terms of the 512 by 342 coordinate grid imposed on the entire screen). **where** is used in conjunction with routines that identify where a mouse down event occurred.

**modify** holds information about the state of a number of Macintosh keys; the **modify** word works as a series of flags. If set, each flag indicates that a particular key was pressed. **modify** monitors the mouse button, the cloverleaf key, the shift key, the caps lock key, and the options key. It also records whether an "activate" event represents the deactivating or activating of a window. Take a look at Table 8.2 to see the bit assignments in **modify**.
When retrieving events from the event queue, an application can choose to receive all events in order or only events of a specific type. Events can be filtered out by using a specific event mask. It is no accident that there are 16 possible event types. Each type corresponds to one bit in an integer, or word, length event mask. For example, if bit 0 is set, then the mask will include null events. If bit 1 is set, the mouse will also include mouse down events. The bit positions that represent the various types of events appear in Table 8.3.

The Macintosh will accept a mask of −1 to select every event, the mask which should be used in most instances. In other words, an application should retrieve
every event from the queue and then compare the **what** field of the event record against the types of events the application wishes to process.

In some special cases it may be necessary to construct a special mask. For example, when managing its windows and TextEdit records, the Video Tape Index program must remove some spurious activate and deactivate events from the event queue. Table 8.3 indicates that activate events are selected when bit 9 is set. Therefore, the mask used to remove those events is %-0000000100000000 or more simply, **256**.

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>No event reported</td>
</tr>
<tr>
<td>1</td>
<td>Mouse down</td>
</tr>
<tr>
<td>2</td>
<td>Mouse up</td>
</tr>
<tr>
<td>3</td>
<td>Key down</td>
</tr>
<tr>
<td>4</td>
<td>Key up</td>
</tr>
<tr>
<td>5</td>
<td>Auto key</td>
</tr>
<tr>
<td>6</td>
<td>Update</td>
</tr>
<tr>
<td>7</td>
<td>Disk insertion</td>
</tr>
<tr>
<td>8</td>
<td>Activate</td>
</tr>
<tr>
<td>1Ø</td>
<td>Network</td>
</tr>
<tr>
<td>11</td>
<td>Device driver</td>
</tr>
<tr>
<td>12 - 15</td>
<td>Application defined</td>
</tr>
</tbody>
</table>

Setting any given bit in a mask word will instruct **GetNextEvent** to report events of that type. For example, if bits 1 and 3 are set, **GetNextEvent** will report only mouse down and key down events. A mask of -1 will select all types of events.

**Table 8.3 The Structure of an Event Mask**

As part of the initialization process, an application should flush the event queue to remove any events that may have been posted prior to the application's execution. Usually this occurs immediately after the various managers have been initialized with a call to **FlushEvents**. **FlushEvents** is an operating system routine:

```
PROCEDURE FlushEvents (eventMask, stopMask: INTEGER);
```

The event mask is constructed as described above. The stop mask says "return all events that meet the event mask until an event that matches the stop mask is encountered." In most cases, use a stop mask of 0 to indicate that all events specified by the event mask should be removed.

**FlushEvents** expects to find both of its parameters in D0 – the event mask in the low-order word and the stop mask in the high-order word. It is easiest to load the register with one **MOVE** statement:

```
MOVE.L $00000000,D0
```
This installs a stop mask of 0 and an event mask of −1 (the $FFFF represents −1 as a two's complement integer).

The procedure is then called by using:

```plaintext
__FlushEvents
```

Normally, `FlushEvents` is called only once, at the start of an application.

## Retrieving Events

Before an application can retrieve events from the event queue, it must prepare storage for the event record. The event record data structure can be defined to be part of the application itself (using DC) or assigned to the applications globals area (with DS). The Video Tape Index places its event record storage within the program:

```plaintext
EventRecord
What       DC  0
Message    DC.L 0
When       DC.L 0
Point       DC.L 0
Modify      DC 0
```

Using `EventRecord` will reference the entire 16-byte data structure. Each field can also be referenced separately using its own symbolic address.

Programmers who wish to keep all read/write storage in the applications globals area should use:

```plaintext
EventRecord   DS.B 16
```

which will set aside the required 16-byte area. The start of the fields within the record are then handled as offsets. They are part of the predefined equates in SysEqu.D (e.g., evtNum, evtMessage, etc.).

Events are retrieved from the event queue with the ToolBox routine `GetNextEvent`:

```plaintext
FUNCTION GetNextEvent (eventMask: INTEGER; VAR theEvent: EventRecord) : BOOLEAN;
```

`GetNextEvent` returns a boolean that is set to false if the event is one that should be handled by the system (and not by the application program) or a null event. The event mask is as discussed above. Only those events which fit the mask will be reported by the call to `GetNextEvent`. 
The event details are returned in the event record data structure. It is passed as a variable parameter rather than placed on the stack as the function result. Therefore, a call to `GetNextEvent` appears as:

```assembly
CLR   -(SP) ; space for boolean result
MOVE  # -1, -(SP) ; -1 is the preset mask for all events
PEA   EventRecord ; since EventRecord is a variable parameter,
                    ; push pointer
__GetNextEvent
```

Calls to `GetNextEvent` form the basis of the loop that controls selection of program actions. Figure 8.1 shows you pseudocode for an event loop. In general, the strategy is to begin by checking the event queue for an event. If none is reported, the loop simply returns to check again. If an event was posted, then the application must isolate the event number from the event record and compare it to the numbers representing each type of event the program must handle. When a match is discovered between a posted event and one the application will deal with, the program should branch to a module that handles that event. In this way, the event loop determines the structure of the program.

```
Repeat
    Retrieve an event from event queue;
    If an event has been posted then
        Retrieve event number from event record;
        Repeat
            If event number equals an event this program monitors then
                Branch to portion of program that handles that event
            Until event number equals an event this program monitors OR all
                possible event numbers have been checked;
        Until users selects Quit.
```

**Figure 8.1 Pseudocode for an Event Loop**

The event loop in the Sample program is deceptively simple. Since that program's only function is to open a window, print a string, and wait for a key or mouse button press, the event mask selects only those two events (a mask of
%00000000000111110); the program either finds a mouse or keyboard event, or it
doesn't:

```
Event  CLR  -(SP)                    ;space for boolean result
       MOVE  #%00000000000111110, -(SP)  ;event mask
       PEA   EventRecord             ;pointer to event record storage
       _GetNextEvent
       MOVE  (SP)+,D0                 ;recover boolean result
       CMP   #0,D0                    ;no event — loop to keep checking
       BEQ   Event
       RTS
```

The reason that this loop is so simple is that it can detect the occurrence of a
desired event merely by checking the boolean result of the call `GetNextEvent'.
Since the action to be taken is identical whether the event is a key or mouse button
press (return to the Finder), there is no need to differentiate between the two types
of events. In terms of meaningful Macintosh applications, this is an unrealistic
situation.

The Video Tape Index must handle four different types of events. Its main event
loop, which traps three types of events, appears in Listing 8.1. Note that this event
loop uses an event mask of -1 to select all types of events and then makes
comparisons with specific event numbers to identify the particular events it must
handle.

```
Listing 8.1 Video Tape Index Main Event Loop
```

```
Event
   CLR  -(SP)                    ;Space for boolean result
   MOVE  #-1,-(SP)               ;Mask for keyboard - select all events
   PEA   EventRecord             ;Place to receive event info
        _GetNextEvent             ;Get next event from queue
   MOVE  (SP)+,D0                 ;Recover event result
   CMP   #0,D0                    ;If no event, branch to look again
   BEQ   Event
   MOVE  What,D0                  ;Recover event ID
   CMP   #mButDwnEvt,D0           ;Was mouse button pressed?
   BEQ   MouseEvent
   CMP   #keyDwnEvt,D0            ;Was key pressed?
   BEQ   KeyEvent
   BRA   Event                    ;Look for another event
```
Each of the program's function modules also have their own event managers. These trap all four kinds of events: mouse down, key down, update, and activate. Code for the module which finds and displays data can be found in Listing 8.2. As you can see, the structure of this loop is essentially the same as the main event loop. This is one situation where repeated code is not necessarily a negative characteristic.

**Listing 8.2** Function Event Loop from the Video Tape Index Program

```assembly
SelectEvent
CLR -(SP) ; space for event type
MOVE #-1,(SP) ; event mask of -1 selects all events
PEA EventRecord ; place to store event record
_GetNextEvent ; get next event from event queue

MOVE (SP)+,D0 ; recover boolean result
CMP #0,00 ; 0 result means no event occurred
BEQ Select ; if no event, branch to keep looking

MOVE What, D0 ; recover event type
CMP #mButDwnEvt, D0 ; mouse down event?
BEQ SelectMouseEvent ; branch to handle event

CMP #keyDwnEvt,D0 ; key down event?
BEQ SelectKeyEvent ; branch to handle event

CMP #activateEvt, D0 ; activate event?
BEQ SelectActivateEvent ; branch to handle event

CMP #updatEvt, D0 ; update event?
BEQ SelectUpdateEvent ; branch to handle event

BRA SelectEvent ; some unwanted type of event occurred - must ; keep checking
```

Though it is possible to write a significant Macintosh application with only one event loop, in most cases doing so creates "spaghetti code," code that is so intertwined with unconditional branches (**JMP** and **BRA**) that it is virtually impossible to follow and even more difficult to modify and debug. The Video Tape Index opts for clear program structure over the tightest, shortest possible code. Unfortunately, such a choice may not be viable if you are writing a large application for the 128K Mac.
Handling Mouse Down Events

The primary task with which an application is faced when a mouse down event occurs is figuring out where the mouse button was pressed. The three major locations an application usually examines are:

1. the menu bar
2. the content area of a window defined by an application
3. a system window (one created, for example, by a desk accessory)

In some applications, the mouse button might also be pressed in the drag region of a window, in a GoAway box, or in a grow box.

The Window Manager routine FindWindow identifies where the mouse button was pressed:

FUNCTION FindWindow (thePt: Point; VAR whichWindow: WindowPtr) : INTEGER;

thePt refers to the screen coordinates where the mouse button down event occurred. It can be obtained from the point field of the event record. On function return, the variable parameter whichWindow will contain the pointer to the window record of the window posting the event. The integer result contains a code that corresponds to the event's general location (e.g., 1 = in the menu bar, 2 = in a system window, etc.). Symbolic addresses for each of the result codes are established in the ToolEqu.D file; the complete set also also appears in Table 8.4.

A call to FindWindow therefore appears as:

<table>
<thead>
<tr>
<th>Symbolic Address</th>
<th>Result Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>inDesk</td>
<td>Ø</td>
<td>not in a window or the menu bar</td>
</tr>
<tr>
<td>inMenuBar</td>
<td>1</td>
<td>in the menu bar</td>
</tr>
<tr>
<td>inSysWindow</td>
<td>2</td>
<td>in a system window (e.g., a desk accessory)</td>
</tr>
<tr>
<td>inContent</td>
<td>3</td>
<td>in the content area of an application</td>
</tr>
<tr>
<td>inDrag</td>
<td>4</td>
<td>in the drag region of an application</td>
</tr>
<tr>
<td>inGrow</td>
<td>5</td>
<td>in the grow region of an application</td>
</tr>
<tr>
<td>inGoAway</td>
<td>6</td>
<td>in the GoAway box of an application</td>
</tr>
<tr>
<td>inButton</td>
<td>10</td>
<td>in a push button</td>
</tr>
<tr>
<td>inCheckBox</td>
<td>11</td>
<td>in a check box</td>
</tr>
<tr>
<td>inUpButton</td>
<td>20</td>
<td>in up button area of a scroll bar</td>
</tr>
<tr>
<td>inDownButton</td>
<td>21</td>
<td>in down button area of a scroll bar</td>
</tr>
<tr>
<td>inPageUp</td>
<td>22</td>
<td>in page up area of a scroll bar</td>
</tr>
<tr>
<td>inPageDown</td>
<td>23</td>
<td>in page down area of a scroll bar</td>
</tr>
<tr>
<td>inThumb</td>
<td>129</td>
<td>in thumb area of a scroll bar</td>
</tr>
</tbody>
</table>

Table 8.4 Symbolic Addresses Associated with FindWindow Result Codes
A call to Find Window therefore appears as:

```
CLR   - (SP) ; space for integer result
MOVE.L Point, - (SP) ; a field from the event record
PEA   WhichWindowPtr ; must be defined with DC or DS
___FindWindow
```

The integer result should then be immediately pulled off the stack:

```
MOVE (SP) + ,D0
```

The code in D0 can then be compared against the codes for locations the application needs to monitor. For example:

```
CMP #inMenuBar,D0
```

will determine whether the mouse button was clicked anywhere in the menu bar. The constant inMenuBar is defined in the ToolBox equates file.

FindWindow also returns location codes for:

1. in a system window
2. in the content region of an application window
3. in the drag region of a window (the title bar)
4. in the grow region
5. in a GoAway box

All of these locations have constants defined for them in the ToolBox equates file.

When a match is found with a location code, the application should branch to handle that particular situation. Each location requires that the program execute a different series of actions. They are discussed separately below.

### Mouse Down Events in Menu Bars

There is a single ToolBox routine MenuSelect, that pulls down menus (displaying the menu items), highlights the menu title, and records which menu and which item within that menu were selected. Any time a program records a mouse down even in the menu bar, it should call MenuSelect:

```
FUNCTION MenuSelect (startPt: Point) : LONGINTEGER;
```

MenuSelect's longinteger result has two parts. The high-order word contains the sequence number of the menu. This is the number assigned to the menu in the
resource file. The low-order word contains the number of the selected menu item. (Remember: when numbering menu items, things such as lines across the menu, like that beneath UnDo in an Edit menu, count as items as far as MenuSelect is concerned.) The parameter startPt is again the Point field from the event record.

A call to MenuSelect therefore appears as:

```
CLR.L - (SP) ;space for longinteger result
MOVE.L Point, - (SP) ;from the event record
```

MenuSelect's result should be pulled from the stack and put into two integer storage locations. Making this work properly takes a bit of care. First, two integer locations should be defined using either DC or DS:

```
WhichMenu DC 0
Whatltem DC 0
```

These declarations should be placed physically next to each other (in the order above) in the program. This ensures that when the storage is allocated during assembly, Whatltem will occupy the word immediately after WhichMenu in memory. Then:

```
MOVE.L (SP)+,D2 ;pulls result from stack
LEA WhichMenu,A0 ;get address for high-order word
MOVE.L D2,(A0)
```

The important step above is this last one — it moves a longinteger rather than just an integer. The high-order word of the result therefore goes into the word associated with the symbolic address WhichMenu. Since the location of Whatltem is physically right after WhichMenu, the low-order byte is automatically stored in the right place. This bit of tricky maneuvering saves several program steps (i.e., having to save WhichMenu, mask off the high-order byte, and then save Whatltem explicitly).

Since MenuSelect does not unhighlight a menu title, every call to MenuSelect needs to be followed by a call to HiLiteMenu, which will remove the highlighting:

```
PROCEDURE HiLiteMenu (menuID: INTEGER);
```

This easiest way to handle this is not to determine exactly which menu needs to have its title unhighlighted, but to unhighlight all menus. To do so, simply use a menuID of 0, which will automatically unhighlight any menu title which is highlighted.

The next task is to compare the contents of WhichMenu with the sequence numbers of the menus currently in the menu bar. These sequence numbers are
the ones assigned to the menus in the application's resource file. When a match is found, the program must then branch to handle actions based on the specific menu.

Assuming that an application adheres to the standard Macintosh user interface, there will be at least three sorts of menus that it must deal with: the Apple menu that selects the desk accessories; the Edit menu which may reflect text editing in an application window or a desk accessory; and application menus (those specific to the particular program). Most applications will also have a standard File menu.

**Implementing the Desk Accessories**

Most of the work involved with handling the Macintosh's standard desk accessories is done by the ToolBox itself. In order for them to be properly updated, however, an application must make repeated calls to `SystemTask`. This procedure (it has no parameters and so is simply called with `SystemTask`) should be placed in an application's event loop. If an application has more than one event loop, it should appear in each of them. If `SystemTask` is not called frequently enough, desk accessories such as the alarm clock will not function properly.

The first task in processing a desk accessory is to identify which desk accessory has been selected. The routine `GetItem` will return the text of a selected menu item:

```plaintext
PROCEDURE GetItem (theMenu : MenuHandle; item: INTEGER;
              VAR itemString: Str255);
```

The first parameter is the handle of the Apple menu. The item should come from `WhatItem` which was retrieved earlier from the call to `MenuSelect`. The result of this procedure — the name of the desk accessory — should go into a storage location that has been defined with a length of 16 words (since that is the maximum length of a desk accessory name):

```plaintext
DeskAccName   DCB.W 16,0
```

or

```plaintext
DeskAccName   DS.W 16
```

The call to `GetItem` appears as:

```plaintext
MOVE.L AppleHandle, -(SP) ;menu handle goes on stack
MOVE WhatItem, -(SP) ;from MenuSelect
PEA DeskAccName(A5) ;assumes DS declaration
__GetItem
```
At this point, the application has enough information to open the desk accessory and turn its execution over to the system. This is accomplished using `OpenDeskAcc`, a routine that is part of the Desk Manager:

```plaintext
FUNCTION OpenDeskAcc (theAcc: Str255) : INTEGER;
```

The result of `OpenDeskAcc` can be ignored. Nevertheless, the call must allocate space for the result and retrieve it from the stack. The parameter `theAcc` is the desk accessory name retrieved in the call to `Getltem`. To call `OpenDeskAcc` use:

```plaintext
CLR  -(SP)                        ;space for useless result
PEA  DeskAccName(A5)             ;assumes storage defined with DS
__OpenDeskAcc
MOVE  (SP)+,D0                    ;removes useless result from stack
```

Once an application calls `OpenDeskAcc`, the desk accessory is opened for the user. The system will handle things such as key down events, but the application should continue to monitor the event queue, looking for mouse down events in system windows.

### Handling Edit Functions in Desk Accessories

When a user makes a selection in the Edit menu, there are two possibilities: either the edit request concerns a system window (e.g., the note pad) or an application window. Edit functions in application windows will be discussed later in Chapter 9 along with the other TextEdit routines, but this is an appropriate place to consider how to differentiate between the two sources of edit requests and how to handle those in system windows.

The strategy for telling system edit requests from application edit requests is very straightforward. An application should simply attempt to let the system handle the request. If it can, it will. If the system is unable to handle an edit (because it occurred in an application window) it will return a result of FALSE. A FALSE result therefore means that the application must handle the edit itself.

Editing in system windows is taken care of by a single ToolBox routine:

```plaintext
FUNCTION SysEdit (editCmd: INTEGER) : BOOLEAN;
```

The `editCmd` is equal to the item number from the edit menu less 1, assuming that the items in the Edit menu have been set up in the standard order.
To initiate a system edit, then:

```
MOVE Whatltem,D0 ;retrieve original menu item number
SUBQ #1,D0 ;adjust the item number to suit SystemEdit
CLR -(SP) ;space for boolean result
MOVE D0, -(SP) ;put adjusted item number on stack
_SYSEdit ;let the system handle the edit request
MOVE (SP)+ ,D1 ;get result to verify if request was handled
```

Control returns to the application when the edit has been completely processed.

Handling Mouse Down Events in Application Menus

Precisely what occurs as the result of selecting any given menu item will obviously depend on the nature of the menu. Nonetheless, the strategy for identifying the item selected is the same — compare `Whatltem` against each of the item numbers present in the selected menu. When a match is found, branch to a program module that implements the particular function. (For an example, see Listing 8.3, the code that selects actions from the Video Tape Index's Options menu.)

**Listing 8.3** Selecting Program Actions Based on Menu Selections (from the Video Tape Index)

<table>
<thead>
<tr>
<th>(a) Options</th>
<th>MOVE Whatltem,D0 ;Move item selected to D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) CMP</td>
<td>#1,D0</td>
</tr>
<tr>
<td></td>
<td>BNE Item2</td>
</tr>
<tr>
<td>(c) JSR</td>
<td>Enter</td>
</tr>
<tr>
<td>Item2</td>
<td>CMP #2,D0</td>
</tr>
<tr>
<td></td>
<td>BNE Item3</td>
</tr>
<tr>
<td></td>
<td>JSR Change</td>
</tr>
<tr>
<td>Item3</td>
<td>CMP #2,D0</td>
</tr>
<tr>
<td></td>
<td>BNE Item4</td>
</tr>
<tr>
<td></td>
<td>JSR Delete</td>
</tr>
<tr>
<td>Item4</td>
<td>CMP #4,D0</td>
</tr>
<tr>
<td></td>
<td>BNE Item5</td>
</tr>
<tr>
<td></td>
<td>JSR Select</td>
</tr>
<tr>
<td>Item5</td>
<td>CMP #5,D0</td>
</tr>
<tr>
<td></td>
<td>BNE Item6</td>
</tr>
<tr>
<td></td>
<td>JSR Print</td>
</tr>
<tr>
<td>Item6</td>
<td>CMP #6,D0</td>
</tr>
<tr>
<td></td>
<td>BEQ Quit</td>
</tr>
</tbody>
</table>

;Enter new tapes

;Modify existing tapes

;Delete tapes

;Retrieve info

;Print lists

;Exit the program
The basic strategy behind the code in Listing 8.3 is based on knowing the order in which menu items were listed in the resource file. Listing 8.3 figures out which item in the Options menu was selected. The item list for that Options menu is:

Enter
Change
Delete
Select
Print
Quit

The Macintosh assigns the number 1 to the first item in the list (Enter), a 2 to the second (Change), and so on. The quantity stored in WhatItem (retrieved from the event record) therefore corresponds to the number of whichever item was selected. The only way to identify the particular item is to begin comparing the quantity in WhatItem with the numbers of menu items. That is exactly what the code in Listing 8.3 is doing.

WhatItem is first moved into a data register (a). The comparisons begin at (b), where the item selected is compared against a 1, the number that stands for Enter. If a match is found, program control is transferred to a subroutine that handles entering new records (c). Since blocks of code that correspond to individual menu items can be selected repeatedly while the program is running, it makes sense in terms of program structure to place each block in a separate subroutine. The procedure is repeated for each possible item until a match is found.

Handling Mouse Down Events in System Windows

If a call to FindWindow determines that a mouse down event has occurred in a system window (i.e., a desk accessory), then the application can simply turn control over to the system to handle the event. A call to SystemClick will process any type of mouse down event in a system window:

PROCEDURE SystemClick (theEvent: EventRecord; theWindow: WindowPtr);

SystemClick needs the entire event record (push its address onto the stack) and the result of FindWindow, generally stored in a location like WhichWindowPtr. The pointer can simply be moved onto the stack.

SystemClick will handle all manner of mouse down events in system windows. It will, for example, close a desk accessory if the mouse down event occurred in the desk accessory's GoAway box. It handles the mouse down events that operate the calculator. It will also select and make active a desk accessory that was previously deactivated by selection of another window. SystemClick will drag, scroll, and size system windows as well.
Handling Mouse Down Events in Application Windows

A mouse down event in the content region of an application-defined window (identified by the constant inContent) usually means one of two things: if the window is inactive, then it should be brought to the front of the screen and activated; if it is already active, then the mouse down event most often indicates that the cursor should be moved, regardless of whether the window contains text to be edited or pictures to be drawn.

The first task after identifying a mouse down event in an application window, then, is to discover whether or not the window posting the event is active. The Window Manager routine FrontWindow will return a pointer to whichever window is in front of all others on the screen (this will be the active window):

FUNCTION FrontWindow : WindowPtr;

Note that this function has no parameters. To call it, allocate space on the stack for a longinteger result and then issue the function call. After the result is retrieved from the stack, it can be compared to FindWindow's result. If the two pointers match, then the event occurred in the active window and the cursor should be moved. (See the discussion of TextEdit later in Chapter 9 for details.) If the pointers do not match, then a window must be activated. Such a code sequence might appear as:

CLR.L - (SP) ;space for longinteger result
___FrontWindow ;get pointer to active window
MOVE.L (SP)+,A0 ;recover pointer to active window
CMP.L WhichWindowPtr,A0 ;check active window against clicked window
BNE Mustactivate ;routine which activates clicked window
; followed by code to move the cursor in the active window

A window should be activated by calling SelectWindow. (See Chapter 7 for details.) This will bring the window to the front of the screen, deactivate and unhighlight the current window, and highlight the new active window as appropriate. It will also generate a deactivate event for the previously active window, and activate and update events for the new active window.

Mouse Down Events in Application Windows With Scroll Bars

If an application window contains scroll bars, then the procedure for processing mouse down events in the content area of that window is more complex than described above. After detecting an event in the content area, the application must then determine whether or not the mouse down event was in the scroll bars.
The Control Manager routine `FindControl` will identify which control, if any, was the site of the mouse down event:

```
FUNCTION FindControl (thePoint: Point; theWindow: WindowPtr;
   VAR whichControl: ControlHandle) : INTEGER;
```

`FindControl` returns two results. The first is the handle to the control record of the control that posted the mouse down event. If the mouse down event was not in a control, the control handle will be set to 0. The function's integer result corresponds to the part of the control that was clicked.

Scroll bars have five parts (see Figure 8.2). The numbers in parentheses in Figure 8.2 correspond to each part's identification number. One of these numbers will be returned as `FindControl`'s integer result for a mouse down event in a scroll bar.

![Figure 8.2 The Parts of a Scroll Bar](image)

`FindControl` needs to know the point where the mouse button was clicked. Unfortunately, the Event Manager returns the point in global coordinates and `FindControl` requires local coordinates. The QuickDraw routine `GlobalToLocal` (discussed further in Chapter 9) will handle the conversion:

```
PROCEDURE GlobalToLocal (VAR pt: Point);
```
An application might use the following code to determine whether a mouse down event occurred in a scroll bar:

```
PEA Point                  ;push address so value can return
__GlobalToLocal           ;convert
CLR            ;space for part code result
MOVE.L    Point, -(SP)       ;coordinates now local
MOVE.L    WhichWindowPtr, -(SP) ;result of call to FindWindow
PEA WhichControlHandle     ;push address so value can return
__FindControl
MOVE   (SP) +,D0            ;retrieve part code
CMP    #0,D0
BEQ    InContentRegion     ;not in scroll bar

;continue to process event in scroll bar
```

If `FindControl` returns a part code greater than 0, then the mouse down event was indeed in a scroll bar (this assumes that there are no other controls in the window). The application should then call `TrackControl`:

```
FUNCTION TrackControl (theControl: ControlHandle; startPt: Point; actionProc: ProcPtr) : INTEGER

TrackControl performs a number of important tasks. If the user has pressed the mouse button in the thumb of a scroll bar, `TrackControl` will continue to drag that thumb as long as the mouse button is held down. If the mouse button is pressed in the up or down arrow, `TrackArrow` will highlight the arrow until the mouse button is released.

`TrackControl`'s result is either the part code for the part of the control that posted the mouse down event, or 0. A value of 0 indicates that the user moved the mouse pointer from the part of the control where the event originally occurred. If that is the case, the application should abort processing the event and return to the top of the event loop.

The parameter `theControl` refers to the result of `FindControl`. `startPt` is the same point, in local coordinates, that was passed to `FindControl`. The third parameter, `actionProc`, is an optional pointer to a routine that should be executed while the user continues to hold down the mouse button. It can be set to 0 if there is no action procedure.
Calling \texttt{TrackControl} would therefore appear as follows:

\begin{verbatim}
CLR     - (SP)   ;space for part code result
MOVE.L WhichControlHandle, - (SP)  ;FindControl result
MOVE.L Point, - (SP)  ;in local coordinates
CLR.L   - (SP)   ;no action procedure
MOVE    (SP) +,D0  ;retrieve part code result
CMP     #0,D0     ;has user moved to different part
BEQ     Event     ;Yes - go to top of event loop
\end{verbatim}

;otherwise, scroll the content of the window appropriately

The actual scrolling of text will be discussed in Chapter 9, when we talk about TextEdit.

\section*{Handling Mouse Down Events in GoAway Regions}

If a document window has been defined with a GoAway box (also known as a close box), then the application should check the result of \texttt{FindWindow} against the constant \texttt{inGoAway} (equated to the value of 6 in the Toolbox equates file). A mouse down event in a GoAway box indicates that the window should be closed.

To adhere to the standard Macintosh user interface, the GoAway box should be highlighted as long as the mouse button is pressed. The Window Manager routine \texttt{TrackGoAway} will do so:

\begin{verbatim}
FUNCTION TrackGoAway (theWindow: WindowPtr; thePt: Point): BOOLEAN;
\end{verbatim}

The parameter \texttt{theWindow} is the pointer to the window record of the window posting the event; it is the result of a call to \texttt{FindWindow}. \texttt{thePt} is the point where the mouse down event occurred. It is expressed in global coordinates and can therefore be taken directly from the event record. The boolean result is set \texttt{TRUE} if the mouse pointer was still in the GoAway box when the mouse button was released; it is set \texttt{FALSE} if the pointer was moved. In the latter case, the user has effectively cancelled the request to close the window, and the application should simply return to the top of the event loop to check for another event.
Code to handle a mouse down event in a GoAway box might appear as follows:

CLR.B \(\text{-(SP)}\); space for boolean result (system will
\hspace{1cm}push extra byte to keep stack pointer
\hspace{1cm}; even)

MOVE.L WhichWindowPtr, \(\text{-(SP)}\); from FindWindow

MOVE.L Point, \(\text{-(SP)}\); from the event record

\_TrackGoAway

MOVE.B \(\text{(SP) + ,D0}\); retrieve boolean result

CMP \#0,D0; did user move pointer?

BEQ Event; Yes – go get another event

; application must continue by closing the window

Closing the window may involve simply calling \text{CloseWindow} or \text{DisposWindow} to remove the window from the screen or from memory. If the contents of the window should be saved to disk before closing, then the application may execute a disk save routine before closing the window. (See Chapter 11 for details.)

Handling Mouse Down Events in Drag Regions

In a standard document window, the drag region is the bar in which the title appears. Mouse down events in that area indicate that the user wishes to move the window somewhere else on the desktop.

The Window Manager routine \text{DragWindow} will handle the entire process:

\text{PROCEDURE} DragWindow (theWindow: WindowPtr; startPt: Point; boundsRect: Rect);

The first two parameters are identical to those for \text{TrackGoAway}. The third parameter, \text{boundsRect}, is a rectangle in global coordinates that describes the boundaries within which the window can be moved. While the rectangle could theoretically encompass the entire screen, it is generally 4 pixels in from each edge of the screen to ensure that at least 4 pixels of a document window's title bar will always be seen.

\text{DragWindow} will continue to drag an outline of the window around the screen until the user releases the mouse button. At that point, assuming the mouse pointer is within the boundary rectangle, the window will be redrawn in its new location. If the mouse pointer is not within the boundary rectangle, the window will be left in its original spot.
As an example, assume that a boundary rectangle has been defined with coordinates 4 pixels in from each edge of the screen (remember that the screen is 342 pixels high and 512 pixels wide):

\[
\text{BoundaryRect DC 4,4,338,508}
\]

A call to \texttt{DragWindow} would then appear as:

\[
\begin{align*}
\text{MOVE.L} & \quad \text{WhichWindowPtr, } -(\text{SP}) & \quad ;\text{from FindWindow} \\
\text{MOVE.L} & \quad \text{Point, } -(\text{SP}) & \quad ;\text{from the event record} \\
\text{PEA} & \quad \text{BoundaryRect} \\
\text{BRA} & \quad \text{DragWindow} & \quad ;\text{go get another event}
\end{align*}
\]

It is important to remember that \texttt{DragWindow} does not change the size of a window; it merely moves it around the screen.

**Handling Mouse Down Events in Grow Regions**

If a mouse down event occurs in the grow icon, the user wishes to change the size of the window. Sizing a window requires a sequence of calls to at least five Window Manager routines:

1. Make calls to \texttt{InvalRect} to place any parts of the window that you know will need to be changed into the window’s update region. The update region holds all of the parts of the window that have been disturbed by some program function and therefore need to be updated. If anything is present in the update region, the system will generate an update event for the window. If a window contains scroll bars, they should be placed in the update region.

2. Call \texttt{GrowWindow} to get an outline of the new size that will follow the outline of the mouse pointer until it is released. \texttt{GrowWindow} returns the coordinates of the bottom right corner of the new size. When a window is sized, its top left corner is anchored on the screen; only the position of the bottom right corner changes.

3. Call \texttt{SizeWindow} to actually change the size of the window.

4. Update the window. This may include re-drawing scroll bars and the grow icon based on the window’s new size. For a summary of the update process, see the end of this chapter.
   a. Call \texttt{BeginUpdate} (among other things, clears out the update region)
   b. Call \texttt{EraseRect}
   c. Re-draw window contents
   d. Call \texttt{HideControl} to get rid of the old scroll bars
   e. Call \texttt{MoveControl} to move the scroll bars
   f. Call \texttt{SizeControl} to change the size of a scroll bar’s rectangle
CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS 215

g. Call **ShowControl** to make the scroll bars appear
h. Call **DrawGrowlcon** to redraw the grow region
i. Call **EndUpdate**

Let's now look at a code that will implement the above sequence. The window that will be sized is the window with scroll bars from Figure 7.3b. The first step in the process is to take care of ensuring that the scroll bars get into the update region. To do so requires two calls to **InvalRect**, one for each scroll bar:

**PROCEDURE InvalRect (badRect: Rect);**

In order to call this routine, an application needs to know only the scroll bars' boundary rectangles. But if the boundary rectangles change each time the window is sized, how can the application keep track of them? The coordinates in the resource file will no longer be valid once a window has been sized. The answer lies in the control record. The coordinates of a control's current boundary rectangle are contained in the third field of the control record:

```
MOVE.L BottomControlHandle,A0 ;get handle
MOVE.L (A0),A0 ;get pointer
LEA cntrlRect(A0),A0 ;get starting address of ;boundary rectangle
MOVE.L A0, – (SP) ;pushes pointer to ;rectangle
_InvalRect

MOVE.L SideControlHandle,A0 ;get handle
MOVE.L (A0),A0 ;get pointer
LEA cntrlRect(A0),A0 ;address of rectangle
MOVE.L A0, – (SP) ;push pointer
_InvalRect
```

The second step is to call **GrowWindow**:

**FUNCTION GrowWindow (theWindow: WindowPtr; startPt: Point; sizeRect: Rect); LONGINT;**

**theWindow** comes from a call to **FindWindow**; it is the window reporting the mouse down event. **startPt** is the **point** field from the event record, the place where the mouse pointer was when the button was first pressed. **sizeRect** is a rectangle that defines the maximum size that the window can be. For this example, **sizeRect** will be defined as 4 pixels in from each edge of the screen (4,4,338,508):

```
CLR.L – (SP) ;space for coordinate result
MOVE.L WhichWindowPtr, – (SP) ;from FindWindow
MOVE.L Point, – (SP) ;from event record
```
**PEA** SizeRect ; defined as constant

_GrowWindow

MOVE.L (SP)+,D0 ; retrieve result

_GrowWindow_’s result is a long integer containing the new bottom right coordinates of the window. The high-order word contains the bottom (vertical) coordinate and the low-order word the right (horizontal) coordinate. These coordinates must, in turn, be passed to _SizeWindow_:

PROCEDURE SizeWindow (theWindow: Window; w, h: INTEGER; 
fUpdate: BOOLEAN);

The parameter _theWindow_, as before, comes from the call to _FindWindow_. _w_ refers to the new horizontal coordinate of the bottom right corner of the window, _h_ to the new vertical coordinate. At first it may seem that these are in the opposite order from which they were returned by _GrowWindow_. They are not. The horizontal coordinate should be deeper in the stack than the vertical coordinate, since it appears first in the procedure’s parameter list. Therefore, if the entire long integer from _GrowWindow_ is moved onto the stack as a unit, its low-order word will appear as an integer that is just below its high-order word, placing the two coordinates in the proper order.

_fUpdate_ is a flag that indicates whether or not the system should generate an update event if the sizing changes the window’s contents. A value of TRUE, the setting most commonly used, tells the system to do so:

```
MOVE.L WhichWindowPtr, - (SP) ; from FindWindow
MOVE.L D0, - (SP) ; coordinates recovered
ST. "ST" - (SP) ; yes – generate update
```

_SizeWindow_

The final major step in the process of sizing a window is to update the new window. All updates begin with a call to _BeginUpdate_ and should then erase the window. Note that since the QuickDraw routine _EraseRect_ works on the current graport, the application must first ensure that the window posting the mouse down event is the current graport (see Handling Update Events):

```
MOVE.L WhichWindowPtr, - (SP) ; from FindWindow
_BeginUpdate_
MOVE.L WhichWindowPtr, A0 ; retrieve handle to content
MOVE.L contRgn(A0), A0 ; region of window from
```

>window record
CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS

Preparing for the call to `EraseRect` above presents a similar problem as did the calls to `InvalRect`: this block of code should be general enough to work regardless of which window posted the mouse down event. Therefore, the boundary rectangle of the area which should be erased cannot be defined explicitly within the program code. The coordinates must be retrieved from the window record. The process is somewhat indirect. First, a handle to the record that defines the content region is pulled from the window record. (We are not interested in updating the structure region.) That handle is de-referenced to get a pointer. The actual rectangle begins two bytes past the address contained in the pointer; as discussed in Chapter 7, the first two bytes of a region record contain the size of the region. Therefore, if the quantity 2 (for two bytes) is added to the pointer to the region record, we will have the address of the start of the region’s boundary rectangle, which in turn can be passed to `EraseRect`.

Once the current contents of the window have been erased, the application should redraw the contents as appropriate. This may involve calls to QuickDraw routines or to special TextEdit updating routines (see Chapter 9).

The application must then update the scroll bars. First, the existing scroll bars must be hidden:

```plaintext
PROCEDURE HideControl (theControl: ControlHandle);
```

In order to write general code that will apply to any of an application’s windows, there has to be some way to retrieve the handles for a particular window’s controls. This can be done by using both the window record and the records of any controls associated with the window. As discussed in Chapter 7, the window record maintains a handle to the first control in its control list. Each control record maintains a handle to the next item in the control list.

There is no field in a control record that explicitly indicates what type of control that record represents. Therefore, as mentioned in Chapter 7, the reference value field can be used for an application assigned code to identify control types. In the example below, a horizontal scroll bar was arbitrarily given a reference value of 1 and a vertical scroll bar a reference value of 2; other controls were given higher values.
The "next step" is to actually move the control:

PROCEDURE MoveControl (theControl: ControlHandle; h,v: INTEGER);

The location to which this routine moves a control is specified by giving the control new top left coordinates in the local coordinate system of its window. h is the horizontal (left) coordinate and v the vertical (top). An application can determine these coordinates from the bottom right coordinates returned by GrowWindow. First Growwindow's global coordinates are converted to local coordinates. Then, for a horizontal scroll bar, the vertical coordinate will be 16 less than the the bottom. (The horizontal coordinate will remain at 0.) For a vertical scroll bar, the vertical coordinate will remain 0 and the horizontal coordinate will be 16 less than the right. Assume that GrowWindow's result is in D0 and that the handle to a control record for a scroll bar is in A0. Remember that this example has arbitrarily assigned vertical scroll bars a reference value of 2 and horizontal scroll bars a reference value of 1. Also note that the two coordinates returned by GrowWindow must be stored in RAM so their address can be passed to GlobalToLocal:

MOVE.L D0,Points(A5) ;store coordinates in RAM
PEA Points(A5)
_GlobalToLocal ;convert the coordinates
MOVE Points(A5),D0 ;recover local coordinates
MOVE D0,D1 ;get low order word (right)
SWAP D0 ;flip the words in the register
MOVE D0,D2 ;get high order word (bottom)
MOVE.L (A0),A1 ;pointer to control record
MOVE.L ctrlRfCon(A1),D7 ;control value

AnotherControl
MOVE.L nextCtrl(A1),A0 ;handle to next control
BEQ NextStep ;no more controls
MOVE.L (A0),A1 ;get pointer
MOVE.L controlRfCon(A1),D7 ;get reference value
CMP #2,D7 ;not a scroll bar
BGT AnotherControl
MOVE.L AO, -(SP) ;push handle on stack
_HideControl
BRA AnotherControl

AnotherControl
MOVE.L AO, -(SP) ;push handle on stack
_HideControl

The "next step" is to actually move the control:

PROCEDURE MoveControl (theControl: ControlHandle; h,v: INTEGER);

The location to which this routine moves a control is specified by giving the control new top left coordinates in the local coordinate system of its window. h is the horizontal (left) coordinate and v the vertical (top). An application can determine these coordinates from the bottom right coordinates returned by GrowWindow. First Growwindow's global coordinates are converted to local coordinates. Then, for a horizontal scroll bar, the vertical coordinate will be 16 less than the the bottom. (The horizontal coordinate will remain at 0.) For a vertical scroll bar, the vertical coordinate will remain 0 and the horizontal coordinate will be 16 less than the right. Assume that GrowWindow's result is in D0 and that the handle to a control record for a scroll bar is in A0. Remember that this example has arbitrarily assigned vertical scroll bars a reference value of 2 and horizontal scroll bars a reference value of 1. Also note that the two coordinates returned by GrowWindow must be stored in RAM so their address can be passed to GlobalToLocal:

MOVE.L D0,Points(A5) ;store coordinates in RAM
PEA Points(A5)
_GlobalToLocal ;convert the coordinates
MOVE Points(A5),D0 ;recover local coordinates
MOVE D0,D1 ;get low order word (right)
SWAP D0 ;flip the words in the register
MOVE D0,D2 ;get high order word (bottom)
MOVE.L (A0),A1 ;pointer to control record
MOVE.L ctrlRfCon(A1),D7 ;control value

AnotherControl
MOVE.L nextCtrl(A1),A0 ;handle to next control
BEQ NextStep ;no more controls
MOVE.L (A0),A1 ;get pointer
MOVE.L controlRfCon(A1),D7 ;get reference value
CMP #2,D7 ;not a scroll bar
BGT AnotherControl
MOVE.L AO, -(SP) ;push handle on stack
_HideControl
BRA AnotherControl

AnotherControl
MOVE.L AO, -(SP) ;push handle on stack
_HideControl


CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS

CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS

```
CMP     #1,D7          ;this is a vertical control bar
BNE     Vertical       
SUB     #16,D2         ;adjust vertical coordinate
MOVE    #0,D1          ;horizontal coordinate stays 0
BRA     RoutineSetUp   

RoutineSetUp

MOVEL   A0, -(SP)      ;push handle onto stack
MOVE    D1, -(SP)      ;horizontal coordinate goes first
MOVE    D2, -(SP)      ;vertical coordinate

MOVE    D1, -(SP)      ;horizontal coordinate goes first
MOVE    D2, -(SP)      ;vertical coordinate

 появила постоянно волшебника моря ставить

PROCEDURE SizeControl (theControl: ControlHandle; w, h: INTEGER);

The parameters w and h correspond respectively to the new right and bottom
coordinates of the control, expressed again in the local coordinates of the control's
window. Neither the horizontal nor vertical scroll bars can use the coordinates
returned by GrowWindow directly. A horizontal scroll bar will have a w value of 15
less than that returned by GrowWindow to allow space for the grow icon; h will not
need to be altered. A vertical scroll bar can accept w from GrowWindow's result
but must subtract 15 from the horizontal coordinate. Assume again that
GrowWindow's result is in D0 (now in local coordinates since they were converted
before the call to MoveWindow) and that the handle to a control is in A0:

MOVE    D0,D1          ;get horizontal coordinate
SWAP    D0             ;exchange register halves
MOVE    D0,D2          ;get vertical coordinate
MOVE    (A0),A1        ;pointer to control record
MOVEL   contrRfCon(A1),D7 ;reference value
CMP     #1,D7          ;horizontal or vertical scroll bar?
BNE     Vertical2      ;must be vertical
SUB     #15,D1         ;adjust horizontal coordinate
BRA     PassParameters 

Vertical2

SUB     #15,D2         ;adjust vertical coordinate
```
PassParameters

MOVE.L A0, - (SP) ; put handle on stack
MOVE D1, - (SP) ; horizontal coordinate goes first
MOVE D2, - (SP) ; vertical is next

Proc SizeControl

To be on the safe side, an application should follow SizeControl with a call to
ShowControl, to be sure that the control is displayed:

PROCEDURE ShowControl (theControl: ControlHandle);

If the control is invisible, ShowControl will make it visible. If the control is already
visible, ShowControl will have no effect.

Finally, the update process should be completed with a call to EndUpdate.
Code for these last two steps might appear as:

MOVE.L A0, - (SP) ; control handle in A0
Proc ShowControl

MOVE.L WhichWindowPtr, - (SP)
Proc EndUpdate

----------

Handling Key Down
Events

Most applications will have a variety of uses for key down events. Primarily, they can be used to display text on the screen or, in combination with the cloverleaf key, they can substitute for using the mouse to make a menu selection.

The first step in processing a key down event is therefore to determine which, if
any, of the modifier keys were held down in conjunction with the key press. The
modifier flags are stored in the high-order byte of the Modify field from the event
record. If the cloverleaf key was pressed, bit 0 of that byte will be set (i.e., the byte
will have a value of 1). This gives the entire modifier word, by the way, a value of
256.

To test for the cloverleaf key, an application could:

MOVE.B Modify, D0 ; get high-order byte only
CMP.B #1, D0 ; compare with cloverleaf value
BEQ KeyboardCommand ; branch to handle command
The same result could also be obtained by:

```
MOVE Modify, D0 ;get entire modifier word
CMP #256, D0 ;compare with cloverleaf value
BEQ KeyboardCommand
```

If `Modify` does reveal that the cloverleaf key was pressed along with some other key, then the application can assume that it was intended to be a substitution for a mouse selection from a menu. The information needed to process such a selection is identical to that needed to process a mouse down menu selection — which menu and which item within that menu.

`MenuKey` is a single routine that will return both the menu and the item numbers corresponding to the keyboard equivalent selected by a cloverleaf command. It assumes that a press of the cloverleaf key has already been detected and therefore only needs to know what additional character was pressed at the same time. That character can be found as part of the event record's `Message` field. More precisely, it is in the low-order word of that field. The address of the character is therefore two bytes beyond the beginning of `Message` and can be indicated by using `Message + 2` as the effective address.

`MenuKey`'s format is:

```
FUNCTION MenuKey (ch: CHAR) : LONGINTEGER;
```

It should be called by using:

```
CLR.L - (SP) ;space for longinteger result
MOVE Message + 2, - (SP) ;put character on the stack
  MenuKey
```

`MenuKey` returns its result in exactly the same format as `MenuSelect`. The menu number is in the high-order word and the item number in the low-order word. Therefore, immediately after issuing the call to `MenuKey`, an application can branch to join the same processing sequence that occurs after `MenuSelect`, including retrieving the result and unhighlighting the menu title.

Key down events that are not accompanied by the cloverleaf key generally indicate text that should be displayed on the screen by TextEdit. Processing of these events is discussed in Chapter 9.

---

**Handling Update Events**

An update event is posted to the event queue whenever a window is newly activated or when something has occurred to disrupt the display of the window's
contents. The latter is usually the result of one window overlaying another. For example, if you open a desk accessory while in the midst of one of the Video Tape Index's functions that permit text editing, the desk accessory's window has the potential of covering one or more of the text entry windows and their prompts (which are actually displayed on the main window). Any window which covers another erases the contents of the windows underneath it. When the front window is closed or moved to the back, any windows which are uncovered will be minus their contents. An update event alerts the application that the newly exposed windows need to have their contents redrawn.

Updating in text windows is handled by a special TextEdit routine. Updating the contents of other windows can be done in two ways: either by drawing only the region of the window that needs updating, or by erasing the entire window and redrawing all of its contents. The latter is far easier to implement.

Regardless of whether you are updating text windows or other windows, all update processes must start with a call to **BeginUpdate** and finish with a call to **EndUpdate**. These two routines manage a portion of the window known as the *update region*. Update regions are important if you are attempting to do an update by redrawing only that portion of a window's contents that have been erased. Even if the update will erase the window and completely redraw it, the process must nonetheless be bracketed by these two routines.

Each takes the pointer of the window to be updated as a parameter:

```assembly
PROCEDURE BeginUpdate (theWindow: WindowPtr);
PROCEDURE EndUpdate (theWindow: WindowPtr);
```

The easiest way to update a non-text window is therefore to:

1. Call **BeginUpdate**
2. Erase the window (using **EraseRect**)
3. Redraw the window's contents (procedure will vary with the window in question)
4. Call **EndUpdate**

Since the prompts for the Video Tape Index's text entry windows are drawn on the main window, any time a desk accessory is opened while one of the four text entry functions are in process, those prompts will be disturbed. Immediately after the desk accessory is closed, the main window must therefore be updated. The code to do so appears as:

```assembly
MOVE.L MainWindowPtr, -(SP) ;window pointer
__BeginUpdate
MOVE.L MainWindowPtr, -(SP) ;start update process
__SetPort
```

...
CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS

PEA MainWindowRect ;boundary rectangle
_EraseRect ;erase the window
JSR DisplayPrompts ;redraw window contents
MOVE.L MainWindowPtr, -(SP) ;window pointer
_EndUpdate ;finish update process

Note that update events that occur while the program is in its main event loop can be ignored since the main window has no content at that point.

A Word About Activate Events

In most cases, activate events can be ignored. Since SelectWindow handles highlighting and unhighlighting windows, usually the only time an application needs to respond to an activate event is when a text edit window has been selected. Activating and deactivating text entry windows takes care of respectively displaying and removing the straight-line cursor (see Chapter 9).

Pulling Things Together Thus Far — WindowPlay

In Listing 8.4 you will find the source code for a demonstration program called WindowPlay. Its resource file appears in Listing 8.5. This program uses many of the concepts and techniques presented in Chapters 7 and 8, including defining windows and menus and trapping a variety of events. It is very short for a complete Macintosh assembly language application and really doesn't do any useful work, but it does illustrate how to create and manipulate windows and menus.

WindowPlay has a template for one window of each window type in its resource file. The windows can be displayed by selecting the window type from the Windows menu, creating a display like the one in Figure 8.3. The windows overlap on the screen, but their position in the plane can be changed by clicking on a window with the arrow cursor. Those windows which support title bars can be closed by clicking in their GoAway boxes. Any active (frontmost) window can be closed by selecting Close from the File menu.
Listing 8.4 WindowPlay

Include MacTraps.D
Include ToolEqu.D
Include SysEqu.D

PEA -4(A5)
_InitGraf ;initialize QuickDraw
_InitFonts ;initialize Font Manager
MOVE.L #0000FFFF,D0
_FlushEvents ;flush all events from event queue
_InitWindows ;initialize Window Manager
_InitMenus ;initialize Menu Manager
_CLR.L -(SP)
_InitDialogs ;initialize Dialog manager
_TEInit ;initialize Text Edit
_InitCursor ;get arrow cursor

_CLR -(SP)
PEA 'MAL.files:WindowPlay.Rsrc'
_OpenResFile ;open the resource file
MOVE (SP)+,D0 ;discard unused result

; ----------------- Set up the menus -------------------------------
_CLR.L -(SP) ;space for handle
_MOVE #1,-(SP) ;menu sequence number
_GetRMenu ;get Apple menu template
MOVE.L (SP)+,AppleHandle(A5) ;retrieve and store handle
MOVE.L AppleHandle(A5),-(SP) ;put handle back on stack
_ADDResMenu ;resource type for desk accessories
MOVE.L #DRVR',-(SP) ;get desk accessories
MOVE.L AppleHandle(A5),-(SP) ;handle back on stack
_CLR -(SP) ;this menu goes after all others
_InsertMenu ;put menu in menu list

_CLR.L -(SP)
_MOVE #2,-(SP)
_GetRMenu
MOVE.L (SP)+,FileHandle(A5)
MOVE.L FileHandle(A5),-(SP)
_CLR -(SP)
_InsertMenu

_CLR.L -(SP)
_MOVE #3,-(SP)
_GetRMenu
MOVE.L (SP)+,EditHandle(A5)
MOVE.L EditHandle(A5),-(SP)
_CLR -(SP)
_InsertMenu

_CLR.L -(SP)
_MOVE #4,-(SP)
_GetRMenu
MOVE.L (SP)+,WindowHandle(A5)

(continued)
CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS

MOVE.L WindowHandle(A5),-(SP)
CLR -(SP)
_InsertMenu

_DrawMenuBar ;finally, make it all appear

MOVE #Ø,DØ ;initialize window counter

--------------------------Event loop comes next to control actions --------------------------
Event _SystemTask ;update desk accessories

CLR -(SP) ;space for boolean result
MOVE #1,-(SP) ;mask to select all events
PEA EventRecord(A5) ;address of event record
_GetNextEvent ;retrieve event from queue

MOVE (SP)+,DØ ;recover result
CMP #Ø,DØ ;did event occur?
BEQ Event ;no event

MOVE EventRecord(A5),DØ ;this retrieves 1st word of record - the event type

CMP #mButDwnEvt,DØ ;mouse button pressed?
BEQ MouseEvent

CMP #keyDwnEvt,DØ ;key pressed?
BEQ KeyEvent

BRA Event ;not an event this program handles

; ------------------------------- Handle key down events -------------------------------
KeyEvent

MOVE EventRecord+evtMeta(A5),DØ ;get modify word
BTST #cmdKey,DØ ;cloverleaf key held down?
BEQ Event ;not a menu selection

CLR.L -(SP) ;space for menu item selection
MOVE EventRecord+evtMessage+2(A5),-(SP) ;put character pressed on stack
_MenuKey ;identify menu and item

BRA Selections ;join menu processing

; ------------------------------- Handle mouse down events -------------------------------
MouseEvent

CLR -(SP) ;space for "what" result
MOVE.L EventRecord+evtMouse(A5),-(SP) ;place where event occurred
PEA WhichWindowPtr(A5) ;window affected goes here
_FindWindow ;determine which window posted event

MOVE (SP)+,DØ ;recover result

CMP #inMenuBar,DØ ;mouse down event in menu bar
BEQ MenuBar

CMP #inSysWindow,DØ ;mouse down event in system window
BEQ SysEvent

CMP #inContent,DØ ;mouse down event in application window
BEQ ApplWindow

(continued)
Listing 8.4 (continued)

CMP  #inGoAway,D0   ;mouse down event in GoAway box
BEQ  CloseWindow   ;not a place this program monitors
BRA  Event

; -------------------------------- Mouse down event in system window ----------------------------------
SysEvent
PEA  EventRecord(A5)   ;address to event record on stack
MOVE.L  WhichWindowPtr(A5),-(SP)   ;window posting event
    SystemClick   ;system does all the work
BRA  Event   ;get another event

; -------------------------------- Mouse down event in menu bar ------------------------------------
MenuBar
    CLR.L  -(SP)   ;space for menu ID and menu item
    MOVE.L  EventRecord+evtMouse(A5),-(SP)   ;place where mouse button went down
    _MenuSelect   ;find menu number and menu item

Selections
    MOVE.L  (SP)+,D0   ;recover result
    MOVE  D0,D1   ;D1 now has low-order word (menu item)
    SWAP  D0   ;menu ID now in low-order word of D0
    MOVEM.L  D0/D1,-(SP)   ;save registers
    CLR  -(SP)   ;selects all menus
    _HiLiteMenu   ;remove highlighting from menu
    MOVEM.L  (SP)+,D0/D1
    CMP  #1,D0   ;in Apple menu?
    BNE  Menu2
    BRA  AppleMenu   ;handle desk accessories
Menu2  CMP  #2,D0   ;in File menu?
    BNE  Menu3
    BRA  FileMenu
Menu3  CMP  #3,D0   ;in Edit menu?
    BNE  Menu4
    BRA  EditMenu
Menu4  CMP  #4,D0   ;in Window menu?
    BNE  Event   ;something weird happened...
    BRA  WindowEvent

--------------------------------------- Handle desk accessories ---------------------------------------
AppleMenu
    MOVE.L  AppleHandle(A5),-(SP)   ;menu handle on stack
    MOVE  D1,-(SP)   ;menu item on stack
    PEA  DeskAccName(A5)   ;space for desk accessory name
    _GetItem   ;retrieve name of desk accessory
    CLR  -(SP)   ;space for reference number
    PEA  DeskAccName(A5)   ;point to desk accessory name
    _OpenDeskAcc   ;open the desk accessory
    MOVE  (SP)+,D0   ;discard reference number result
    BRA  Event

(continued)
CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS 227

:----------------------------- Handle editing in desk accessories -----------------------------

EditMenu

SUBQ #1,D1 ;adjust item selected for SysEdit
CLR -(SP) ;space for problem result
MOVE D1,-(SP) ;adjusted item number goes on stack
_SysEdit ;let system handle to edit
MOVE (SP)+,DØ ;get rid of result
BRA Event

:---------------------------- Handle File Menu -----------------------------

FileMenu

CMP #1,D1 ;Close the active window?
BEQ WindowClose
CMP #2,D1 ;Quit?
BNE Event

RTS ;This returns to the Finder

WindowClose

CLR.L -(SP) ;space for pointer to active window
_FrontWindow ;get pointer
MOVE.L (SP)+,A6 ;save pointer

MOVE.L A6,-(SP) ;put pointer back on stack
_CloseWindow ;close the window

SUBQ #1,D7 ;decrement window counter
BNE Fix
MOVE.L FileHandle(A5),-(SP)
MOVE #1,-(SP) ;if no windows present, disable Close

Fix CMP.L Window1Ptr(A5),A6 ;identify which window was closed
BNE Fix2
MOVE #1,D1
BRA ReEnable

Fix2 CMP.L Window2Ptr(A5),A6
BNE Fix3
MOVE #2,D1
BRA ReEnable

Fix3 CMP.L Window3Ptr(A5),A6
BNE Fix4
MOVE #3,D1
BRA ReEnable

Fix4 CMP.L Window4Ptr(A5),A6
BNE Fix5
MOVE #4,D1
BRA ReEnable

Fix5 CMP.L Window5Ptr(A5),A6
BNE Fix6
MOVE #5,D1
BRA ReEnable

(continued)
Listing 8.4 (continued)

Fix6
MOVE #6, D1

ReEnable
MOVE.L WindowHandle(A5),(SP) ; handle to menu
MOVE D1,(SP) ; window #
_EnableItem ; turn the menu item back on
BRA Event

; ---------------------------------------------------------- Handle Window Menu ----------------------------------------------------------
WindowEvent
MOVE D1,(SP) ; save register contents
MOVE.L WindowHandle(A5),(SP)
MOVE D1,(SP) ; window number same as menu item #
_DisableItem ; turn off this window
MOVE (SP)+, D1

CMP #1, D1 ; window 1?
BNE Window2
CLR.L -(SP) ; space for window handle
MOVE #1,(SP) ; window ID
PEA Window1Strg(A5) ; pointer to window record
MOVE.L #1,(SP) ; put window in front
_GetNewWindow ; create the window
MOVE.L (SP)+, Window1Ptr(A5) ; retrieve the pointer
BRA WindowCount

Window2
CMP #2, D1 ; repeat for all windows
BNE Window3
CLR.L -(SP)
MOVE #2,(SP)
PEA Window2Strg(A5)
MOVE.L #1,(SP)
_GetNewWindow
MOVE.L (SP)+, Window2Ptr(A5)
BRA WindowCount

Window3
CMP #3, D1
BNE Window4
CLR.L -(SP)
MOVE #3,(SP)
PEA Window3Strg(A5)
MOVE.L #1,(SP)
_GetNewWindow
MOVE.L (SP)+, Window3Ptr(A5)
BRA WindowCount

Window4
CMP #4, D1
BNE Window5
CLR.L -(SP)
MOVE #4,(SP)
PEA Window4Strg(A5)
MOVE.L #1,(SP)
_GetNewWindow
MOVE.L (SP)+, Window4Ptr(A5)
BRA WindowCount

(continued)
CONTROLLING PROGRAM ACTIONS: MONITORING EVENTS

Window5
CMP #5,D1
BNE Window6
CLR.L -(SP)
MOVE #5,-(SP)
PEA Window5Strg(A5)
MOVE.L #-1,-(SP)
_GetNewWindow
MOVE.L (SP)+,Window5Ptr(A5)
BRA WindowCount

Window6
CLR.L -(SP)
MOVE #6,-(SP)
PEA Window6Strg(A5)
MOVE.L #-1,-(SP)
_GetNewWindow
MOVE.L (SP)+,Window6Ptr(A5)

WindowCount
ADDO #1,D7
CMP #1,D7
BNE Done

;count number of windows on screen

MOVE.L FileHandle(A5),-(SP)
MOVE #1,-(SP)
_EnableItem

;first window, enable

Done
BRA Event

;handle to window menu

Close
CLR.B -(SP)
MOVE.L WhichWindowPtr(A5),-(SP)
_BSelectWindow

;bring window to front & make active

BRA Event

;handle mouse down in application window

AppWindow
MOVE.L WhichWindowPtr(A5),-(SP)
_BSelectWindow

;space for boolean result

CloseWindow
CLR.B -(SP)
MOVE.L WhichWindowPtr(A5),-(SP)
MOVE.L EventRecord+evtMouse(A5),-(SP)
_TrackGoAway

;window posting event

Close
MOVE.B (SP)+,D0
CMP #0,D0
BEQ Event
BRA WindowClose

;get result

;did user change mind?

;don't close

;close window just like menu selection

;data structures

AppleHandle DS.L 1
EditHandle DS.L 1
FileHandle DS.L 1
WindowHandle DS.L 1

Window1Ptr DS.L 1
Window2Ptr DS.L 1
Window3Ptr DS.L 1
Window4Ptr DS.L 1
Window5Ptr DS.L 1
Window6Ptr DS.L 1

(continued)
Listing 8.4 (continued)

Window1Strg  DS  windowSize  ;storage for window records
Window2Strg  DS  windowSize
Window3Strg  DS  windowSize
Window4Strg  DS  windowSize
Window5Strg  DS  windowSize
Window6Strg  DS  windowSize

WhichWindowPtr  DS.L  1  ;for FindWindow result
DeskAccName  DS  16  ;for desk accessory name

EventRecord  DS.B  16

Listing 8.5  Resource File for WindowPlay

WindowPlay.Rsrc

TYPE MENU
,.1
\14  ;; Apple menu

,.2
File
(Close
Quit/Q

,.3
Edit
 Undo/Z
(-
 Cut/X
Copy/C
Paste/V
Clear

,.4
Windows
documentProc
dBoxProc
plainDBox
altDBoxProc
noGrowDocProc
rDocProc

TYPE WIND
,.1
Sample Window
40 160 300 480
visible GoAway
Ø
Ø

;; Window selection menu

;; standard document window

(continued)
WindowPlay supports four menus: a standard Apple menu for the desk accessories, a File menu, an Edit menu (for the desk accessories only), and the application menu Windows that controls which windows appear on the screen.

The structure of WindowPlay is typical of a Macintosh assembly language program. The set-up process involves:

1. Initialization of all ToolBox and operating system managers
2. Opening the resource file
3. Reading menu templates from the resource file and creating the menu bar
4. Entering an event loop

The event loop itself determines the structure of the remainder of the program. WindowPlay looks for mouse down and key down events. Since there is no text editing, key down events are meaningful only as keyboard equivalents for menu
selections. Menu selections represent either a request for a desk accessory, a request to return to the Finder, or a request to affect one of the windows displayed by the application.

When WindowPlay is launched, the screen is blank and the Close option of the File menu is dimmed. (It makes no sense to allow the user to close a window when no windows are visible.) WindowPlay keeps track of how many windows are displayed and always disables Close when the count drops to 0; it enables Close when the count rises to 1.

You will also notice that whenever a window is selected for display, its name in the Windows menu is disabled. This ensures that only one window of any given type will be displayed at any given time. When the window is closed, its name in the Windows menu is re-enabled.

Questions and Problems

1. Create binary event masks to select the following events:
   a. mouse down, mouse up
   b. mouse down, key down
   c. mouse down, key down, update, activate, disk insertion
2. Write an event loop that:
   A. retrieves events from the event queue with an event mask that selects all
      events
   B. checks for mouse down, key down, update, activate and disk insertion
      events
   C. branches to an appropriately named subroutine to handle each type of
      event

   Be sure to allocate any data structures your event loop will use.

3. Write an ordered list of the ToolBox and/or operating system routines that
   must be used to identify a user request for the note pad desk accessory.
   Indicate the information returned by each call. Assume that an event loop has
   already detected a mouse down event and that the Apple menu is Menu
   number 0.

4. Write a block of assembly language code to implement the procedure you
   outlined in problem 3. Use the event record field names defined in Chapter 8.
   Allocate any other data structures your code will use.

5. Write an ordered list of the ToolBox and/or operating system routines that
   must be used to identify a user request to scroll the text in a document
   window one page up. Indicate the information returned by each call. Assume
   that an event loop has already detected a mouse down event.

6. Write a block of assembly language code to implement the procedure you
   outlined in problem 5. Use the event record field names defined in Chapter 8.
   Allocate any other data structures your code will use.

7. Write an ordered list of the ToolBox and/or operating system routines that are
   needed to identify which menu item has been selected by a combination
   cloverleaf-alphanumeric key press. Indicate the information returned by
   each call. Assume that an event loop has already detected a key down event.

8. Write a block of assembly language code to implement the procedure you
   outlined in problem 7. Use the event record field names defined in Chapter 8.
   Allocate storage space for any other data structures your code will use.
9. Code and implement the following modifications to the program WindowPlay from Listings 8.4 and 8.5:

A. Draw scroll bars and a grow icon in the standard document window. The scroll bars should be defined in the resource file.

B. Trap for events in a title bar so that the three windows with title bars can be moved about the screen.

C. Handle moving the three windows with title bars, including the scroll bars in the standard document window.

D. Trap for events in a grow icon.

E. Handle sizing the standard document window.
Chapter Objectives

1. To understand the data structures that support Macintosh text editing
2. To learn to establish those data structures
3. To learn to manage windows that support text editing
4. To learn to implement text editing functions: entering text, deleting text, cut, paste, and copy
5. To learn to manage text characteristics such as font size and type
6. To learn to create the resource file templates that establish alerts and dialog boxes
7. To learn to use alerts and dialog boxes to control program actions

Entering, Displaying, and Editing Text

TextEdit is a collection of powerful ROM routines that permit the easy entry and editing of text. Text is written into a rectangle defined for that purpose. The rectangle may be an entire window or only part of a window.
To be precise, editing text requires two rectangles — a destination rectangle and a view rectangle. The destination rectangle establishes the bounds in which the text should be drawn; the view rectangle defines the area in which the text will be seen. Though both must be specified, they are usually identical. Destination and view rectangles are defined in the local coordinates of the grafport in which the text will appear. In other words, in order to enter or edit text, the window that contains the destination and view rectangles must be the current grafport (set with the SetPort routine).

Information about the editing environment is kept in an edit record. An edit record contains, in part, the coordinates of the destination and view rectangles, font and text justification information, the current selection range, the length of the text, a handle to where the text is stored, the number of lines in the text, and positions of line starts within the text. The text itself is stored as a packed array of characters (i.e., each ASCII character code occupies only one byte).

The structure of a text edit record is:

```
TeRec = RECORD
  destRect: Rect; the destination rectangle
  viewRect: Rect; the view rectangle
  selRect: Rect; boundaries of selection range
  lineHeight: INTEGER; height of a line of text
  fontAscent: INTEGER; number of pixels a font rises
  selPoint: Point; location of mouse button click
  selStart: INTEGER; start of selection range
  selEnd: INTEGER; end of selection range
  active: INTEGER; used internally — do not change
  wordBreak: ProcPtr; used to change how TextEdit views the end of a word
  clikkLoop: ProcPtr; used to implement automatic scrolling
  clickTime: LONGINT; used internally — do not change
  clickLoc: INTEGER; used internally — do not change
  caretTime: LONGINT; used internally — do not change
  caretState: INTEGER; used internally — do not change
  just: INTEGER; text justification
  telLength: INTEGER; text length in # of characters
  hText: Handle; handle to the text itself
  recalBack: INTEGER; used internally — do not change
  recalLines: INTEGER; used internally — do not change
  clikStuff: INTEGER; if negative, indicates that a new line should start only after a CR
  crOnly: INTEGER;
  txFont: INTEGER; font ID number
  txFace: Style; text style (e.g., bold or italic)
  txMode: INTEGER; pen mode
  txSize: INTEGER; font size
```

As with the other types of records we've discussed, an application won't need to retrieve data from most fields of the edit record directly. There are, however, some exceptions. A program may, for example, need the length of the text or the handle which contains the pointer to the text itself. Printing characters from a text edit record requires knowing how many lines of text there are and where each line begins. Equates for field offsets into an edit record are part of the ToolBox equates file. For example, teLength refers to a file $3E$ bytes in from the beginning of an edit record. If ToolBox.D is INCLUDEd at the beginning of an application's source code, the offsets to all fields in the edit record are available to the program.

Before using any TextEdit routines, you must initialize the manager with TEInit. This procedure takes no parameters. It should appear at the top of an application, along with the other initializations; TEInit can be the last call in the initialization sequence. Because text manipulation also often involves manipulating font characteristics, the initialization sequence should also include a call to InitFonts, which initializes the Font Manager.

A complete initialization sequence, one that will be complete enough for most applications, appears as follows:

```
PEA _lnitGraf _lnitFonts _lnitWindows _lnitMenus CLR.L -(SP) _lnitDialogs _TEInit _lnitCursor
```

;initialize QuickDraw
;initialize the Font Manager
;initialize the Window Manager
;initialize the Menu Manager
;initialize the Dialog Manager (discussed below)
;initialize TextEdit
;get arrow cursor

Use this block of code exactly as it appears. Because the various managers interact so closely, it is imperative that the initializations are performed in this order.

Data Structures for Text Edit

Windows for text editing should be defined in a resource file, just like any other window. They can then be created with GetNewWindow. Text edit windows can also be manipulated like other windows in terms of visibility and position on the

<table>
<thead>
<tr>
<th>inPort:</th>
<th>GrafPtr;</th>
<th>pointer to grafport</th>
</tr>
</thead>
<tbody>
<tr>
<td>highHook:</td>
<td>ProcPtr;</td>
<td>used by low-level routines</td>
</tr>
<tr>
<td>caretHook:</td>
<td>ProcPtr;</td>
<td>used by low-level routines</td>
</tr>
<tr>
<td>nLines:</td>
<td>INTEGER;</td>
<td>number of lines of text</td>
</tr>
<tr>
<td>lineStarts</td>
<td>ARRAY [0..16000] of INTEGER</td>
<td>positions of starts of lines</td>
</tr>
</tbody>
</table>
screen. Doing the editing, though, requires the destination and view rectangles mentioned above.

In most cases, the destination and view rectangles will cover an entire window. (The major exception is in dialog boxes, which will be discussed at the end of this chapter.) Therefore, to determine the coordinates of the destination and view rectangles, you need only figure out how many pixels the window encompasses. Consider, as an example, the Video Tape Index’s text edit windows.

The Video Tape Index uses one text window for each field in the TapeArray record. (For database applications this tends to simplify the text handling.) The Producer window, for example, has global coordinates of 75, 240, 95, 415. These coordinates appear in the program’s resource file. The height of the window is therefore 21 pixels (bottom – top + 1) and its width 176 pixels (right – left + 1). Since the view and destination rectangles will occupy the entire window, they could theoretically have coordinates of 0, 0, 21, 176. In practice, though, the rectangles should come in at least one pixel from the edges of the windows. Otherwise, it’s possible that the first and last characters will not be displayed completely. Therefore, the destination and view rectangles for the Producer text window have coordinates of 1, 1, 19, 175. These coordinates are defined within the source code:

```
ProducerViewRect   DC   1,1,19,175
ProducerDestRect   DC   1,1,19,175
```

Storage must also be set aside for a handle to each text window’s edit record. You need not reserve space for the edit record itself; this will be done by the system when the record is created. The handle to an edit record has the data type TEHandle and therefore requires a longword of space:

```
ProducerTextHandle   DC.L  0
```

or

```
ProducerTextHandle   DS.L  1
```

### Allocating Text Edit Records

Text edit records are allocated by the routine `TENew`:

```
FUNCTION TENew (destRect, viewRect: Rect) : TEHandle;
```

The addresses of the destination and view rectangles are pushed onto the stack. A longword handle is returned.

The above function looks simple. There is something important, however, to be aware of when allocating edit records. An edit record includes the grafport of the
editing environment in its `inPort` field; the `TENew` routine will automatically absorb whatever grafport is current at the time the call to `TENew` is placed. Therefore, prior to allocating an edit record, the window for the destination and view rectangles must have been created. Immediately before calling `TENew`, `SetPort` must be used to make that window the current grafport.

The Video Tape Index defines all of its windows and only then creates text edit records. A typical code sequence, assuming the window has been defined and its pointer saved, is:

```assembly
MOVE.L producerWindowPtr, -(SP) ; window is current port
_CLR.L -(SP) ; space for edit record handle
PEA producerDestRect
PEA producerViewRect
_TENew
LEA NameTextHandle,A0 ; address for handle storage
MOVE.L (SP)+,A0 ; retrieve the handle and store
```

Managing Text Edit Windows

Whenever a text edit window is activated (e.g., a mouse down event was recorded somewhere in a deactivate text edit window), an application generally responds by calling `SelectWindow`. As mentioned in Chapter 8, `SelectWindow` generates an activate event for the window selected and a deactivate event for the previously active window.

Activating and deactivating text edit windows is important, since these actions control the appearance and disappearance of the straight-line cursor. Therefore, whenever an application detects an activate event in a text edit window, the event should be handled, not ignored.

TextEdit provides two routines to do the activating and deactivating: `TeDeActivate` removes the straight-line cursor; `TEActivate` makes the straight-line cursor appear at the left-most position of the activated view rectangle. (It does not make the cursor blink.)

The same event type is returned for both activate and deactivate events. An application can distinguish between the two by checking the `Modify` field of the event record. For example:

```assembly
MOVE modify,D0 ; retrieve Modify field
BTST #activeFlag,D0 ; is activate bit set?
BEQ DeActivate ; bit is not set — event is deactivate
```

The instruction `BTST` (for Bit Test) is handy when you need to determine whether or not a specific bit has been set within a register. `activeFlag` is defined in the System equates file as 0, which corresponds to the bit position of the flag which
is set if an activate event really corresponds to activating a window, and cleared if it means the window was deactivated. **BTST** works by looking at the specified bit number in the specified register. If the bit is set, the zero flag in the status register will be set; if the bit is cleared, the zero flag will be cleared. Therefore, an activate event will set the zero flag and a deactivate event will clear it.

In an environment like the Video Tape Index where there is more than one text edit record, an application must also identify which of the text edit windows posted a given event. Remember that for activate events, the **Message** field of the event record will contain the pointer to the window posting the event. Therefore, you need only compare each window pointer in turn with **Message** to identify the correct window.

The actual activating and deactivating of text edit windows is very straightforward, requiring only the handle to the edit record:

```
PROCEDURE TEActivate (hTE: TEHandle);
PROCEDURE TEDeactivate (hTE: TEHandle);
```

Move the handle onto the stack and then call the routine.

The Video Tape Index program's subroutine for detecting which window has posted an activate event and properly handling that event, **ActivateTextWindow**, appears in Listing 9.1. The first step is to retrieve the pointer of the window involved from the event record. This occurs at (a) in Listing 9.1. The application also needs to determine whether the window should be activated or deactivated. Therefore, the modify word is also retrieved from the event record (b). As discussed above, the activate or deactivate decision is based on the value of bit 0 in the modify word. A **BTST** instruction (c) can be used to check the value of the appropriate bit, which is equated to the symbolic address of **activeFlag**. If **activeFlag** has been cleared, then the window should be deactivated (d). The application branches to a block of code that specifically handles deactivations (k).

**Listing 9.1 Activating Text Edit Windows**

```
ActivateTextWindow
(a)   MOVE. L Message,A0          ;get pointer to window which posted event
(b)   MOVE. Modify,D0
(c)   BTST #activeFlag,D0         ;activate bit set?
(d)   BEQ  DeActivate            ;if not set, window was deactivated

Activate1
(e)   CMP. L NameWindowPtr,A0    ;name window event?
(f)   BNE   Activate 2
(g)   MOVE. L NameTextHandle,-(SP)
(h)   TEActivate                
     BRA   Activate99

Activate2
(i)   CMP. L ProducerWindowPtr,A0
     BNE   Activate3
```

(continued)
SCREEN AND KEYBOARD I/O: USING TEXTEDIT, ALERT AND DIALOG BOXES 241

MOVE.L ProducerTextHandle,-(SP)
   _TEActivate
BRA Activate99

Activate3
   CMP.L DateWindowPtr,A0
   BNE Activate4
   MOVE.L DateTimeHandle,-(SP)
   _TEActivate
   BRA Activate99

Activate4
   CMP.L RatingWindowPtr,A0
   BNE Activate5
   MOVE.L RatingTextHandle,-(SP)
   _TEActivate
   BRA Activate99

Activate5
   CMP.L NumberWindowPtr,A0
   BNE Activate6
   MOVE.L NumberTextHandle,-(SP)
   _TEActivate
   BRA Activate99

Activate6
   CMP.L AnnotationWindowPtr,A0
   BNE Activate98 ;not one of our text windows
   MOVE.L AnnotationTextHandle,-(SP)
   _TEActivate

Activate99
   MOVE.L Message,-(SP) ;make this the current grafport
   _SetPort

Activate98
   RTS

DeActivate
   CMP.L NameWindowPtr,A0
   BNE DeActivate1
   MOVE.L NameTextHandle,-(SP)
   _TeDeActivate
   RTS

DeActivate1
   CMP.L ProducerWindowPtr,A0
   BNE DeActivate2
   MOVE.L ProducerTextHandle,-(SP)
   _TeDeActivate
   RTS

DeActivate2
   CMP.L DateWindowPtr,A0
   BNE DeActivate3
   MOVE.L DateTimeHandle,-(SP)
   _TeDeActivate
   RTS

(continued)
Listing 9.1 (continued)

DeActivate3
    CMP.L RatingWindowPtr, A0
    BNE DeActivate4
    MOVE.L RatingTextHandle, -(SP)
    __TeDeActivate
    RTS

DeActivate4
    CMP.L NumberWindowPtr, A0
    BNE DeActivate5
    MOVE.L NumberTextHandle, -(SP)
    __TeDeActivate
    RTS

DeActivate5
    CMP.L AnnotationWindowPtr, A0
    BNE DeActivate6 ; not a text window
    MOVE.L AnnotationTextHandle, -(SP)
    __TeDeActivate
    RTS

DeActivate6
    RTS

Assuming that activeFlag has been set (the window should be activated), the application does not execute the branch at (d). Instead it continues processing with statement (e). This is where the code begins the somewhat tedious process of identifying exactly which text window posted the activate event. There is only one reliable way to do so. The pointer retrieved from the message field of the event record must be compared to the pointer for each text window used in the application. A match indicates that the proper window has been found. Why is this necessary? Because TEActivate requires the text edit handle associated with the window being activated; there is no way to activate the appropriate text edit window without knowing specifically which text edit handle should be placed on the stack. Therefore, the application, at (e), compares the pointer from message (stored in A0) with a pointer to one of the text edit windows. In this case, the program looks first at the window for the tape name, but the order in which the windows are processed is nonetheless arbitrary.

If a match between the two pointers is not found (f), the program must branch to check the next window (i). On the other hand, if the two pointers are the same, then the application can proceed to activate the window. Statement (g) places the appropriate text edit handle on the stack. The window is then activated with a call to TEActivate (h).

One final step remains in the activation process: the newly activated window must be made the current grafport. Otherwise, no drawing can be performed in the window. Therefore, a call to SetPort is performed at (i).

The entire procedure is repeated for each text edit window.
Deactivating the text edit windows, beginning at (k), is more or less the same as activating. The window involved must be identified by comparing its pointer to the pointer from message so that its text edit handle can be placed on the stack. The handle is placed on the stack followed by a call to TeDeActivate. Once the window is deactivated, the application can return to the main program, since no call to SetPort is required.

**Getting a Blinking Cursor**

The blinking cursor in a text edit window is controlled by TEIdle. Like the SystemTask routine that updates desk accessories, TEIdle must be called repeatedly. It should be a part of each event loop in an application. Like the activate and deactivate procedures, TEIdle requires only the handle to the edit record of the text edit window where the cursor should blink (i.e., this must be the handle of the currently active text edit window):

```
PROCEDURE TEIdle (hTE: TEHandle);
```

The fact that TEIdle requires the handle of the currently active text window presents a problem for applications where there is more than one text edit window, any of which might be active while the same event loop is controlling program action. To solve this problem, TEIdle can be called with a sort of "generic" text edit handle. The Video Tape Index, for example, has allocated additional space for a text edit handle called ActiveTextHandle. Whenever a text edit window is selected, its handle is moved into ActiveTextHandle, which is then passed to TEIdle. Therefore, all calls to TEIdle appear as:

```
MOVE.L ActiveTextHandle, - (SP)
__TEIdle
```

**Moving the Cursor (Setting the Selection Range)**

A selection range is what is highlighted in inverse video when you drag the mouse across a range of text or shift-click to indicate everything between the cursor and the click. A selection range can also be a single spot if it simply refers to the position of the blinking cursor.

Text edit records identify the selection range by counting the characters in the text, beginning from the left; the first position is numbered 0. The range can be set by an application's response to mouse actions or by the application itself.

If an application returns a mouse down event in an active text edit window, then the program can assume that the selection range needs to be moved. As you remember, a call to FrontWindow will return a pointer to the currently active
window. If this is the same as the pointer returned by `FindWindow`, then indeed the mouse down event occurred in the active window and the selection range should be adjusted.

`TEClick` takes care of positioning the selection range. It will move the straight-line cursor as well as highlight text and can handle extended selection ranges indicated by shift-click actions. `TEClick` needs to know where the mouse down event occurred, whether to process shift-click actions, and the text edit handle:

```plaintext
PROCEDURE TEClick (pt: Point; extend: BOOLEAN; hTE: TEHandle);

Point is from the event record. If `extend` is true, a shift-click will be processed; if `extend` is false, the cursor will simply be repositioned, regardless of whether the shift key was held down. Therefore, an application must check the `Modify` field of the event record prior to calling `TEClick` to determine what value to give the `extend` boolean. The final parameter is simply the handle to the text edit record whose window posted the mouse down event.

There is one catch here — the `Point` field from the event record returns the position of the mouse down event in global coordinates; `TEClick` requires that they be expressed in the local coordinates of the current grafport (i.e., those of the currently active text edit window). Therefore, before an application can call `TEClick`, the mouse coordinates must be converted to the local coordinate system.

As discussed previously, the QuickDraw routine `GlobalToLocal` will take care of the conversion:

```plaintext
PROCEDURE GlobalToLocal (VAR pt: Point);

Point is passed into the routine as global coordinates and is returned in local coordinates. (As you might expect, there is also a `LocalToGlobal` routine.)

The code for handling selection range movement using the mouse is therefore:

```plaintext
PEA Point ;push address so changed values can return
__GlobalToLocal

MOVE.L Point, -(SP) ;coordinates are now local
BTST #shiftKey,Modify ;was shift key pressed?
SNE D0 ;set true if shift key was held down
MOVE.B D0, -(SP) ;moving byte puts boolean in high order byte — system automatically pushes extra byte to keep stack pointer on even word boundary

MOVE.L ActiveTextHandle, -(SP) __TEClick
```
Note that the quantity `shiftKey` refers to the bit in the `Modify` field that is set when the shift key is held down during a mouse down or key down event.

The second way to control the selection range is to explicitly set it within the application itself. The routine `TESetSelect` will do just that:

**PROCEDURE TESetSelect (selStart, selEnd: LONGINTEGER; hTE: TEHandle);**

`selStart` refers to the position to which the start of the selection range should be set. To set it to the first position, it should take a value of 0 since character positions are, as mentioned above, numbered from the left beginning with 0. `selEnd` refers to the character position which should be the right-most edge of the selection range. The routine also requires the handle to the text edit record.

Why might an application need to set its own selection range? Consider the situation where an application must clear all the text from a text edit record without deleting the record itself since it will be used again. It makes sense to select all the text and then "cut" it out. (Implementing "cut" is discussed below.) The start of the selection range would therefore be set to 0 and its end to the length of the text or the last possible character position allowed by the view and destination rectangles. If the end of the selection range given in the procedure call is beyond the last character actually present, it will be modified to correspond to the position of that last character. To select all the text in the Producer text edit record (a 20 character field), the Video Tape Index uses:

```
MOVE.L #0, -(SP) ;starting position
MOVE.L #20, -(SP) ;ending position
MOVE.L ProducerTextHandle, -(SP)
__TESetSelect
```

If you look at the environment in which the Video Tape Index does this selection range assignment (see Listing 9.3, discussed later in this chapter), you'll notice that the sequence of events includes first selecting the window and then setting the grafport. TextEdit routines are very sensitive about grafports. To be safe, whenever preparing to call a TextEdit routine, be certain to set the correct grafport prior to making the call to TextEdit.

**Inserting Characters into Text Edit Records**

Just as selection ranges can be manipulated by the mouse or explicitly from within an application, characters can also be accepted from the keyboard or inserted into a text edit record by an application itself.

If an application detects a key down event without an accompanying press of the cloverleaf key, then the key press represents a character to be inserted into a
text edit record and displayed on the screen. The routine which actually inserts the character is **TEKey**:

**PROCEDURE TEKey (key: CHAR; hTE: TEHandle);**

The character that was pressed is available in the low-order word of the event record's `Message` field (i.e., `Message + 2`, just as used when identifying the key pressed in conjunction with the cloverleaf key). The second parameter is a handle to the currently active text edit record. **TEKey** inserts the key pressed into the text edit record and displays the character on the screen. It also removes characters that are deleted by the backspace key. To call it, use something like:

```
MOVE Message + 2, -(SP) ; character that was pressed
MOVE.L ActiveTextHandle, -(SP)
__TEKey
```

An application can do its own text insertion and display with **TEInsert**:

**PROCEDURE TEInsert (text: Ptr; length: LONGINT; hTE: TEHandle);**

The text specified by `text` (a pointer to where the text to be inserted is stored) will be inserted into the text edit record and drawn on the screen to the left of the current selection range. The `length` parameter contains the number of characters to be inserted.

The Video Tape Index uses this technique to display a record which has been retrieved using any of its three search strategies: printing all records sequentially; doing a binary search on a tape name; doing a sequential search on producer, date, rating, or tape number. The subroutine that performs the display, **DisplayOneRecord**, appears in Listing 9.2.

The subroutine first removes any previous text stored in the text edit windows (a). This procedure (Listing 9.3) is discussed in detail later in this chapter. The next three statements, beginning with (b), take the number for the record being displayed (stored in `RecordCounter`) and compute a byte offset into `TapeArray`. This offset locates the start of the record whose data will be inserted into the text edit records.

Each text edit record must be handled separately. Since **TEInsert** displays characters as well as inserting them into the text edit record, the first step is to make the appropriate text edit window the current grafport with a call to **SetPort**. This occurs at (c) for the tape name window only.

The subroutine then prepares for the call to **TEInsert**. The first parameter is the starting address of the text that is to be inserted into the text edit record. That address is the sum of three things: the starting address of `TapeArray` (d), the byte offset into `TapeArray` that locates the start of the record (e), and an offset into the record for the field whose contents are being inserted.
Listing 9.2 Inserting Text Directly into Text Edit Records

```
DisplayOneRecord
(a) JSR DisplayWindows ; clears out text edit records (Listing 9.3)
(b) LEA RecordCounter,A0
    MOVE (A0),D5
    MULU #64,D5
    MOVE.L NameWindowPtr,-(SP)
    _SetPort
    LEA TapeArray(A5),A0
    ADD D5,A0
    ADD.L #0Producer,A0
    MOVE.L A0,-(SP)
    MOVE.L #20,-(SP)
    MOVE.L ProducerTextHandle,-(SP)
    _TElnsert
    MOVE.L DateWindowPtr,-(SP)
    _SetPort
    LEA TapeArray(A5),A0
    ADD D5,A0
    ADD.L #0ReleaseDate,A0
    MOVE.L A0,-(SP)
    MOVE.L #4,-(SP)
    MOVE.L DateTextHandle,-(SP)
    _TElnsert
    MOVE.L RatingWindowPtr,-(SP)
    _SetPort
    LEA TapeArray(A5),A0
    ADD D5,A0
    ADD.L #0Rating,A0
    MOVE.L A0,-(SP)
    MOVE.L #4,-(SP)
    MOVE.L RatingTextHandle,-(SP)
    _TElnsert
    MOVE.L NumberWindowPtr,-(SP)
    _SetPort
    LEA TapeArray(A5),A0
    ADD D5,A0
    ADD.L #0TapeNumber,A0
    MOVE.L A0,-(SP)
    MOVE.L #4,-(SP)
    MOVE.L NumberTextHandle,-(SP)
    _TElnsert
    RTS
```
The name of the tape has an offset of 0, since it is the first field in the record and therefore needn't be considered when dealing with the tape name. Note, however, that for the other fields, the offset is included in the address computation. For example, look at statement (j), which adds the offset for the producer's name to the address in A0.

Once the starting address of the source text is computed, it is pushed onto the stack (f). That address is followed by the number of bytes which should be inserted (g) and the handle to the appropriate text edit record (h). The process is completed by callingTECT (i).

This sequence of events is repeated for each text edit window except the annotation window. Since annotations are kept on disk in a direct access file and only brought into memory as needed, annotation display is handled separately.

**Editing Text: Cut, Copy, Paste, and Delete (Clear)**

Those text editing functions for which the Macintosh is famous are surprisingly easy to implement. Cut, copy, paste, and delete (called "clear" in the Edit menu) each base their actions on the current selection range of a given text edit record. As discussed above, the placement of that selection range is controlled by eitherTECT or TESetSelect.

If an application detects the "cut" command (through either a cloverleaf-X key press or a mouse down event in the Edit menu), it should callTECT:

**PROCEDURE TECut (hTE: TEHandle);**

The text in the current selection range will be deleted from the text edit record and copied to the Clipboard. The text will be removed from the screen and the rest of the text adjusted to compensate for the characters that were deleted.

If an application needs to remove text without copying it to the Clipboard, it can useTEDelete instead ofTECT:

**PROCEDURE TEDelete (hTE: TEHandle);**

On the other hand, to get text onto the Clipboard without deleting it from the text edit record, useTECopy:

**PROCEDURE TECopy (hTE: TEHandle);**

Pasting from the Clipboard into a text edit record is similarly straightforward:

**PROCEDURE TEPaste (hTE: TEHandle);**

TEPaste takes whatever is on the Clipboard and inserts it into the text edit record just before the current insertion point. The screen display is adjusted to compensate for the new text. Pasting does not, by the way, disturb the contents of the
Clipboard. Only a Cut or Copy operation will do that. An application, therefore, can repeatedly paste the same text into a text edit record until such time as another Cut or Copy is executed.

The Video Tape Index uses the Cut function to clear out its text edit records. Whenever it becomes necessary to remove all characters from both the text entry windows and the text edit records, the program executes the following sequence of steps:

1. Select a window (SelectWindow)
2. Make it the current grafport (SetPort)
3. Select the selection range to the maximum number of characters that will be stored in this text edit record (TESetSelect)
4. Cut the text (TECut)

The procedure outlined above is used in the Video Tape Index's subroutine DisplayWindows (Listing 9.3). DisplayWindows selects each text edit window in turn, which brings it in front of the main window. It also cuts out any text that might be stored in the text edit records, so that the windows are empty when they appear. As with the other subroutines that deal with the text edit records, DisplayWindows must handle each text edit window separately, repeating the same sequence for every window. DisplayWindows returns with the tape name window active.

The first step is to select the window (a) and to then make it the current grafport (b). At that point, any existing text in the text edit record must be removed. In order to make TECut operate on all characters that are present, the subroutine first sets the selection range to encompass the maximum numbers of characters that can appear in the specific field. It begins the selection range at the first character position (c) and ends it at the last possible character position (d). Note that this will not cause any problems if there are less than the maximum number of characters in the text edit record, since TESetSelect will automatically adjust the ending position to the last character actually present. After placing the appropriate text edit handle on the stack (e), a call is made to TESetSelect (f). The contents of the text edit record can then be removed with TECut (g).

The steps illustrated by statements (a) through (g) are repeated for each of the text edit windows. If you are looking at Listing 9.3, however, you will see that a number of other things happen in DisplayWindows. These are the direct result of the calls made to SelectWindow.

SelectWindow not only brings a window to the front, but it also highlights that window. For the name, producer, rating, date, and number windows, highlighting is unimportant since their windows are simply outlined rectangles. But the annotation window is a standard document window with a title bar. SelectWindow will highlight it and leave it highlighted. Since the annotation window will not be the active window when DisplayWindows returns, it should not be highlighted. Therefore, the three statements beginning at (h) issue a call to HiliteWindow to remove the highlighting.
Listing 9.3  Setting a Text Edit Selection Range from within an Application

DisplayWindows
MOVE.L AnnotationWindowPtr,-(SP)
(a)  _SelectWindow
MOVE.L AnnotationWindowPtr,-(SP)
(b)  _SetPort
(c)  MOVE.L #Ø,-(SP)
(d)  MOVE.L #255,-(SP)
(e)  MOVE.L AnnotationTextHandle,-(SP)
(f)  _TESetSelect ;select all the text in the window
MOVE.L AnnotationTextHandle,-(SP)
(g)  _TECut ;cut out text from previous use

MOVE.L AnnotationWindowPtr,-(SP)
(h)  _HiliteWindow ;get rid of highlighting in this window
SF - (SP)

MOVE.L NumberWindowPtr,-(SP)
_selectWindow
MOVE.L NumberWindowPtr,-(SP)
_setPort
MOVE.L #Ø,-(SP)
MOVE.L 2Ø,-(SP)
MOVE.L NumberTextHandle,-(SP)
_TESetSelect
MOVE.L NumberTextHandle,-(SP)
_TECut

MOVE.L RatingWindowPtr,-(SP)
_selectWindow
MOVE.L RatingWindowPtr,-(SP)
_setPort
MOVE.L #Ø,-(SP)
MOVE.L #4,-(SP)
MOVE.L RatingTextHandle,-(SP)
_TESetSelect
MOVE.L RatingTextHandle,-(SP)
_TECut

MOVE.L DateWindowPtr,-(SP)
_selectWindow
MOVE.L DateWindowPtr,-(SP)
_setPort
MOVE.L #Ø,-(SP)
MOVE.L #5,-(SP)
MOVE.L DateTextHandle,-(SP)
_TESetSelect
MOVE.L DateTextHandle,-(SP)
_TECut

MOVE.L ProducerWindowPtr,-(SP)
_selectWindow
MOVE.L ProducerWindowPtr,-(SP)
_setPort
MOVE.L #Ø,-(SP)
MOVE.L #22,-(SP)
MOVE.L ProducerTextHandle,-(SP)
(continued)
SCREEN AND KEYBOARD I/O: USING TEXTEDIT, ALERT AND DIALOG BOXES

1/0: USING TEXTEDIT, ALERT AND DIALOG BOXES

Move to Window

MOVE.L NameTextHandle,-(SP)
_EQut

(i)

MOVE.L $00000100,D0 ;mask to remove activate events

(j)

.Move.L NameWindowPtr,-(SP)
_SELECTWindow
Move.L NameWindowPtr,-(SP)
_SetPort
Move.L #0,-(SP)
Move.L #32,-(SP)
Move.L NameTextHandle,-(SP)
_EQut
Move.L NameTextHandle,-(SP)
_EQut

(k)

LEA ActiveTextHandle,A0
Move.L NameTextHandle,(A0) ;for TEIdle

RTS

SelectWindow also generates two activate events each time it is called: one for the window being activated and one for the window being deactivated. The activate events from DisplayWindows are in some sense spurious; they do not correspond to any real need to activate or deactivate text edit windows. Their presence will confuse an event loop. Therefore, before dealing with the tape name window, which will be active when the subroutine ends, those extra activate events should be removed. A special mask is created to identify only activate events (i) and then used for a call to FlushEvents (j). This will remove those activate events before they can be processed by an event loop.

Since the tape name window will be active when DisplayWindows returns, TEIdle should have the handle to the name text edit record. Therefore, the name text edit handle is loaded into the generic text handle just before the subroutine finishes (j).

There is, by the way, an alternative way to delete the text in a text edit record – simply dispose of the entire record. The routine TEDispose (it requires only the handle to the text edit record as a parameter) removes the text edit record from memory. It would certainly be possible to dispose and then reallocate the text edit records each time the Video Tape Index finished with a given record. Doing so, however, requires more code than emptying the text edit record by cutting out its entire contents.

Displaying Static Text

In some cases it may be necessary to display text that won't be changed. For example, the Video Tape Index prints the name of the field to the left of each text edit window (see Figure 9.1). These prompts are essential; otherwise the user will
have no idea what information should be entered in each text window. Nonetheless, there is no need to change those prompts once they are printed.

The prompts for the TapeArray fields are printed on the main window. They could have been printed with DrawString, the ToolBox routine used in the Sample program to display text. To use DrawString an application must first move the cursor to the coordinates where printing should begin. DrawString then prints the characters, moving the cursor from left to right. This can be somewhat awkward, especially when the text needs to be justified (e.g., note that the Video Tape Index prompts are printed in a proportional type font which is lined up along the right hand side).

![Figure 9.1 Displaying Static Text](image)

TextEdit provides an alternative for printing static text with the TextBox routine. TextBox prints a string of text inside a rectangle expressed in the local coordinates of the current grafport. The rectangle has no visible borders, nor is any text edit record created. The routine also allows an application to specify text justification within the rectangle (left, right, or centered). The format of the call is:

```pascal
PROCEDURE TextBox (text: Ptr; length: LONGINTEGER; box: Rect; just: INTEGER)
```

As with anything else that requires placing coordinates on the Macintosh screen, using TextBox requires a bit of planning. For example, since the Video Tape Index will print its prompts on the main window, the coordinates of the
rectangles in which the prompts will be printed must be expressed in terms of that main window (upper left-hand corner becomes 0,0 and lower right-hand corner becomes 240,490). To keep things simple, the windows for each prompt are the same size (11 pixels high and 191 pixels wide). The height is dictated by the size of the text; if a window is to display 12-point text, it must be a minimum of 10 pixels high. The width obviously depends on the maximum number of characters that will be printed. Establishing the exact placement of each rectangle nonetheless requires a bit of trial and error.

The parameter just is an integer that indicates how the text should be justified within its rectangle. A value of 0 will left-justify the text, 1 will center it, and -1 will right-justify. length is the number of characters to print, and text is a pointer to the text to be printed.

Consider as an example the code that displays the prompt for the Date text entry window:

```
PEA StringConstant ;pointer to string
MOVE.L #17, -(SP) ;number of characters to print
PEA DatePromptBox ;rectangle where text should go
MOVE # -1, -(SP) ;right justify the text
__TextBox
```

Two things must have occurred before the above code will function properly. First, the main window must be the current grafport (through a call to SetPort). Secondly, the rectangle DatePromptBox must have been defined. For example:

```
DatePromptBox DC 62,10,82,200
```

It is also important to be sure that the string passed to TextBox has the data type Str255 (i.e., its first byte is a length byte). The easiest way to do so is to allocate space for the string with DC. For example:

```
StringConstant DC 'Date of Release'
```

The subroutine that displays the Video Tape Index's prompts, DisplayPrompts, appears in Listing 9.4. DisplayPrompts first establishes the font that should be used when the prompts are drawn (a). Setting the text font is discussed a bit further on in this chapter. Then, the TextBox sequence is repeated for each of the text edit windows that occupy plain document boxes (i.e., all but the annotation window). The text of the prompts have all been established as constants with DC, ensuring that they will be assembled with a length byte. The first step (b), is to push a pointer to the title string onto the stack. That pointer is followed by the number of characters in the string (c), a pointer to the rectangle in which the text should be printed (d), and the text justification (d). The text is actually printed with the call to TextBox (e).
Listing 9.4 Using TextBox to Display Static Text

DisplayPrompts
    MOVE #sysFont, -(SP)
    _TextFont

(b)     PEA NameTitle
    MOVE.L #11, -(SP) ; number of characters to print
(c)     PEA NamePromptBox ; rectangle where text should be printed
(d)     MOVE #1, -(SP) ; to right justify text
    _TextBox

    PEA ProducerTitle
    MOVE.L #22, -(SP)
    PEA ProducerPromptBox
    MOVE #1, -(SP)
    _TextBox

    PEA DateTitle
    MOVE.L #17, -(SP)
    PEA DatePromptBox
    MOVE #1, -(SP)
    _TextBox

    PEA RatingTitle
    MOVE.L #8, -(SP)
    PEA RatingPromptBox
    MOVE #1, -(SP)
    _TextBox

    PEA NumberTitle
    MOVE.L #13, -(SP)
    PEA NumberPromptBox
    MOVE #1, -(SP)
    _TextBox

RTS

NamePromptBox  DC  12,10,32,200
NameTitle      DC  'Tape Name:'
ProducerPromptBox  DC  37,10,57,200
ProducerTitle  DC  'Producer/Distributor:'
DatePromptBox   DC  62,10,82,200
DateTitle      DC  'Date of Release:'
RatingPromptBox DC  87,10,107,200
RatingTitle    DC  'Rating:'
NumberPromptBox DC  112,10,132,200
NumberTitle    DC  'Tape Number:'
Updating Text Edit Windows

TextEdit has its own routine for updating text edit windows. Whenever an update event is detected in a text edit window, an application should execute the following sequence of steps:

1. Call `BeginUpdate`
2. Call `EraseRect` (this ensures that when the window is deactivated, the cursor will disappear)
3. Call the special TextEdit routine `TEUpdate` (discussed below)
4. Call `EndUpdate`

`TEUpdate` redraws the text specified by a rectangle parameter:

```
PROCEDURE TEUpdate (rUpdate: Rect; hTE: TEHandle);
```

Generally, the text edit window's view rectangle is used for the `rUpdate` parameter. It is also important to remember that the text edit window referred to by `TEHandle` must be the current grafport in order for `TEUpdate` to work properly.

The Event Manager will return update events only for an active window. If an application has windows which are visible but not active, any changes in their contents will not be reported. For example, consider the Video Tape Index's text entry screen (e.g., Figure 9.1). Only one text entry window is active at any given time, yet it is possible to use a desk accessory that will overlay, and therefore erase portions of, windows that are not active. Therefore, it may not always suffice to update just the window reporting the update event.

The Video Tape Index handles updating text windows with the subroutine `UpdateTextWindows` (Listing 9.5). Whenever an update event is detected, the program erases and redraws the contents of all windows.

The main window, because it is not a text edit window, is handled a bit differently from the text edit windows. As with all updates, the process begins with a call to `BeginUpdate` (a). To ensure that all routines which affect the screen will function properly, it is then made the current grafport using `SetPort` (b). `UpdateTextWindows` then erases the main window's contents (`EraseRect` at (c)). The window's contents are redrawn by the subroutine `DisplayPrompts` from Listing 9.3 (d). As mentioned earlier, it is usually easier to erase and completely redraw a window's contents than it is to merely redraw the specific portion that has been disturbed by some other program action. The update is completed by calling `EndUpdate` (e).
Updates for the text edit windows begin in the same manner as updates to the main window — calling BeginUpdate (f), setting the grafport (g), and erasing the window (h). Redrawing the window’s contents, however, is where the difference lies. TEUpdate will take care of redrawing the text. That routine requires that the text window’s view rectangle (i) and its text handle (j) be placed on the stack before making the call (k). As usual, the update ends with EndUpdate (l).

Listing 9.5 Updating Multiple Windows

```
UpdateTextWindows
  MOVE.L MainWindowPtr,-(SP) ;Begin
  _BeginUpdate
  MOVE.L MainWindowPtr,-(SP) ;SetPort
  _SetPort
  PEA MainWindowRect ;EraseRect
  _EraseRect
  JSR Display Prompts ;re-draw window’s contents
  MOVE.L MainWindowPtr,-(SP) ;End
  _EndUpdate
  
  MOVE.L NameWindowPtr,-(SP) ;Begin
  _BeginUpdate
  MOVE.L NameWindowPtr,-(SP) ;SetPort
  _SetPort
  PEA NameViewRect ;EraseRect
  _EraseRect
  PEA NameViewRect ;TEUpdate
  _TEUpdate
  MOVE.L NameWindowPtr,-(SP) ;End
  _EndUpdate
  
  MOVE.L ProducerWindowPtr,-(SP) ;Begin
  _BeginUpdate
  MOVE.L ProducerWindowPtr,-(SP) ;SetPort
  _SetPort
  PEA ProducerViewRect ;EraseRect
  _EraseRect
  PEA ProducerViewRect ;TEUpdate
  _TEUpdate
  MOVE.L ProducerWindowPtr,-(SP) ;End
  _EndUpdate
  
  MOVE.L DateWindowPtr,-(SP) ;Begin
  _BeginUpdate
  MOVE.L DateWindowPtr,-(SP) ;SetPort
  _SetPort
  PEA DateViewRect ;EraseRect
  _EraseRect
  PEA DateViewRect ;TEUpdate
  _TEUpdate
  MOVE.L DateWindowPtr,-(SP) ;End
  _EndUpdate
  
(continued)
```
MOVE.L RatingWindowPtr,-(SP)
MOVE.L SetPort 
PEA Erase Rect 
PEA MOV.L TEUpdate 
MOVE.L _EndUpdate 

MOVE.L NumberWindowPtr,-(SP)
MOVE.L SetPort 
PEA EraseRect 
PEA MOV.L TEUpdate 
MOVE.L _EndUpdate 

MOVE.L AnnotationWindowPtr,-(SP)
MOVE.L SetPort 
PEA EraseRect 
PEA MOV.L TEUpdate 
MOVE.L _EndUpdate 

RTS

Changing Fonts and Font Characteristics

One of the things that always excites new users about the Macintosh is its ability to manipulate multiple fonts with varying characteristics within a single text window. The three routines that manage those features are part of QuickDraw.

**TextFont** takes care of setting the font itself:

```pascal
PROCEDURE TextFont (font: INTEGER);
```

Each font is identified by a font number. Equates for the standard release fonts are included in the QuickDraw equates file. The system font (Chicago), for example, has an ID of 0, while NewYork is 2 and London is 6. The address assigned to the standard release fonts appear in Table 9.1.
Table 9.1 Symbolic Addresses Assigned to Standard Release Fonts

<table>
<thead>
<tr>
<th>Symbolic Address/Font Name</th>
<th>Font Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysFont (Chicago)</td>
<td>Ø</td>
</tr>
<tr>
<td>applFont (Geneva)</td>
<td>1</td>
</tr>
<tr>
<td>newYork</td>
<td>2</td>
</tr>
<tr>
<td>geneva</td>
<td>3</td>
</tr>
<tr>
<td>monaco</td>
<td>4</td>
</tr>
<tr>
<td>venice</td>
<td>5</td>
</tr>
<tr>
<td>london</td>
<td>6</td>
</tr>
<tr>
<td>athens</td>
<td>7</td>
</tr>
<tr>
<td>sanFran</td>
<td>8</td>
</tr>
<tr>
<td>toronto</td>
<td>9</td>
</tr>
<tr>
<td>cairo</td>
<td>10</td>
</tr>
<tr>
<td>losangel</td>
<td>11</td>
</tr>
</tbody>
</table>

To change the font, push the font ID number onto the stack and call the routine:

```
MOVE  #venice, -(SP)        ;pushes a 5
  _TextFont
```

It is important to remember that TextFont only affects the current grafport and must therefore be repeated whenever the grafport changes to another window.

The style of a font (boldface, italic, underlined, outlined, shadowed, etc.) is controlled by TextFace:

```
PROCEDURE TextFace (face: Style);
```

The actual style of the font is determine by the style word that is supplied as a parameter to the routine. Bits in the style word represent one type of text face (see Figure 9.2). For example, if bit 0 is set, text will be displayed in boldface. If bit 2 is set, the text will be underlined. If both bits 0 and 2 are set, the text will be both boldface and underlined. To create bold and underlined text, use:

```
MOVE  #5, -(SP)            ;the 5 = 0000 0000 0000 0101
  _TextFace
```

To return to normal text, use a style word of 0.

Like TextFont, TextFace also affects only the current grafport.

TextSize manipulates the size of the text in the current grafport:

```
PROCEDURE TextSize (size: INTEGER);
```

The size of the text is expressed in points, just seen in standard Style menus. Though an application can select virtually any size for any font, the text will look
best if it is expressed in a size that exists in the system. The following instructions will establish a text size of 14 points:

```
MOVE       #14, – (SP)
  —TextSize
```

Text justification is handled by the routine `TESetJust`:

```
PROCEDURE TESetJust (just: INTEGER; h: TEHandle);
```

`just` is one of the three numbers used to specify justification for `TextBox`: 0 to left-justify, 1 for centered text, and -1 for right-justification. `h` is a handle to a text edit record containing the text to be justified. For example, to center text you might code:

```
MOVE       #0, – (SP)
MOVE.L     SomeTextHandle(A5), – (SP)
  —TESetJust
```

`TESetJust` does not affect the text as it is displayed on the screen, only as it is stored in the text edit record. Therefore, to change the justification of the text on the screen, execute a complete update sequence that will erase the text edit window and redraw its contents with the new justification immediately after calling `TESetJust`.

![Style Word Used by TextFont](Figure 9.2 The Style Word Used by TextFont)

To select any font style, set the appropriate bit in the style word. The styles are additive. For example, to get outlined boldface text set bits Ø and 2.
Scrolling Text

Applications which permit the entry of large text documents will need to scroll text within text entry windows. Scrolling activities are implemented whenever the user clicks the mouse button somewhere in a scroll bar, or when the text being entered goes below the bottom edge of the view rectangle.

How far the text should be scrolled depends on what initiated the scrolling action. A single click in an up or down arrow will scroll the text one line. A click in a right or left arrow will scroll the text a character or two. A click in a page up or page down region will move text one "page" (generally one window full). On the other hand, if the user drags the thumb of a scroll bar, the amount to scroll will be proportional to the movement of the thumb.

Scrolling is implemented by a single TextEdit routine:

PROCEDURE TEScroll (dh,dv: INTEGER; hTE: TEHandle);

dh and dv are expressed in pixels. They specify how far the text should be scrolled. If both values are positive, dh refers to the number of pixels to scroll to the right and dv refers to the number of pixels to scroll down. If the values are negative, dh indicates the number of pixels to scroll left and dv the number of pixels to scroll up.

The height, in pixels, of a single line of text is contained in the text edit record in the field lineHeight. This parameter always reflects the current spacing (e.g., single or double spaced). Therefore, once an application determines the number of lines to scroll up or down, the number of pixels can be obtained by multiplying the number of lines to scroll by the number of pixels per line. For example, the following code will scroll text three lines down:

```
MOVE  #3,NumLines(A5) ;# of lines to scroll
MOVE.L  TextEditHandle,A0 ;handle to TE record
MOVE.L  (A0),A0 ;get pointer
MOVE  lineHeight(A0),D0 ;retrieve height of line
MULU  NumLines(A5),D0 ;total number of pixels
MOVE  #0, – (SP) ;don't move to the right
MOVE  D0, – (SP) ;pixels down
MOVE.L  TextEditHandle, – (SP) ;handle to TE record
__TEScroll
```

;application must now update the text edit window

In terms of figuring out which way to scroll, remember that when a user clicks the up arrow of a vertical scroll bar, the text should move down. By the same token, a click in the down arrow will scroll the text up. The same is true of thumb movement — if the thumb moves up, the text should move down; if the thumb moves down, the text should move up.
How far does text move when a thumb is dragged? Consider the situation where a scroll bar has a minimum of 0 and a maximum of 10. If the thumb is moved to the middle of the scroll bar, it will have a value of 5. The text should therefore be scrolled to the middle of document, regardless of the length of the document. If the thumb has a value of 2, the text should be scrolled 20% from the beginning of the document.

Left to right scrolling is usually more rigidly controlled than up and down scrolling. Most text processing applications assign a fixed maximum width to a document. For example, Microsoft Word limits the user to an 8 1/2-inch-wide page, even though margins can be set at will. ThinkTank 512 also limits the user to an 8-1/2-inch line. Therefore, the amount of scrolling that a single click in a left or right arrow will produce does not depend on the size of the font in use, but is a fixed interval based on the maximum width of the document. Dragging the thumb of the horizontal scroll bar is also proportional to the maximum fixed width of the document.

---

**Controlling Program Actions with Alert and Dialog Boxes**

Dialog boxes appear whenever a program needs information from the user in order to proceed. Alert boxes generally appear to warn the user that an error has occurred or that the potential to commit some error exists.

As discussed in Chapter 1, there are two types of dialog boxes — modal and modeless. Modal dialog boxes restrict the user to working within the dialog box. For example, consider the dialog box that appears when you select the PRINT option from a standard File menu (Figure 9.3). Until you either click the OK button with the mouse or hit the ENTER key, the only actions possible are changing the print parameters displayed by the dialog box.

Modeless dialog boxes are more like regular document windows. Their presence on the screen does not prevent the user from performing other activities. The most common example of a modeless dialog box is the window that appears when FIND is selected from a Search menu (Figure 1.12). The user can work in the FIND box, deactivate it by clicking on another visible window, work in another active window, and later reactivate the dialog box without ever removing it from the screen.

Alert boxes are more like modal dialog boxes. They, too, freeze program action until the user responds to the alert. But while modal dialog boxes are used whenever the program needs information, alert boxes signal errors or warnings.

Dialog and alert boxes are handled by the Dialog Manager. To properly set up an application for calls to the Dialog Manager, include a call to `InitDialogs` in the initialization portion of the program. `InitDialogs` takes one parameter — a pointer
to whatever routine should be started whenever a system error occurs and the system must be restarted. The pointer should either restart the current application or be 0:

```
CLR.L – (SP) ; no restart procedure
_InitDialogs
```

The call to `InitDialogs` can be the last initialization in the sequence.

![Figure 9.3 Standard Job Dialog](image)

**Defining Dialog and Alert Boxes**

Dialog and alert boxes, like other windows, are defined in resource files. Though there are routines for defining them completely within an application, it is many times easier to use a resource file. Like other windows, the boundaries of dialog and alert boxes are rectangles expressed in global coordinates. In many cases, dialog boxes appear centered on the screen just below the menu bar; this is the position in which the standard Macintosh user interface guidelines expects them to appear.

The Video Tape Index uses modal dialog boxes to control the progress of a search. Since only one record can be displayed at any one time, there must be
some mechanism to “freeze” the program, leaving that record on the screen until the user is ready to proceed. Therefore, rather than overlaying the text windows, the dialog boxes appear in the lower right-hand corner of the screen, just above the annotation window (see Figure 9.4).

Note that this technique of using modal dialog boxes to freeze program execution until the user is ready to go on is very much like using a dummy input sequence in a Pascal program. The Pascal statements:

```pascal
write ('Hit <CR> to continue:');
readln (Dummy);
```

have the same effect as using a modal dialog box, since in either case the program will not resume execution until the user responds.

The “Find More?” dialog box freezes program action until the user clicks the mouse button with the cursor in either the OK or Cancel button.

**Figure 9.4 Using a Dialog Box to Freeze Program Action**

The Video Tape Index uses an alert box whenever a search has been chosen from a menu but no search criteria has been entered. This box also appears in the lower right-hand side of the screen above the annotation window (see Figure 9.5).
Dialog Boxes

The template for dialog boxes appears much like a regular window template. As an example, consider the resource definition for the Video Tape Index's None Found dialog box:

```
TYPE DLOG
,1

"None Found"
100 300 170 490
Visible NoGoAway
2
0
1
```

;indicates the the following definitions are for dialog boxes
;sequence number for this dialog box. Must be unique within the resource type (i.e., no other dialog box can have this number)
;place for any message you like
;coordinates of the box's boundary rectangle
;same as for regular window definitions
;number corresponding to type of window
;reference value (always use 0)
;reference number to item list where box's contents are defined (see below for details)

Rules that apply to other resource definitions hold for dialog boxes as well. For example, the sequence number must be preceded by a space and a comma. The number indicating the type of window should be selected from those available for regular window definitions.
Alert Boxes

Alert boxes have their own resource template:

```
TYPE ALRT ; indicates that an alert box follows
 ,4 ; unique sequence number within all alerts
100 300 170 490 ; boundary rectangle
4 ; reference number to item list where box's contents are defined
7765 ; "stages" word (in hex)
```

The only unusual thing about an alert box definition is the stages word. This number, expressed in hexadecimal, controls what will happen each time the alert is invoked. It means that if the user continues to make the same error, the consequences can vary. Alerts can be instructed to beep the Mac's speaker one or more times, cause the menu bar to flash, display or not display the box itself, and change which button within the box will be the default button (i.e., the button that is selected when the user presses ENTER or RETURN).

Each alert has four stages; if an alert is called more than four times, it will simply repeat whatever actions are specified by the fourth stage. Each stage is controlled by four bits within the stages word:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>stage one</td>
</tr>
<tr>
<td>4–7</td>
<td>stage two</td>
</tr>
<tr>
<td>8–11</td>
<td>stage three</td>
</tr>
<tr>
<td>12–15</td>
<td>stage four</td>
</tr>
</tbody>
</table>

Within the four bits allocated to each stage, the highest-order bit refers to the item number of the default button minus 1 (the item number is the position of the item [usually a button] within the item list; item lists are discussed below). By convention, the first item in the list is always the OK button. It appears in the stages word as a 0. If a CANCEL button is present, it will be the second item in the item list and therefore is indicated as a 1 in the status word. The next lower-order bit is set if the alert box is to be drawn and cleared if it should not be drawn. The two lowest-order bits refer to how many times the speaker should be beeped (0 to 3).

To create a stages word, first design it in binary and then translate it to hexadecimal. If we convert the Video Tape Index's stages word to binary, we can see exactly what actions it instructs the Mac to take when the alert is invoked:

```
$7765 = %0111 0111 0110 0101
```

In all four stages, the highest-order (left-most) bit is 0. That indicates that the default item will always be the OK button. (As you will see below in the discussion of item lists, the OK button is the first item in the item list.) The second bit from the left is always 1. Therefore, the box will be displayed at all four stages. The difference between the four stages is in the number of times the speaker will sound. At stage
one it will beep once, at stage two twice, and three times at stages three and four. Note that if the value of the sound bits is 0, the speaker will not sound at all but the menu bar will flash.

**Item Lists**

The items which appear in alert and dialog boxes are also best defined in a resource file. They are linked to the appropriate box by the item list number within the alert or dialog box definition. Therefore, each alert and dialog box must assign a unique number to its item list; the Mac can't tell the difference between lists that belong to dialog boxes and those that belong to alerts.

A number of special items can appear in dialog and alert boxes. The phrase that should be used to identify the item in a resource file appears in boldface:

1. Buttons [button] (the hot-dog shaped buttons)
2. Check boxes [checkbox]
3. Radio buttons [radiobutton] (the round buttons)
4. Static text [staticText] (text that is simply displayed on the screen; it cannot be edited)
5. Edit text [editText] (text that can be edited; available only in dialog boxes)

Note that both static text and edit text items are limited to 241 characters.

Buttons and check boxes are controls. You can manage them directly through routines in the Control Manager, but when they are part of alert and dialog boxes, the Dialog Manager will make the calls to the Control Manager for you.

The item list for dialog box will appear as follows. This one is for the Video Tape Index's *None Found* dialog box:

```
TYPE DILT                      ; indicates that item lists follow
   ,1                          ; same as item list reference number
   2                           ; number of items in the list
   button                      ; type of item
   40 110 60 170                ; boundary rectangle for the item, expresssed in local coordinates of the dialog box
OK                            ; content of the item
   staticText                  ; type of item
   10 41 30 149                ; coordinates to enclose the text
   None Found                  ; text to be printed
```

Note that the location of each item in the dialog or alert box is indicated by a boundary rectangle. That rectangle is expressed in the local coordinates of the specific dialog or alert box.
The item number to which we referred earlier simply corresponds to an item's physical position in the item list. The first item that appears (in this case, the OK button) is item #1; the second item that appears (the text "None Found") is item #2.

For dialog boxes, the default item (the one that is selected when the user presses ENTER or RETURN) is always the first item in the list. As mentioned above, the stages word determines whether the first or second item will be the default for alerts.

The complete resource file templates for the Video Tape Index's alerts and dialog boxes are reprinted in Listing 9.6. The important thing to notice about these definitions is how the item lists are connected to the appropriate alert or dialog box by matching the number of the item list with the item list parameter in the alert or dialog box template.

**Listing 9.6 Resource Templates for the Video Tape Index's Alerts and Dialog Boxes**

```
TYPE DLOG
  ,1 Dialog box for "None Found" condition
    100 300 170 490
    Visible NoGoAway
  2
  Ø
  1

  ,2 Dialog box for "One Found/Find More?" condition
    100 300 170 490
    Visible NoGoAway
  2
  Ø
  2

  ,3 Dialog box for "One Found" condition
    100 300 170 490
    Visible NoGoAway
  2
  Ø
  3

TYPE ALRT
  ,4
    100 300 170 490
    4
    7765

  ,5
    50 140 120 390
    5
    4444

  ,6
    50 140 120 390
    6
    5555
```

(continued)
Listing 9.6 (continued)

TYPE DITL

button
40 110 60 170
OK

staticText
10 41 30 149
None Found

button
40 110 60 170
OK

button
40 20 60 80
Cancel

staticText
10 41 30 149
Find More?

button
40 110 60 170
OK

staticText
10 5 30 185
Selection criteria?

button
40 180 60 240
OK

staticText
10 10 30 240
Turn on printer. Press "Enter".

(continued)
Data Structures for Alert and Dialog Boxes

Dialog and alert boxes require only two data structures: a block of storage to hold the dialog window record (one will do if an application will never have more than one dialog box or alert on the screen at any given time), and a place to put a pointer to the dialog or alert window (this is returned by the routine that creates the box):

```
DialogWindRec    DS    dWindLen
DialogWindPtr    DS.L   1
```

The parameter `dWindLen` refers to the number of words in a dialog window record and is defined in the ToolBox equates file.

Creating and Disposing of Dialog Boxes

Unlike other windows, dialog boxes are usually created only when they are needed. They also are not hidden or made invisible when an application no longer needs them; rather, they are completely removed from the system. Re-use of the same dialog box during the same program run requires re-creation of the dialog box. Though modal dialog boxes can be managed like other windows (using `HideWindow`, `ShowWindow`, `BringToFront`, etc.), they generally are not, since they are used infrequently and their presence occupies memory the Mac can use for other purposes. Modeless dialog boxes are handled like other windows until the user clicks the GoAway box to close them, at which point they are deleted.

The ToolBox routine `GetNewDialog` will create and display a dialog box:

```
FUNCTION GetNewDialog (dialogID: INTEGER;
                      dStorage: Ptr; behind: WindowPtr) : DialogPtr;
```
The first parameter, `dialogID`, refers to the sequence number given to the dialog box in the resource file. (Don't confuse this sequence number with the number of the dialog box's item list; though the two numbers are often the same for convenience, they need not be.) `dStorage` is a pointer to the area set aside to store the dialog window record.

`behind` has the same function as the `behind` parameter in the `GetNewWindow` routine; it determines the placement of the dialog box with respect to the other windows in the screen. A value of −1 will place the dialog box in front of all others.

The result of `GetNewDialog` is a pointer to the dialog window. It is essential to save this pointer if any Window Manager routines are going to be used on this dialog window.

To create its `None Found` dialog box, the Video Tape Index uses this code:

```
CLR.L    -(SP)       ;space for dialog window pointer
MOVE     #1, -(SP)   ;this is dialog box 1
PEA      DialogWindRec(A5) ;pointer to dialog window record storage
MOVE.L   #-1, -(SP)  ;put dialog box in front
___GetNewDialog
MOVE.L   (SP) +(,DialogWindPtr(A5))  ;recover the window pointer
MOVE.L   DialogWindPtr(A5), -(SP)    ;make dialog box the current grafport
___SetPort
```

The final step in this sequence is an important one. The dialog box must be made the current grafport so that activities within the box will be properly recorded.

Disposal of a dialog box is taken care of by `CloseDialog`:

```
PROCEDURE CloseDialog (theDialog: DialogPtr);
MOVE theDialog's window pointer onto the stack and call the routine:

MOVE.L   DialogWindPtr(A5), -(SP)    ;CloseDialog
```

This will not only remove the dialog box from the screen, but will dispose of all data structures associated with the box.

**Managing Modal Dialog Box Actions**

There is no need to return to an application's event loop to monitor events relating to modal dialog boxes. The ToolBox routine `ModalDialog` performs all
necessary event trapping. **ModalDialog** polls the event manager by calling **GetNextEvent**. It also makes repeated calls to **SystemTask** to make sure that desk accessories are properly updated. If a mouse down event occurs outside the dialog box, the speaker will beep.

The Pascal format for **ModalDialog** is:

```
PROCEDURE ModalDialog (filterProc: ProcPtr;
    VAR itemHit: INTEGER);
```

**filterProc** refers a pointer to a function that determines how **ModalDialog** should interpret events from the event queue. A value of 0 for the filter procedure pointer will cause **ModalDialog** to default to the standard filter procedure. The standard filter procedure returns the value 1 for **itemHit** whenever the user hits the ENTER or RETURN keys. Assuming that the dialog box's OK button is the first item in the item list, then the standard filter procedure will allow the user to select OK with the ENTER or RETURN keys. This usage is consistent with the standard Macintosh user interface.

Using **ModalDialog** to monitor for an OK requires only a simple loop:

```
Loop
    MOVE.L #0, -(SP)
    PEA WhatItem(A5)
    ModalDialog
    MOVE WhatItem(A5),DO
    CMP #okButton,DO
    BNE Loop
```

The constant **okButton** is defined in the Toolbox equates file.

If other actions are possible, then the loop must continue to check item numbers and take the appropriate action. This process is directly analogous to identifying which item was selected from a menu.

Note that events in modeless dialog boxes are handled like those in other windows. An application's event loop must monitor any modeless dialog boxes that are present along with system and application windows.

### Creating and Managing Alert Boxes

A single Toolbox routine handles creating alert boxes and monitoring events until either the ENTER or RETURN key is pressed or a mouse down event occurs in the box. That same routine also takes care of disposing of the box when it is no longer needed. Note that if an application needs something other than an OK or a CANCEL reaction to some condition, then an alert box is probably not the correct way to control the situation; use a dialog box instead.
The routine that takes care of alert boxes is simply called `Alert`:

```plaintext
FUNCTION Alert(alertID: INTEGER; filterProc: ProcPtr): INTEGER;
```

`alertID` is the sequence number of the alert box's definition in the resource file. `filterProc` is a pointer to a procedure that indicates how `Alert` should select events from the event queue. As with `ModalDialog`, using a 0 will select the standard filter procedure (pressing ENTER or RETURN selects the default button just as if the user clicked on it with the mouse).

`Alert` returns a result that corresponds the position in the item list of the item that was selected. If an alert box has only an OK button, then `Alert`'s result can be disregarded. Nonetheless, the result must be removed from the stack. If the box has box OK and CANCEL buttons, then the application must examine the result to determine which button was selected and what action to take.

The Video Tape Index uses one of its three alert boxes (Figure 9.5) to indicate that a search request was made before selection criteria was entered. Therefore, the box only contains some static text and an OK button (the box merely freezes program action until the user is ready to continue). A "no selection criteria" condition (indicated by a length of 0 in the text edit record for the field on which the chosen search is to be based) initiates the following actions:

```
CLR # (SP) ;space for alert item result
MOVE #4, # (SP) ;alert box sequence number
MOVE.L #0, # (SP) ;use standard filter procedure
__Alert
MOVE (SP) +, D0 ;retrieve result to keep stack pointer in good order
```

Note that the result of `Alert` is not checked in this case, since the only button present is the OK button.

---

**Questions and Problems**

1. Assume that a window has been defined with a boundary rectangle of 10, 10, 335, 500. Write a block of code that will define a text edit record that uses the entire window. Allocate any necessary constants and data structures, including needed rectangles. Be sure to retrieve the text edit handle from the stack.
2. A. What sequence of events generates an activate event for a text edit window?  
B. Under what circumstances should the window be deactivated?  
C. Under what circumstances should it be activated?  

3. Write an ordered list of the ToolBox and/or operating system calls needed to activate a text edit window. Indicate the information returned by each call. Assume that an event loop has already detected an activate event.

4. Write the assembly language code to implement the procedure outlined in problem 2. Remember to distinguish between the need to activate or deactivate a window. Use the event record field names as defined in Chapter 8. Allocate any other data structures your code will use.

5. A user has pressed the cloverleaf and X keys together (the keyboard equivalent of selecting "cut" from the Edit menu). Write an ordered list of ToolBox and/or operating system calls needed to process the cut operation. Assume that an event loop has already detected a key down event. Indicate the information returned by each call.

6. Write the assembly language code to implement the cut operation outlined in problem 5. Assume that the Edit menu is menu #2. Use the event record field names as defined in Chapter 8. Allocate any other data structures your code will use.

7. Describe the differences between the following ToolBox routines, each of which displays text:
   a. DrawChar
   b. DrawString
   c. TEKey
   d. TElInsert
   e. TextBox

8. Write resource file templates to define a dialog box that will appear across the top quarter of the screen. The box is approximately 4" wide and 3" high. The items in the box are:
   A. a static text item (the actual text is up to you)
   B. an edit text item to hold one line of text
   C. an OK button
   D. a Cancel button

Choose the boundary rectangle for the dialog box and decide on placement of the items within it.
9. Write a block of assembly language code to create and monitor user actions in the dialog box defined in problem 8. Close the dialog box when the user clicks the mouse button in the OK or Cancel buttons, or presses the Enter or Return key. Allocate any data structures the code will require.

10. Write resource file templates to define an alert that will appear centered on the screen. It should be approximately 2" high and 3" wide. The items in the box are a line of static text of your choosing and an OK button. Select an appropriate boundary rectangle for the alert and decide on placement of the two items. The Mac should beep once the first time the alert is invoked, twice the second time, and three times the third and fourth times. The box should always be displayed; the OK button is always the default button.

11. Write a block of assembly language code to create and monitor user actions in the alert defined in problem 10. Close the alert box when the user clicks the mouse button in the OK button, or presses the Enter or Return key. Allocate any data structure the code will require.
Chapter Objectives

1. To understand the difference between draft and spooled printing
2. To learn the sequence of Printing Manager routines that control the printing process
3. To learn to position and produce images on a printed page

Introduction

Like most other aspects of writing a Macintosh application in assembly language, printing requires a great deal of planning. An application must not only figure out where to place text and graphics on the page, but must also determine parameters such as the space between lines (determined by the size of the font). Nonetheless, the printing process is rather "cookbook"; the basic steps are the same for all applications.

The Macintosh supports two types of printing — draft printing and spool printing. In draft printing, a document is printed directly, line by line, as text is sent to the Printing Manager. It is a very fast way to print, but is generally only suitable for printing text, since graphics requires the ability to move the cursor freely about the page. (This kind of movement is often referred to as direct cursor addressing.) Spool printing creates a disk file that contains an image of an entire document.
Since the print file is a direct access file, graphics images that require random cursor movement can be stored. Once a spooled print file is complete, it can then be printed line by line in a separate step.

The actual results of draft and spool printing are not the same, even though the same program code may be used to produce both kinds of output. For example, compare Figures 10.1 (draft printing) and 10.2 (spool printing). Both were created with exactly the same program statements. The only difference is the choice the user made when selecting the type of printing. Draft printing will not necessarily look exactly like what is seen on the screen. To duplicate screen displays exactly, use spool printing.

Spool printing does have some drawbacks. First of all, it is slower than draft printing. Secondly, "imaging" the print file to print it requires a great deal of memory. In many cases, it becomes necessary to swap much of the application program out of memory before beginning the print operation. Therefore, the program must be segmented. (Such operations are handled by the Segment Loader.) Program segmentation is a conceptually complex operation requiring intimate knowledge of where storage space has been allocated in RAM and how to perform memory management with the Memory Manager. It is an advanced operation that you should attempt only when you are comfortable with the concepts presented in this book. Details can be found in *Inside Macintosh*.

Spool printing also uses up a great deal of disk space to store the print file. Consider what happens with MacWrite: if you have a 512K Mac you can store as many as 80 pages of text in RAM, but you can only spool a document of 27 pages, assuming that there is nothing on the startup disk but the MacWrite program file and a system folder. If, though, you switch the print mode from standard to draft, the Printing Manager will not attempt to create a print file on disk, but will print directly from RAM, allowing you to print the entire 80 pages. The drawback to switching to draft printing is that it limits the type fonts and type styles that can be used.

---

**Accessing the Printing Manager**

The routines that comprise the Printing Manager are not in ROM; they are stored on disk. The Macintosh can therefore support a variety of printers. The discussion that follows, though, assumes that printing will be done on the Imagewriter printer.

Since Printing Manager routines are not in ROM, they are not called with the usual trap mechanism (i.e., an underbar followed by the name of the routine). Instead, they are external subroutines and are therefore invoked with a JSR.
Figure 10.1 Video Tapes (Draft Printing)

<table>
<thead>
<tr>
<th>Title</th>
<th>Producer</th>
<th>Date</th>
<th>Rtg</th>
<th>Numb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empire Strikes Back, The</td>
<td>Lucas Films</td>
<td>1980</td>
<td>pg</td>
<td>2</td>
</tr>
<tr>
<td>Return of the Jedi</td>
<td>Lucas Films</td>
<td>1983</td>
<td>pg</td>
<td>3</td>
</tr>
<tr>
<td>Search for Spock, The</td>
<td>Paramount</td>
<td>1984</td>
<td>pg</td>
<td>6</td>
</tr>
<tr>
<td>Star Trek: The Movie</td>
<td>Paramount</td>
<td>1978</td>
<td>g</td>
<td>4</td>
</tr>
</tbody>
</table>

This sequel outdid its predecessor, bringing new depth to its characters. The evil Darth Vader emerged as a true villain, while Luke, Leia and Han became true forces of good.

Tied up all the loose ends created in the first two films and provided a satisfactory ending to this middle trilogy (Lucas says there will be six more films).

Gives Spock a new beginning but leaves the rest of the crew in jeopardy, since the Enterprise was destroyed and the crew as mutineers.

A valiant effort to recapture the magic of the television series. Unfortunately, it fell short of expectations.

Printing requires two special files. To print with the Imagewriter, the Imagewriter file must be part of the system folder on the startup disk. This file is a resource file containing information that describes the printer. By replacing the file with one that describes another printer, an application can produce printed output on other kinds of printers.
Figure 10.2 Video Tapes (Spool Printing)

<table>
<thead>
<tr>
<th>Title</th>
<th>Producer</th>
<th>Date</th>
<th>Rating</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empire Strikes Back, The</td>
<td>Lucas Films</td>
<td>1980</td>
<td>pg 2</td>
<td></td>
</tr>
<tr>
<td>Return of the Jedi</td>
<td>Lucas Films</td>
<td>1983</td>
<td>pg 3</td>
<td></td>
</tr>
<tr>
<td>Search for Spock, The</td>
<td>Paramount</td>
<td>1984</td>
<td>pg 6</td>
<td></td>
</tr>
<tr>
<td>Star Trek: The Movie</td>
<td>Paramount</td>
<td>1978</td>
<td>g 4</td>
<td></td>
</tr>
<tr>
<td>Star Wars</td>
<td>Lucas Films</td>
<td>1977</td>
<td>pg 1</td>
<td></td>
</tr>
</tbody>
</table>

This sequel outdid its predecessor, bringing new depth to its characters. The evil Darth Vader emerged as a true villain, while Luke, Leia, and Han became true forces of good.

Tied up all the loose ends created in the first two films and provided a satisfactory ending to this middle trilogy (Lucas says there will be six more films).

Gives Spock a new beginning but leaves the rest of the crew in jeopardy, since the Enterprise was destroyed and the crew as mutineers.

A valiant effort to recapture the magic of the television series. Unfortunately, it fell short of expectations.

This landmark film raised our expectations with regard to what science fiction films should be. It set a new standard for special effects.

The second file, PrLink.Rel, must be linked to the application's object code. PrLink.Rel (you will find it on MDS2 in the Sample Programs folder along with the Printing Sample program) contains the machine language version of the Printing Manager routines that are not a part of the application itself.
Setting up the Linker Control File for an application that supports printing is only marginally more complex than what was created for the Sample program. For example, a Linker Control File to handle the Video Tape Index appears as:

```plaintext
Tapes.Rel
PrLink.Rel
$
```

### Data Structures for Printing

Each printing job uses a rather complex data structure known as a print record. It is made up of a few single parameters and a number of subrecords. Equates for the field names are in the file PrEqu.Txt, which should be INCLUDEd in your source code.

The fields in a printing record are filled in three ways:

1. An application can store information directly into the record.
2. The Printing Manager routine `PrintDefault` can be used to fill in default information.
3. The user can change information in some fields by making selections from the standard Style and Job dialogs (Figures 10.3a and 10.3b).

The top-level structure of a print record appears as:

```plaintext
TPrint = RECORD
  iPrVersion: INTEGER;  // Printing Manager version
  prInfo: TPrInfo;      // subrecord for printer information
  rPaper: Rect;         // boundary coordinates of printer paper
  prStl: TPrStl;        // subrecord for style information
  prInfoPT: TPrInfo;    // copy of printer information subrecord
  prXInfo: TPrXInfo;    // subrecord for band information
  prJob: TPrJob;        // subrecord for job information
  printX: ARRAY [1..19] of INTEGER;  // used internally
END;
```
This is the standard Style Dialog displayed and managed by a call to the Printing Manager routine `PrStlDialog`.

**Figure 10.3(a) Standard Style Dialog**

This is the standard Job Dialog displayed and monitored by a call to the Printing Manager routine `PrJobDialog`.

**Figure 10.3(b) Standard Job Dialog**
The data types which begin with T are pointers to subrecords. Actually, each subrecord is allocated sequentially. For example, the printer information subrecord begins two bytes from the beginning of the printer record and occupies the next 14 bytes (see below for details). The rectangle which describes the boundaries of the printer paper occupies the next eight bytes, from byte 16 through byte 23. The style information subrecord follows immediately, beginning with byte 24.

The **printer information subrecord** contains information about the printer being used in this particular printing job:

```plaintext
TPrInfo = RECORD
  iDev: INTEGER;     information about the printer driver
  iVRes: INTEGER;    vertical resolution of printer
  iHRes: INTEGER;    horizontal resolution of printer
  rPage: Rect;       boundaries of actual printing surface
END;
```

These parameters are filled when an application initializes the Printing Manager. The last three can be changed by the user through the standard Style dialog.

Generally, the only field of the printer information subrecord that an application will use directly is rPage, a rectangle that describes the coordinates of the actual surface available for printing. Its top left coordinates are always 0,0. It is somewhat smaller than rPaper, which contains the coordinates of the physical printer paper. This means that the top left coordinates of rPaper will be negative.

The printer information subrecord is duplicated in the print record. The copy is used internally by the Printing Manager during the printing process.

The **style information subrecord** contains parameters that further describe the paper being used:

```plaintext
TPrStl = RECORD
  wDev: TWord;       used internally
  iPageV: INTEGER;   height of printer paper
  iPageH: INTEGER;   width of printer paper
  bPort: SignedByte; port to which printer is connected
  feed: TFeed;       type of paper (e.g., cut sheet or pin feed)
END;
```

iPageV and iPageH refer to the physical dimensions of the printer paper, expressed in 120ths of an inch. They are set when the user makes choices from the standard Style dialog. bPort indicates whether the printer being used is connected to the printer or the modem port.

The final parameter, feed, is set from the standard Job dialog. The user chooses either continuous or single sheet. If single-sheet is selected, the Printing Manager will automatically pause between pages so the user can insert paper.
Parameters that describe a specific printing job are found in the job information subrecord:

```plaintext
TPrJob = RECORD
  ifstPage: INTEGER; first page to print
  ilstPage: INTEGER; last page to print
  icopies: INTEGER; number of copies to print
  bJDocLoop: SignedByte; 0 if draft, 1 if spoiled
  fFromUser: BOOLEAN; source of printing request
  pldleProc: ProcPtr; pointer to background procedure
  pfileName: TPStr80; name of spool file
  iFileVol: INTEGER; volume reference number
  bfileVers: SignedByte; version of spool file
  bJobX: SignedByte; unused
END;
```

Some of these parameters are set through the standard Job dialog. Others should be stored directly to the printer record.

- `ifstPage`, `ilstPage`, and `icopies` are selected by the user from the standard Job dialog. For spool printing, the system will automatically check the `icopies` field and print the correct number of copies. When an application does draft printing, however, the application must check `icopies` and implement multiple copy printing within its own code.

- `bJDocLoop` is also set by the user from the standard Job dialog. During calls to routines that actually create printed images, regardless of whether an application is doing draft or spool printing, the system will check `bJDocLoop` and direct the material being printed to the appropriate output device. The application must then explicitly examine the contents of `bJDocLoop` to decide whether to print a spool file.

- `fFromUser` indicates the source of the printing request. If the `fFromUser` byte is set true, then the request came from within the application; if the byte is clear, then the printing request came from the Finder. This parameter is set by the system.

Creating a document that can be printed from the Finder requires special preparation. The Finder must be able to identify which application created the document in order to launch that application to perform the print activity. The Finder looks at the document to examine its creator type, a unique four-character sequence that identifies an application. If the Finder can't find an application with a matching type, it displays the alert box “An application can't be found to open this document.” Creator types are assigned by Macintosh Technical Support so they will be unique across all Macintosh applications.

- `pldleProc` is a pointer to the routine that should execute in the background of the printing task. This is based on the idea that printing is a fairly slow operation; printer output happens at significantly slower speeds than activities which happen completely in main memory. Therefore, the CPU will have some idle time while it waits for a printer operation to finish. The background procedure can be anything
appropriate. If the `pIdleProc` is set to 0, the Mac will run its default background procedure. This routine periodically checks the keyboard to see if the cloverleaf-period has been pressed to interrupt printing.

`pFileName` is a pointer to the name of the spool file. By default the Printing Manager fills this field with a pointer to “Print File.” If an application will have more than one spool file on disk at any given time, then this parameter can be changed by storing directly to the print record. A spool file name contains no more than 80 characters; the first byte of the string must be a length byte.

`iFileVol` identifies the physical disk volume on which a spool file is stored. `bFileVers` refers to the version number of the spool file. Volumes and file versions are discussed in more detail in Chapter 11.

The final subrecord is the **band information subrecord**. In this context, the term `band` refers to a strip cut from a page. It takes an enormous amount of memory to print from a spool file, far more than will fit in memory at a single time. Therefore, a page to be printed is broken up into a series of strips called bands. The bands can run from right to left, left to right, top to bottom, or bottom to top. Bands can then be brought into memory one at a time and printed individually. An application will rarely need to access the individual fields of the band information subrecord. Its parameters are set by the Printing Manager.

---

**Programming Technique — Packing an Equates File**

Offsets for all fields in a print record are assigned symbolic addresses in the file `PrEqu.Txt`. However, unlike the other equates files, there is no packed version of the printer equates on MDS2. Packed symbol files are identified by the `.D` extension. They are created from text files, like `PrEqu.Txt`, by the application PackSyms. Packing an equates file will speed up the assembly process and will also save disk and memory space.

Packing an equates file is a two step process. First, the text version of the equates file is assembled into a symbol file with an extension of `.Sym`. Then the symbol file is packed by PackSyms.

The creation of a symbol file is controlled by the assembler directive `.DUMP`. `.DUMP` places all equates in the current program into a file with the `.Sym` extension. For example, assembling this code will create the file `PrEqu.Sym`:

```asm
INCLUDE .DUMP PrEqu.Txt ;get the text of the equates file
PrEqu ;create the symbol file
```

The two-line file above should be saved with the name `PrEqu.Asm`. It can then be assembled. Note that the program does not have to be linked or run to create `PrEqu.Sym`; assembling is enough.
To do the actual packing, run the PackSym program that comes on MDS1. Choose the "Select input" option from the File menu and double click on the name of the file that should be packed, in this case PrEqu.Sym. PrEqu.Sym will be packed, but not automatically saved as PrEqu.D. When the packing of the file is completed, you can either select another file to pack by choosing "Select input" again, or you can save all files that have been packed during the current run. Choose "Select output" from the File menu to save the packed file. The system will display a file name — the name of the last file packed with a .D extension. Either confirm the file name by hitting the Enter key or enter another filename.

---

**Establishing Print Records**

Space for a print record is allocated in the application heap. That means that an application doesn't need to set aside a location for the entire 120 byte record but merely a handle to that location. The handle to the print record is created with a Memory Manager routine, NewHandle;

```
FUNCTION NewHandle (logicalSize:Size) : Handle;

NewHandle is an operating system routine. It takes one parameter — the number of bytes of storage that should be allocated — that is placed in D0. It returns a handle to that storage area in A0. Assuming that PrEqu.txt has been INCLUDEd in the source code, the constant iPrintSize contains the size of a print record. To set aside space for a print record, then:

MOVE.L   #iPrintSize,D0
__NewHandle
MOVE.L   A0,PrintRecordHandle(A5) ;save handle
```

A word of caution is in order here with regard to storage space while using the Printing Manager. It is true that it is good practice to place the storage for all locations to which an application will write in the applications globals area (i.e., they should be allocated with DS rather than DC). Nevertheless, the Printing Manager has a bad habit of altering values stored in the applications globals area. To put it bluntly, it trashes storage locations that it has no business touching. If you find that a particular value is mysteriously changed after a call to a Printing Manager routine, allocate its storage with DC. Though the examples in this chapter will use storage locations in the applications globals area, be aware that on occasion you may have to resort to writing to the code portion of an application.
The Sequence of Printing Manager Routines

A single printing activity is bounded by calls to routines that open and close the Printing Manager. The printing of one document is surrounded by calls to routines that open and close documents. Printing a single page is bounded by calls that open and close a page. This nested arrangement of procedure and function calls is diagrammed in Figure 10.4.

Figure 10.4 Nested Printing Manager Calls
In general, an application will organize a printing activity in the following order:

1. Open the Printing Manager (PrOpen)
2. Allocate a print record (NewHandle)
3. Fill the print record with default information (PrintDefault)
4. Present standard Style dialog to user to fill in more information (PrStlDialog)
5. Present standard Job dialog to user in to finish collecting print record parameters (PrJobDialog)
6. Open a document (PrOpenDoc)
7. Open a page (PrOpenPage)
8. Print the page
9. Close a page (PrClosePage)
10. Repeat steps 7 through 9 until all pages in the document have been printed
11. Close the document (PrCloseDoc)
12. If spool printing was done, image and print the spool file (PrPicFile)
13. Repeat steps 6 through 12 until all documents have been printed
14. Free the storage used by the print record (DisposHandle)
15. Close the Printing Manager (PrClose)

Those Printing Manager routines that return result codes do so in D0. A value of 0 indicates no error. The only other error unique to the Printing Manager is −108, which indicates that there wasn't enough heap space to complete the requested operation. All other errors generated by calls to Printing Manager routines are represented by Resource Manager result codes (see Table 10.1).

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Decimal Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No error</td>
</tr>
<tr>
<td>FFFFFFF4C</td>
<td>-180*</td>
<td>Not enough memory to image and print spool file</td>
</tr>
<tr>
<td>FFFFFFF4Ø</td>
<td>-192</td>
<td>Resource not found (generally means something is wrong with the printer resource file)</td>
</tr>
<tr>
<td>FFFFFFF3F</td>
<td>-193</td>
<td>Printer resource file is missing</td>
</tr>
</tbody>
</table>

*This is a Printing Manager result code, not a Resource Manager result code.
Opening and Closing the Printing Manager

An application should call **PrOpen** and **PrClose** only once — at the very beginning and very end of printing activity. Neither routine takes any parameters and both are therefore called by a simple:

```assembly
JSR PrOpen
```

or

```assembly
JSR PrClose.
```

**PrOpen** opens both the printer driver and the printer resource file. **PrOpen** will do nothing, however, if either of the two things are missing or there is a problem with the printer resource file. A value of 0 in DO indicates that the call to **PrOpen** was successful. Otherwise, the routine returns one of the Resource Manager error codes.

Collecting Information for the Print Record

The first step in assembling the necessary information to complete a printing operation is to fill the fields of the print record with the default values for the parameters. These are stored in the printer resource file. They include the last selections made from the standard Style and Job dialog boxes. **PrintDefault** needs only the handle to the print record:

```assembly
PROCEDURE PrintDefault (hPrint: THPrint);
```

As with all other Printing Manager routines, the parameter is placed on the stack and the routine called with a **JSR**:

```assembly
MOVE.L PrintRecordHandle(A5), -(SP)
JSR PrintDefault
```

Once the print record has been filled with the default information, the user has the opportunity to change it through the standard Style and Job dialogs. Both dialog boxes are predefined within the Printing Manager and do not need to be included in an application's resource file.
Usually, the Style dialog box is presented first:

**FUNCTION PrStiDialog (hPrint: THPrint) : BOOLEAN;**

The function displays the dialog box and returns a boolean result indicating whether the user closed the dialog box with the ENTER button (a value of true) or the CANCEL button (a value of false). If the function result is true, any changes made by the user will be reflected in the print record.

The code to display the Style dialog box might be:

```assembly
CLR – (SP) ; space for boolean result
MOVE.L PrintRecordHandle(A5), – (SP)
JSR PrStiDialog
```

The standard Job dialog is handled in precisely the same way as the Style dialog. It, too, is a function that returns a boolean result:

**FUNCTION PrJobDialog (hPrint: THPrint) : BOOLEAN;**

Any changes made by either **PrStiDialog** or **PrJobDialog** are reflected not only in the print record in RAM, but in the printer resource file as well. The next time an application attempts to print from this same disk, it will be presented with the new values as default values.

---

**Opening and Closing a Document**

Macintosh printing actually involves opening a special kind of grafport - called a *printer port* - in which text and graphics images are drawn. The exact nature of the printing port depends on whether the user selected draft or spool printing. Nonetheless, it is **PrOpenDoc** that establishes the printing port and makes it the current grafport:

**FUNCTION PrOpenDoc (hPrint: THPrint; pPrPort : TPPrPort; plOBuf:Ptr) : TPPrPort;**

**hPrint** is the handle to the print record. **pPrPort** is a pointer to storage for the printer port. If this parameter is 0, the Printing Manager will allocate a new printer port and return a handle to it as the function's result. An application does not need to explicitly set aside storage for a printer port; only space for its pointer (one longinteger location) is required.
plOBuf is important for spool printing. It is a pointer to a portion of memory that should be used as temporary storage when creating a spool file. (I/O buffers are discussed in detail in Chapter 11.) Normally, it is not necessary to supply an explicit I/O buffer for spooling: the value of plOBuf will be 0, telling the system to use the volume's I/O buffer.

To open a printing port:

```
CLR.L  - (SP) ;space for printer port pointer
MOVE.L PrintRecordHandle(A5), - (SP)
CLR.L  - (SP) ;new printer port will be created
CLR.L  - (SP) ;use the volume I/O buffer
JSR    PrOpenDoc ;call the routine
MOVE.L (SP) +,PrPortPtr(A5) ;recover printer port pointer
```

PrCloseDoc terminates a printing task. If draft printing, it sends a form feed to the printer. If spool printing, it closes the spool file. If the spooling was unsuccessful, it closes and then deletes the spool file. To call the routine, an application needs only the pointer to the printer port:

```
PROCEDURE PrCloseDoc (pPrPort: TPPrPort);
```

---

### Printing a Single Page

The real work in programming Macintosh printing activities comes in laying out the printed page. Each page begins with a call to PrOpenPage:

```
PROCEDURE PrOpenPage (pPrPort:TPPrPort;  
pPageFrame: TPRect);
```

pPrPort is nothing more than the pointer to the printer port that was returned by PrOpenDoc. pPageFrame is a rectangle that describes boundaries within which QuickDraw images will be drawn. When a spool file is printed, this rectangle will be scaled to fit onto the printer paper. The easiest way to handle pPageFrame is to set it to 0. In that case, the Printing Manager will use the page rectangle (rpage) from the printer records as the page frame. The page will then not be scaled when it is printed.

PrOpenPage checks the page range parameters in the print record (iFstPage,iLstPage). If the page to be printed doesn't fall within that range, no printing will be performed.

The actual printing on a page is handled by QuickDraw. An application can draw to the printer port using any QuickDraw routines, just as it would to the screen. Remember that PrOpenDoc makes the printer port the current grafport.
That means that any calls to QuickDraw routines that draw something will affect the printer port until it is closed by `PrCloseDoc`. Graphics printing (always spooled) will use the coordinates of the `rPage` rectangle to ensure that printed images are within the boundaries of the page.

Printing straight text, especially if draft printing is possible, requires a somewhat different method for establishing spacing between lines and determining when a page has been filled. When printing, it is not possible to use TextEdit to space and justify characters; text is printed with either `DrawChar`, `DrawString`, or `DrawText`. The latter routine is the easiest to use if the text to be printed is stored in a text edit record. `DrawString` is convenient when the text is not stored in main memory in the format in which it will be printed.

The position on the page at which text should be printed is set with `MoveTo`, the QuickDraw routine that handles cursor placement. An application must therefore carefully compute the size of the font being used to determine how far apart lines of text must be.

Information about the size of characters in a font can be retrieved with the QuickDraw routine `GetFontInfo`:

```assembly
PROCEDURE GetFontInfo(V AR info: Fontinfo);
```

This procedure returns an eight-byte record, a pointer to which should be placed on the stack before calling the routine:

```assembly
PEA FontInfoStorage(A5)
    _GetFontInfo
```

`GetFontInfo` provides four parameters about the font for the current grafport, each expressed in terms of pixels: ascent (how many pixels letters like "h" rise), descent (how many pixels letters like "y" descend below the line), maximum character width, and the number of pixels between the descent of one line and the ascent of the next line below it (known as "leading"). The `FontInfo` record structure is:

```assembly
FontInfo = RECORD
    ascent: INTEGER;
    descent: INTEGER;
    widMax: INTEGER;
    leading: INTEGER;
END;
```

Offsets into the `FontInfo` record are available in the QuickDraw equates file, which should be INCLUDEd at the beginning of the source code.

The height of a line is the sum of ascent, descent, and leading:

```assembly
MOVE  FontInfoStorage + ascent(A5),D4
ADD   FontInfoStorage + descent(A5),D4
ADD   FontInfoStorage + leading(A5),D4
```
D4 contains the height, in pixels, of a single line of text in whatever font is set for the current grafport. In this case, the current grafport is the printer port. Assuming that D3 is used to hold the vertical position of the pen (read "pen" as cursor or print head, if you like), then D3 will be incremented by the quantity in D4 every time a line is printed.

One other parameter is necessary for text printing — the coordinate, in pixels, of the bottom of the page. This will be compared to the current vertical position (stored, in this example, in D3) to determine if a full page has been printed. The coordinate of the bottom can be retrieved from the print record:

```
MOVE.L PrintRecordHandle(A5),A0 ;get handle
MOVE.L (A0),A0 ;get pointer
MOVE prInfo + rPage + bottom(A0),D6 ;get bottom
```

How does this work? The first step retrieves the handle to print record. The second de-references the handle to get the pointer to the record. At this stage in the process, the actual address of the start of the print record is in A0. The third step uses Address Register Indirect with Offset addressing to locate one precise piece of information. `prInfo` and `rPage` are constants defined in the printer equates file. `prInfo` stands for the number of bytes the printer information subrecord is offset from the beginning of the print record. `rPage` is an offset within the printer information subrecord. `rPage` is a rectangle; it has four components — top, left, bottom, right — that are defined in the QuickDraw equates file. `bottom`, therefore, refers to the third field in the rectangle, `rPage`. To compute the address for the `MOVE`, the Macintosh adds the three constants to obtain the offset and then adds that quantity to the contents of A0.

Since some characters do descend below the printing line, it is wise to subtract the descent from the bottom coordinate to ensure that characters that do descend will be completely printed:

```
SUB FontInfoStorage + descent(A5),D6
```

The initial vertical position for printing text is not 0; it is down the height of a single line from the top of the printing page. Therefore, assuming that D3 is being used to hold the vertical position of the pen, it should be initialized to the height of a line before any printing activity begins:

```
MOVE D4,D3
```

### Moving the Pen

Printing or drawing, whether on the screen or on paper, is only possible if you have control over where the display activity will begin. The QuickDraw routine `MoveTo` positions Mac's pen anywhere within a grafport. Remember that a printing port is a special kind of grafport and that once a printer port has been
opened, any calls to QuickDraw routines that affect output (e.g., drawing or moving the pen) will affect the printed page.

To use **MoveTo**, an application must supply the horizontal and vertical coordinates, in pixels, of the new pen position:

```pascal
PROCEDURE MoveTo (h, v: INTEGER);
```

**h** is the horizontal coordinate; **v** is the vertical coordinate.

In spool printing, there is virtually no limit to how the pen can be moved, since writing to the spool file allows random access. When draft printing, however, be aware of the abilities of the specific printer being used. Some printers, like the Imagewriter, can move the platten backwards; that is, it is possible to pass the Imagewriter a vertical coordinate less than the most recent vertical coordinate. Many printers, however, are not only unable to move the platten backwards, but are unable to backspace; that is, they cannot accept a horizontal coordinate less than the most recent horizontal coordinate.

Though the Imagewriter can do more or less random print head movement, sending print images directly to the printer in that manner will significantly slow down the printing process. Therefore, draft printing is really not suitable for a printing activity that includes graphics.

### Printing Text with DrawString

**DrawString** is a QuickDraw routine that will print text from left to right, beginning at the current position of the pen. Like the other QuickDraw routines that print characters, it does no formatting. In other words, the application must decide how many characters will fit on a single printed line.

Calling **DrawString** requires only a pointer to the text of the string:

```pascal
PROCEDURE DrawString (s: Str255);
```

It is important to realize that the data type of the string (Str 255) requires that the first byte in the string be a length byte. The system checks that length byte to determine the number of characters to print.

The Video Tape Index program uses **DrawString** to print information about a single video tape. The data for that print line is found in the TapeArray in RAM and must therefore be reformatted before it is printed.

About 100 characters of 12-point type will fit across a 8 1/2" piece of paper. Therefore, the Video Tape Index sets up a 102-character print string. The first byte will be a length byte; the last byte is an extra byte appended to keep the total length of the string even. The strategy to assemble and print a single line of data is therefore:

1. Fill print line with blanks
2. Move each field from its storage location in TapeArray to its proper position in the print string
3. Set font characteristics (TextFont, TextFace, TextSize)
4. Move the pen (MoveTo)
5. Draw the string (DrawString)
6. Increment the register that holds the vertical position of the pen

The code for this procedure appears in Listing 10.1. The subroutine ClearPrintLine (a) fills the print string with blanks. It uses a pre-defined string of 102 blanks (stored as PrintLineMask) which is simply moved to the print string with BlockMove (b). ClearPrintLine also installs a length byte in the print string (c).

**Listing 10.1 Printing One Record from TapeArray**

(a) ClearPrintLine
    LEA PrintLineMask,A0
    LEA PrintLine(A5),A1
    MOVE #102,D0
    _BlockMove
(b) MOVE.B #100,Printline(AS)
    RTS
(d) PrintOneRecord
(e) JSR ClearPrintLine
    LEA TapeArray(A5),A2
    MOVE.L D7,-(SP) ;save record counter
    MOVE D7,D5
(f) JSR ComputeAddress2 ;address returned in A4 (see Listing 5.1 or 6.1)
    MOVE.L (SP)+,D7 ;restore record counter
(g) MOVE.L A4,A0 ;start of record
(h) LEA PrintLine+12(A5),A1
(i) MOVE #30,D0
(j) _BlockMove ;moves TapeName
    MOVE.L A4,A0
    ADD.L #0Producer,A0
    LEA PrintLine+44(A5),A1
    MOVE #20,D0
    _BlockMove ;moves Producer
    MOVE.L A4,A0
    ADD.L #0ReleaseDate,A0
    LEA PrintLine+66(A5),A1
    MOVE #4,D0
    _BlockMove ;moves Date
(continued)
Actual printing of a single line is handled by PrintOneRecord, beginning at (d). On input, the number of the TapeArray record to be printed is stored in register D7. PrintOneRecord begins by calling ClearPrintLine (e) to erase the previous contents of the print string and reset the length byte. Then it assembles the data from TapeArray into their proper positions in the print line.

To do so, PrintOneRecord must first compute the main memory address of the particular record being printed. Subroutines to compute such addresses already exist as part of the straight-insertion sort (see Listing 5.1 or 6.1) and therefore can simply be called rather than rewritten (f). Using the address returned by ComputeAddress2, PrintOneRecord then moves one field at a time with repeated calls to BlockMove. The starting address of the field being moved is loaded into A0 (g), the starting position for the field in the print string into A1 (h), and the length of the field into D0 (i). The transfer is completed with the operating system call (j).

Steps (g) through (j) are repeated for each field. Note, however, that there is an extra step required for all fields but the first one. The offset of the field in the record must be added to the starting address of the record. For example, at (k) PrintOneLine adds the offset of the Rating field. The offsets have been equated to symbolic addresses for ease of use.

The actual printing process begins at (l) with the set-up sequence to move the pen. The horizontal coordinate is moved onto the stack; its value is 0 since printing
should begin at the far left-hand side of the page. The vertical coordinate follows it (m); it's value is stored in D3 while the page is being printed. A call to **MoveTo** actually moves the pen (n). To draw the print string, a pointer to the string is pushed onto the stack (o) followed by the call to **DrawString** (p).

Only one task remains. The register containing the vertical print coordinate, D3, must be incremented by the height of a single print line to prepare for printing the next line. Therefore, the contents of D4 (the register set aside to contain the height of a print line) are added to D3 (q). PrintOneLine can then return to the part of the program that called it.

There is an alternative to using one string to print an entire line: rather than moving the text to be printed into a single place, each string can be drawn individually. In this case, the strategy is:

1. Set font characteristics (**TextFont**, **TextFace**, **TextSize**)
2. Move the pen to the beginning horizontal and vertical position of the line (**MoveTo**)
3. Draw the first string (**DrawString**)
4. Set font characteristics if desired (**TextFont**, **TextFace**, **TextSize**)
5. Move the pen horizontally (using the same vertical coordinate) to the position of the next set of characters on the line (**MoveTo**)
6. Draw the string (**DrawString**)
7. Repeat steps 4 through 6 until the entire line is printed

The advantage to this second strategy is that you can vary the font characteristics of the text across the line, something which is not possible when the entire print line is a single string.

**Printing Text with DrawText**

**DrawText** prints an entire line of text from a specified storage location in RAM. It differs somewhat from **DrawString**. When using **DrawString**, an application pushes a pointer to the text onto the stack; the length of the string is imbedded in the string itself. **DrawText** requires a pointer to the starting location of an entire block of text, an offset into that block, and the number of bytes to print:

```plaintext
PROCEDURE DrawText (textBuf:QDPtr;firstByte,byteCount: INTEGER);
```

It is therefore best suited to printing text that is stored in a text edit record.

Since **DrawString** does not do any text formatting, it does not know where **TextEdit** marked the end of lines. Therefore, an application must use two pieces of
information from the text edit record to control the printing operations — the total number of lines in the text and the character positions of the line starts.

The strategy to print text from a text edit record is:

1. Get handle to the text edit record (TEGetText)
2. De-reference handle to get a pointer to the text
3. Retrieve total number of lines in the text
4. Initialize a line counter
5. Check to see if the counter contains the number of the last line. If so, jump to step 11.
6. Retrieve starting position of current line
7. Retrieve starting position of next line
8. Subtract starting position of current line from starting position of next line to obtain number of characters in current line
9. Print the line (DrawText)
10. Increment the line counter. Jump to step 5.
11. Retrieve total number of characters
12. Subtract starting position of last line from total number of characters + 1 to obtain number of characters in last line
13. Print the line (DrawText).

The Video Tape Index's implementation of this procedure to print annotations appears in Listing 10.2. The code that begins with EnoughRoom is initiated after the program determines that there is enough room left on the page to print the entire annotation.

The first step is to call PrintOneLine to print the data from TapeArray that applies to the tape in question (a). A blank line must then appear between the TapeArray data and the first line of the annotation. Getting a blank line is straightforward — the register holding the vertical position of the pen (D3) is simply incremented by the height of a single line (held in D4) without printing any text (b).

Printing the annotation requires a loop that uses the total number of lines in the annotation as a target value. Therefore, before the actual printing can begin, the program must retrieve the number of lines from the text edit record. The three statements beginning at (c) get the handle to the text edit record and de-reference it to obtain a pointer to the record. The number of lines in the text is then stored in D0 (d). D1 is initialized to act as a line counter (e).

As indicated in the printing procedure described above, the last line in the text must be handled separately from all other lines. The first activity in the printing loop must therefore be a "look ahead" to determine if the last line has been reached. The line counter is incremented by 1 (f) and compared to the total number of lines in the
text (g). If the two values are equal, the program branches out of the loop to print the last line (h). Assuming that the last line has not been reached, the line counter is decremented to restore the correct line number (i).

**Listing 10.2 Printing an Annotation that is Stored in a TextEdit Record**

```
EnoughRoom
   MOVEM.L D2/D7,-(SP)
(a)  JSR PrintOneRecord ;(Listing 10.1)
(b)  ADD D4,D3 ;get a blank line
   MOVEM.L (SP)+,D2/D7
(c)  LEA AnnotationTextHandle,A2
   MOVE.L (A2),A2
   MOVE.L (A2),A2
(d)  MOVE teNLines(A2),D0 ;get number of lines again
(e)  MOVE #0,D1

AnotherLine
   MOVEM.L D2/D4,-(SP)
(f)  ADDQ #1,D1 ;look at next line
(g)  CMP D1,D0 ;at last line?
(h)  BEQ LastLine ;restore current line #
(i)  SUBQ #1,D1
   MOVE #2,D4
(j)  MULU D1,D4 ;line starts are stored as integers
(k)  MOVE teLines(A2,D4),D2 ;line start of this line
   ADDQ #2,D4
(l)  MOVE teLines(A2,D4),D5 ;start of next line
(m)  SUB D2,D5 ;D5 has number of bytes

(n)  CLR.L -(SP)
(o)  MOVE.L AnnotationTextHandle,-(SP)
(p)  _TEGetText
   MOVE.L (SP)+,A6 ;get handle to annotation text
   MOVE.L (A6),A6 ;de-reference to get pointer
(q)  MOVE #20,-(SP) ;annotation is indented 20 pixels
(r)  MOVE D3,-(SP)
(s)  _MoveTo
   MOVEM.L D0/D1/D7/A2/A6,-(SP)
(u)  MOVE.L A6,-(SP) ;pointer to text
(v)  MOVE D2,-(SP) ;starting position
(w)  MOVE D5,-(SP) ;number of bytes to print
(x)  _DrawText
   MOVEM.L (SP)+,D0/D1/D7/A2/A6
   MOVEM.L (SP)+,D2/D4
(y)  ADDQ #1,D1 ;increment line counter
(z)  ADD D4,D3 ;space to next line
(aa) BRA AnotherLine

LastLine
   SUBQ #1,D1 ;restore current line #
   MOVEM.L D1/D3/D7/A2/A6,-(SP)
   MULU #2,D1
```

(continued)
The next task is to prepare for the call to `DrawText` which will be used to print a single line. `DrawText` needs to know the starting address of the text, a byte offset into that text where printing should start, and the total number of characters to print.

The positions within the text where new lines start are stored in the text edit record as integers. The start of the first line is stored immediately after the total number of lines in the text. Therefore, the starting position of the line being printed can be found by:

1. Multiplying the line number by 2 to account for the line starts being stored as integers (j)

2. Adding that result to the starting address of the text edit record and the offset for the number of lines, `telines` (k).

Statement (k) stores the line start in D0.

In order to figure out the number of characters in the line, the program also needs the line start of the following line (l). Then it subtracts the starting position of the current line from the starting position of the next line to obtain the number of characters in the current line (m).

The final piece of data needed by `DrawText` is the starting address of the text itself. `TEGetText` will return a handle to the text in a text edit record. The program
calls it by clearing space on the stack for the handle result (n), pushing the handle to the text edit record on the stack (o), and then calling the routine (p). Once the result is pulled from the stack it must be de-referenced to obtain a pointer (q).

Before actually printing, the pen must be moved. Since the annotation is indented from the left-hand margin of the page, the horizontal position is not 0, but 20, an arbitrary indentation chosen merely because it looks nice on the page (r). The vertical coordinate is again taken from register D3, which stores the vertical position while a page is printed (s). MoveTo takes care of positioning the pen (t).

The set-up for the call to DrawText requires pushing the pointer to the text onto the stack (u), followed by the starting position in the text (v), and the total number of bytes to print (w). The call actually draws the text (x).

The program then increments the line counter (y) and the vertical position of the pen (z). This completes printing one line of the annotation. Therefore, the program must branch to print another line (aa).

Printing the last line is only slightly different from printing the other lines. The difference lies in determining how many characters are in the line. For the last line, there is no "next" line. The total number of characters in the last line is equal to the starting position of the last line (bb) subtracted from the total number of characters in the text plus 1 (cc). The remainder of the procedure is exactly the same.

Once the annotation is printed, the final task is to print two blank lines beneath it. This is accomplished by simply incrementing the vertical position of the pen twice (dd).

---

**Finishing a page**

Generally, an application will decide to finish printing a page when the vertical pen coordinate is greater than or equal to the bottom of the page coordinate, or when the entire document has been printed. (The Video Tape Index closes a page when all records from TapeArray have been printed, even though an entire physical page may not be filled.) When that occurs, a call to PrClosePage is necessary. If the application is draft printing, the call will eject the current page and, if printing from single sheets, will prompt the user to insert another sheet. If the application is spool printing, the call will simply close the printer port for the page being printed.

Calling PrClosePage needs only a pointer to the printer port as a parameter:

**PROCEDURE PrClosePage (pPrPort: TPrPort);**

At this point, the application must decide whether there are more pages to print or whether the document should be closed.
Imaging and Printing Spool Files

As far as the Macintosh is concerned, the term *imaging* refers to the process of taking the picture of a printed page that is stored in a spool file and turning it into an array of dots of the right size and shape. That array can then be sent to the printer one band at a time, so the printer can easily print the page from the top down.

Assuming that there is sufficient memory to hold the image of a single band from a page of the spool file, imaging and printing the file is a simple process — it requires only a call to **PrPicFile**. This routine takes care of breaking the spool file into bands for printing, bringing the bands into memory one by one, and printing them.

The format of **PrPicFile** is:

```pascal
PROCEDURE PrPicFile (hPrint: THPrint; pPrPort: TPPrPort: plOBuf: Ptr; pDevBuf: Ptr; VAR prStatus: TPrStatus);
```

The first parameter, **hPrint**, is the handle to the print record. The second parameter, **pPrPort**, looks, at first, to be the same as the printer port used to create the spool file, but it is not. The printer port created by **PrOpenDoc** was closed by the call to **PrCloseDoc**. **PrPicFile** requires its own printer port. A value of 0 for **pPrPort** will instruct the system to allocate its own printer port.

**plOBuf** is a pointer to the area in memory which should be used to hold information as it is read from the disk. Though application may set aside its own area, generally it's just as easy to pass a 0 for this parameter, allowing the system to use the disk volume's buffer for this purpose. **pDevBuf** is also a pointer. It locates an area known as the "band buffer" that is used to hold data to be printed. Passing a 0 for **pDevBuf** will cause the system to allocate the buffer on the heap.

The variable parameter **prStatus** is a pointer to a printer status record. The printer status record monitors the activity of the system while it is printing from a spool file. The structure of a status record is:

```pascal
TPrStatus = RECORD
  iTotPages: INTEGER; total number of pages
  iCurPage: INTEGER; page being printed
  iTotCopies: INTEGER; number of copies to print
  iCurCopy: INTEGER; copy being printed
  iTotBands: INTEGER; number of bands per page
  iCurBand: INTEGER; band being printed
  fPgDirty: BOOLEAN; TRUE if page is being printed
  fImaging: BOOLEAN; TRUE if page is being imaged
  hPrint: THPrint; handle to print record
  pPrPort: TPPrPort; pointer to printing port
  hPic: PicHandle; used internally — do not change
END;
```
An application must allocate space for the entire printer status record:

```
PrinterStatusRec DS.B iPrStatSize ;where iPrStatSize is
equated to the total number
of bytes in a printer
status record
```

The printer status record is generally most useful to applications that are running their own background procedure. The background procedure can repeatedly check the fields of the printer status record to determine the status of the printing process. If an application relies on the default background procedure, the printer status record is of minimal importance.

The code to image and print a spool file from the Video Tape Index program appears as follows:

```
MOVE.L PrintRecordHandle(AS), - (SP) ;put handle on stack
CLR.L - (SP) ;system allocates its own printing port
CLR.L - (SP) ;system uses volume I/O buffer
CLR.L - (SP) ;system uses its own band buffer
PEA PrinterStatusRec(A5) ;push address
JSR PrPicFile ;image and print
```

---

**Completing the Printing Task**

When an application has finished printing, regardless of whether it has draft printed or imaged and printed a spool file, there is no longer any need to retain the print record on the heap. The storage held by the print record should therefore be released through a call to `DisposHandle`:

```
PROCEDURE DisposHandle (h: Handle);
```

The handle to the print record must be in A0 before calling the routine:

```
MOVE.L PrintRecordHandle(AS), A0 ;get the handle
__DisposHandle ;release the heap storage
```

The final step is, as discussed earlier in this chapter, to call `PrClose` to close the Printing Manager.
Putting it All Together — BannerPrint

BannerPrint is a demonstration program that prints large upper-case letters sideways on 8 1/2 x 11 computer paper. To create a banner, the sheets of paper must be separated and then taped together to hide the gaps between the pages. Source code for BannerPrint appears in Listing 10.3; its resource file can be found in Listing 10.4.

BannerPrint creates a small text edit window in which the user can enter upper-case letters and spaces. The standard editing functions are supported in that window. The banner can be draft or spool printed; which method is used is determined by the user's choice in the standard job dialog box.

The large letters are stored as strings in the code portion of the program (i.e., they are defined as constants). There are certainly other ways to specify how the letters should be printed, but this particular method was chosen because it is easy to type in from a printed listing; you can see the shape of the letters on the screen as you work. Note that while each string has its own DC directive, it doesn't have a unique name. The symbolic address Letters refers to the entire block of letter templates.

While each line of a letter is exactly 30 characters long, all the letters are not made up of the same number of lines (i.e., "I" has only four lines, "W" and "M" have 16, and all the rest have 12). That means that the program must have some way of locating the start of a letter within the Letters block. BannerPrint uses a technique known as a "jump table."

The jump table (stored under the symbolic address JumpTable) itself consists of 26 numbers. Each corresponds to the number of bytes the start of a letter template is offset from the address assigned to Letters. Therefore, if the program knows the ordinal position of a character in the alphabet, it can look in the jump table to discover how far beyond Letters it should begin. The ordinal position a character is determined by comparing it against the letters in the alphabet; those letters are stored as OrdinalList.

BannerPrint must also have a way to determine when all the lines of a particular character have been printed; since the number of lines per character vary, it can't simply count lines printed. Instead, the program looks ahead to the next line. If the first non-blank character in the next line is different from the character being printed, then the character must be complete.

To keep it relatively short, BannerPrint was written without a number of checks that would catch user errors. It does not, for example, trap the situation where the user enters a character other than an upper-case letter or a space. It also does not prompt the user to ready the printer. For suggestions on what you can do to make the program "bullet-proof," see Problem 10.
Listing 10.3 BannerPrint

```
#include MacTraps.D
#include ToolEqu.D
#include SysEqu.D
#include PrEqu.Txt
#include QuickEqu.D

PEA -4(A5)
  _InitGraf
  _InitFonts
  MOVE.L #$80000000, D0
  _FlushEvents
  _InitWindows
  _InitMenus
  CLR.L -(SP)
  _InitDialogs
  _TEInit
  _InitCursor

CLR -(SP)
PEA 'MAL.files:BannerPrint.Rsrc'
  _OpenResFile
  MOVE (SP)+, D0

;------------------------- Set up menus -------------------------

  CLR.L -(SP)
  MOVE #1,-(SP)
  _GetMenu
  MOVE.L (SP)+, AppleHandle(A5)
  ;retrieve & store handle

  MOVE.L AppleHandle(A5),-(SP)
  MOVE.L 'DRUR', -(SP)
  _AddResMenu

  CLR -(SP)
  _InsertMenu

  CLR.L -(SP)
  MOVE #2,-(SP)
  _GetMenu
  MOVE.L (SP)+, FileHandle(A5)

  MOVE.L FileHandle(A5),-(SP)
  CLR -(SP)
  _InsertMenu

  CLR.L -(SP)
  MOVE #3,-(SP)
  _GetMenu
  MOVE.L (SP)+, EditHandle(A5)

  MOVE.L EditHandle(A5), -(SP)

;-----------end file
```
Listing 10.3 (continued)

CLR -<SP>
_InsertMenu
_DrawMenuBar

; Event loop starts here

MOVE #0,WindowFlag(R5) ; will be set if window is present

Event _SystemTask ; update desk accessories

MOVE WindowFlag(R5),D0 ; text edit window open?
BEQ NoWindow
MOVE.L TextHandle(R5),-(SP)
_TIdle

NoWindow

CLR -<SP> ; space for boolean result
MOVE #1,-<SP> ; mask to select all events
PER EventRecord(R5) ; pointer to event record
_GetNextEvent

MOVE (SP)+,D0 ; retrieve boolean result
BEQ Event ; no event

MOVE EventRecord(R5),D0 ; get event type

CMP #mButDownEvt,D0 ; mouse down event?
BEQ MouseEvent

CMP #keyDownEvt,D0 ; key down event?
BEQ KeyEvent

CMP #upDatEvt,D0 ; update
BEQ Update

BRA Event

; Handle key down events

KeyEvent

MOVE EventRecord+evtMeta(R5),D0
BTST.L #cmdKey,D0 ; command key pressed?
BNE KeyboardEquivalent

MOVE EventRecord+evtMessage+2(R5),-(SP) ; character pressed
MOVE.L TextHandle(R5),-(SP)
_TEKey ; insert character

BRA Event

KeyboardEquivalent

CLR.L -<SP> ; place for menu ID & item number
MOVE EventRecord+evtMessage+2(R5),-(SP) ; character
_MenuKey

BRA Selections ; process with mouse down selection

(continued)
Update

MOVE.L CharWindPtr(A5),-(SP)
  ;BeginUpdate

MOVE.L CharWindPtr(A5),-(SP)
  ;SetPort

PEA ViewRect(A5)
MOVE.L TextHandle(A5),-(SP)
  ;TEUpdate

MOVE.L CharWindPtr(A5),-(SP)
  ;EndUpdate

BRA Event

; Update the text window

---- Handle mouse down events -----

MouseEvent

CLR -(SP)
  ;space for "what" result

MOVE.L EventRecord+evtMouse(A5),-(SP)
  ;place where event occurred

PEA WhichWindowPtr(A5)
  ;window affected goes here

_CMPFindWindow
  ;get exact location of event

MOVE (SP)+,00
  ;recover result

CMP *inMenuBar,00
  ;in menu bar?
  BEQ MenuBar

CMP *inSysWindow,00
  ;desk accessory?
  BEQ SysEvent

CMP *inContent,00
  ;in the text edit window?
  BEQ AppIWindow

CMP *inGoAway,00
  ;close the window?
  BEQ GoAwayBox

BRA Event
  ;not an event this program handles

---- Handle events in system windows -----

SysEvent

PEA EventRecord(A5)

MOVE.L WhichWindowPtr(A5),-(SP)
  ;window posting event

_CMPSystemClick
  ;let system handle it

BRA Event

---- Handle events in content area of window -----

AppIWindow

PEA EventRecord+evtMouse(A5)
  ;place where event occurred

_CMPGlobalToLocal
  ;make local

MOVE.L EventRecord+evtMouse(A5),-(SP)
  ;coordinates now local

MOVE.L EventRecord+evtMeta(A5),D0
BTST.L *shiftKey,D0
  ;extended selection?

SNE D0
MOVE.B D0, -(SP)
  ;extend or not extend

(continued)
Listing 10.3 (continued)

MOVE.L TextHandle(5),-(SP) ;establish the selection range

; ------------------ Handle events in the menu bar ------------------

Evento.

; space for menu ID and menu item

MOVE.L EventRecord+evtMouse(5),-(SP) ; place where event occurred

MenuSelect ; find menu ID and menu item

Selections

MOVE.L (SP)+,D7 ; recover result

MOVE D7,D6 ; D6 now has menu item

SWAP D7 ; low-order word has menu ID

CLR -(SP) ; selects all menus

HiLiteMenu ; remove highlighting from menu

CMP #1,D7 ; apple menu?

BEQ AppleMenu

CMP #2,D7 ; file menu?

BEQ FileMenu

CMP #3,D7 ; edit menu?

BEQ EditMenu

BRA Event

; ------------------------ Handle desk accessories ------------------------

AppleMenu

MOVE.L AppleHandle(5),-(SP) ; menu item

MOVE D6,-(SP) ; space for desk accessory name

GetYItem

CLR -(SP) ; space for reference number

PEA DeskAccName(5) ; item name

GetDeskAcc

MOVE (SP)+,D0 ; discard result

BRA Event

; ------------------------ Handle editing ------------------------

EditMenu

SUBQ #1,D6 ; adjust item selected for SysEdit

CLR -(SP) ; space for result

MOVE D6,-(SP) ; adjusted item number

SysEdit

MOVE (SP)+,D0 ; get result

BNE Event ; system handled edit

ADDQ #1,D6 ; restore item number

CMP #3,D6 ; cut?

(continued)
BNE EditMenu2
MOVE.L TextHandle(A5),-(SP)
 TekCut
BRA Event

EditMenu2
CMP #4,06 ;copy?
BNE EditMenu3
MOVE.L TextHandle(A5),-(SP)
 _TECopy
BRA Event

EditMenu3
CMP #5,06 ;paste?
BNE EditMenu4
MOVE.L TextHandle(A5),-(SP)
 _TEPaste
BRA Event

EditMenu4
CMP #6,06 ;clear?
BNE Event
MOVE.L TextHandle(A5),-(SP)
 _TDElete
BRA Event

;------------------ Handle File Menu ------------------

FileMenu
CMP #1,06 ;New window?
BEQ NewWindow

CMP #2,06 ;Close the window
BEQ CloseWindow

CMP #3,06 ;Print the banner
BEQ Print

CMP #4,06 ;Quit
BNE Event
RTS ;return to Finder

; ------------------ Open a new window with text edit record ------------------

NewWindow
CLR.L -(SP) ;space for window pointer
MOVE #1,-(SP) ;window ID
PEA CharWindStrg(A5) ;window storage
MOVE.L #1,-(SP) ;put window in front
_GetNewWindow
MOVE.L (SP)+,CharWindPtr(A5) ;get pointer

MOVE.L CharWindPtr(A5),-(SP) ;make this the current grafport
CLR.L -(SP) ;space for text edit handle
PEA DestRect ;destination rectangle
PEA ViewRect ;view rectangle

(continued)
Listing 10.3 (continued)

_TENew
MOVE.L (SP)+,TextHandle(A5) ;establish text edit record

MOVE +1,WindowFlag(A5) ;get text handle

MOVE.L TextHandle(A5),-(SP) ;set window flag

_TEAactivate
MOVE.L TextHandle(A5),-(SP) ;activate the text window

BRA Event

; --------------------------- Close a window ---------------------------------
GoAwayBox
CLR.B -(SP) ;space for boolean result
MOVE.L WhichWindowPtr(A5),-(SP) ;window pointer
MOVE.L EventRecord+evtMouse(A5),-(SP) ;point of event
TrackGoAway ;monitor GoAway box

MOVE.L <SP>+,00
BEQ Event ;get result

CloseWindow
MOVE.L TextHandle(A5),-(SP) ;don't close

MOVE.L CharWindPtr(A5),-(SP) ;close the window

MOVE *0,WindowFlag(A5) ;close text edit record

BRA Event

; --------------------------- Print the banner ---------------------------------
Print
JSR PrOpen ;open printing manager

MOVE.L #1PrintSize,D0 ;size of print record

_MoveHandle ;space on heap for printer record

MOVE.L A0,PrRecHandle(A5) ;save handle

MOVE.L PrRecHandle(A5),-(SP) ;fill record with default info

Jsr PrintDefault ;open printing manager

CLR -(SP) ;space for boolean result

MOVE.L PrRecHandle(A5),-(SP) ;draft or spooled?

Jsr PrJobDialog ;user canceled

MOVE. <SP>+,D0 ;system will allocate port

BRA Event ;use system I/O buffer

CLR.L -(SP) ;get the pointer

MOVE.L PrRecHandle(A5),-(SP) ;system will allocate port

CLR.L -(SP) ;use system I/O buffer

Jsr prOpenDoc

MOVE.L <SP>+,PrPortPtr(A5) ;get the pointer

MOVE #monaco,-(SP) ;set the font

_TextFont

(continued)
MOVE #12,-(SP)

TextSize ;set the size

PEA FontInfoStrg(A5)

GetFontInfo ;get size of font
MOVE FontInfoStrg+ascent(A5),D4
ADD FontInfoStrg+descent(A5),D4
ADD FontInfoStrg+leading(A5),D4 ;height of line
MOVE.L PrRecHandle(A5),A0
MOVE.L (A0),A0
MOVE.FontInfo+Page+bottom(A0),PageBottom(A5) ;bottom of page
MOVE.L TextHandle(A5),A0 ;handle to text edit record
MOVE.L (A0),A0
MOVE telength(A0),D7
MOVE.L teTextH(A0),A0
MOVE.L (A0),A0
MOVE.0,00 ;handle to text edit record

NewPage
JSA StartAPage ;begins new page at start of character

OuterLoop
MOVE.B (A0,D0),D6
MOVE.0,01
LEA OrdinalList,A1
CMP.B BNE MOVE.MULU MOVEM.BRA (A1,D1,D6)

Loop1 CMP.B (A1,D1),D6 ;attempt to identify character
BEQ Found ADDQ #1,D1
BRA Loop1 ;character not found

Found LEA JumpTable,A1
MULU #2,D1
MOVE (A1,D1),D5
LEA Letters,A6 ;starting address of letter data

OneLine
MOUEN.L D8-D0/D6/D7/A0/A1/A6,-(SP)
MOVE #0,-(SP) ;horizontal coordinate
MOVE D3,-(SP) ;vertical coordinate
MoveTo ;set the pen
MOUEN.L (SP)+,D8-D0/D6/D7/A0/A1/A6

MOUEN.L D8-D0/D6/D7/A0/A1/A6,-(SP)
MOVE.L A6,-(SP) ;pointer to start of text
MOVE D5,-(SP) ;offset into block
MOVE #30,-(SP) ;number of bytes in line

(continued)
Listing 10.3 (continued)

`_DrawText` ;draw the line
MOVM.L (SP)+,D8-D4/D6/D7/R8/R1/R6
ADD D4,D3 ;increment vertical pointer
CMP PageBottom(A5),D3 ;at bottom of page?
BLT SamePage
JSR CloseAPage
JSR StartAPage ;begins new page in middle of character

SamePage
ADD #30,D5 ;offset to next line
MOVE D5,A3

BlankCheck
MOVE.B A6,A3,D2 ;get first character of next line
CMP.B # ’ ’ ,D2 ;is it a blank?
BNE CheckChar
ADDQ #1,A3 ;increment index to skip over blank
BRA BlankCheck

CheckChar
CMP.B D2,D6 ;has character changed?
BEQ OneLine ;no change
ADD D4,D3
ADD D4,D3 ;space between characters

Endings
ADDQ #1,D0 ;increment character counter
CMP PageBottom(A5),D3 ;at bottom of page?
BLT RoomLeft
JSR CloseAPage
BRA NewPage

RoomLeft
CMP D8,D7 ;all characters printed?
BNE Outerloop
MOVE.L PrPortPtr(A5),-<SP>
JSR PrClosePage

MOVE.L PrPortPtr(A5),-<SP>
JSR PrCloseDoc

MOVE.L PrRecHandle(A5),A0
MOVE.L (A0),A0
MOVE.B prJob+bDocLoop(A0),D0 ;draft or spooled?
BEQ DonePrinting

MOVE.L PrRecHandle(A5),-<SP> ;spooler uses its own printing port
CLR.L -<SP> ;spooler uses its own buffer
CLR.L -<SP> ;spooler uses its own device buffer
PEA PrStatusRec(A5)
JSR PrPicFile ;image and print spool file

(continued)
DonePrinting
    MOVE.L PrRecHandle(A5),A0
     _DisposHandle
    JSR PrClose
    BRA Update

;free space taken by print record
;close printing manager
;update window - it was covered by job
;dialog

StartRPage
    MOVEM.L D0/D4/D7/A0,-(SP)
    MOUE.L PrPortPtr(A5),-(SP)
    CLR.L -(SP)
     ;no scaling
    JSR prOpenPage
    MOVEM.L (SP)+,D0/D4/D7/A0

    MOVE D4,D3
     ;initialize vertical coordinate
    RTS

CloseRPage
    MOVEM.L D0/D4/D7/A0,-(SP)
    MOUE.L PrPortPtr(A5),-(SP)
    JSR PrClosePage
    MOVEM.L (SP)+,D0/D4/D7/A0
    RTS

;--------------------------------- Data Structures -----------------------
CharWindPtr DS.L 1
CharWindStrg DS WindowSize

EventRecord DS.B 16
WhichWindowPtr DS.L 1
DeskAccName DS 16
WindowFlag DS 1

AppleHandle DS.L 1
FileHandle DS.L 1
EditHandle DS.L 1
TextHandle DS.L 1
PrRecHandle DS.L 1
PrPortPtr DS.L 1
FontInfoStrg DS 4

PrStatusRec DS.B 3,3,47,287
PageBottom DS 1

ViewRect DC 3,3,47,287
DestRect DC 3,3,47,287

OrdinalList DC.B 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

JumpTable DC 0,360,720,1080,1440,1800,2160,2520,2880,3000,3360,3720,4080
DC 4560,4920,5280,5540,6000,6360,6720,7080,7440,7880,8280,8640,9000

(continued)
<table>
<thead>
<tr>
<th>Letters</th>
<th>DC.B</th>
</tr>
</thead>
</table>
|         | 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Note: This listing contains all strings of 30 characters in length.
(continued)
Listing 10.3 (continued)

DC.B "JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
Listing 10.3 (continued)

DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'
DC. B 'RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR'

DC. B 'SSSSS SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
Listing 10.3 (continued)

DC. B 'ZZZZZZZZ' ZZZZZZZZ'
DC. B 'ZZZZZZZZZZ' ZZZZZZZZ'
DC. B 'ZZZZZZZZZZZZ' ZZZZZZZZ'
DC. B 'ZZZZZZZZZZZZZZ' ZZZZZZZZ'
DC. B 'ZZZZZZZZZZZZZZZZZZZ ZZZZZZZZ'
DC. B 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'

Listing 10.4  Resource File for BannerPrint

BannerPrint.Rsrc

TYPE MENU
  .1
  \14

  .2
File
New/N
Close/W
Print/P
Quit/Q

  .3
Edit
Undo/Z
(-
Cut/X
Copy/C
Paste/V
Clear

TYPE WIND
  .1
Banner Text
50 110 100 400
Visible
GoAway
Ø
Ø
Questions and Problems

1. When doing draft printing, the Printing Manager will print only one copy of a document, regardless of the contents of the ICopies field in the print record. Write assembly language code to control the printing of multiple draft copies. For the actual printing details, include:

   JSR  PrintOneCopy

   in the body of your loop. Assume that print record has been allocated and has a handle stored as PrRecHandle in the applications globals area.

2. Write assembly language code to change the name of a spool file the Printing Manager will create from the default “Print File” to any other name of your choosing. Allocate any data structures your code will require. Assume the print record has been allocated and has a handle stored as PrRecHandle in the applications globals area.

3. Write pseudocode that describes the logic necessary to print a multi-page document. Indicate the details of drawing a single page by writing “Draw one page.” (The purpose of this problem is to summarize the sequence of Printing Manager calls.) Use the names of Printing Manager routines as appropriate.

4. If you look carefully at Figure 10.2 (spooled output), you will notice that the line under the column headings stretches across the entire page. This occurs because the entire line is printed as one string.

   A. Suggest a strategy that would restrict the underlining to the column headings themselves, as it appears in Figure 10.1 (draft output).
   B. What difficulty does this present for the programmer? Hint: think in terms of what information is required to properly space the heading.

5. Assume that an array of data is stored in the applications globals area. A pointer to the starting location of the array has the symbolic address BookStuff. Each record is 80 bytes long. The fields within the array are defined by the following equates:

<table>
<thead>
<tr>
<th>Field</th>
<th>EQU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>0</td>
</tr>
<tr>
<td>Author</td>
<td>30</td>
</tr>
<tr>
<td>Publisher</td>
<td>50</td>
</tr>
<tr>
<td>Date</td>
<td>75</td>
</tr>
</tbody>
</table>
Write the assembly language code that draws one record from this array onto a printer port. Assume:

a. The print line will hold 100 characters.
b. The number of the record being printed is in D0.
c. All necessary Printing Manager calls have been made.
d. The current vertical pen position is in D1.
e. The height of the print line is in D2.

Format the lines so there is space between the fields even when each field contains the maximum number of characters.

6. Expand the code from problem 5 to print an entire page. As well as adding a loop to print repeated records from the array, include:

a. A page number in the upper right-hand corner of the page (retrieve it from the printer record which is stored under PrRecHandle)
b. A page heading of your choice centered on the page two lines below the page number
   c. A heading above each column, three lines below the page heading

Assume that the page's bottom coordinate is stored in D3. All necessary printing manager calls have been made. The code need not check to see if all records in the array have been printed, but does need to check for the bottom of the page. Print the records single spaced.

7. Modify the code from problem 6 to set font characteristics as follows:

a. The page number should be boldface.
b. The page heading should be boldface and underlined.
c. Column headings should be standard print and underlined.
d. All headings should be printed using the system font (Chicago); the body of the page (the records themselves) should use the Geneva font.

8. Write a block of assembly language code that decides whether a user requested spooled or draft printing. Assume that the print record has a handle, PrRecHandle, stored in the applications globals area.

9. Expand the code from problem 8 to image and print the spool file.

10. As mentioned at the end of this chapter, the program BannerPrint is far from bullet-proof. As it appears in Listing 8.3, it has many holes into which a user could fall. Code and implement the following modifications to BannerPrint, each of which will isolate a user from his or her mistakes:

   A. Write an alert template to indicate that the user has entered something other than an upper-case letter or a space. Store the template in the
resource file (don't forget to recompile it with RMaker). Display the alert at the appropriate place in the program.

B. Write an alert template to get the user to turn on the printer. Store the template in the resource file. Display the alert at the appropriate place in the program. Printing should not begin until the user clicks OK to indicate that the printer is ready.

C. Add code to disable the New option from the File menu when a text edit window is created. Re-enable the New option when the window is closed. Disable the Close option when no window is present; enable it whenever a window is on the screen.

D. Add code to trap lower-case letters and transform them to upper-case before printing. In this case, the alert created in (a) should be displayed only if a character is not a letter or a space.

11. The appearance of the large letters produced by BannerPrint depend entirely on the type font and type size being used. The letters will be unrecognizable if the font is not mono-spaced (i.e., all characters are the same width), but that restriction nonetheless leaves a great deal of flexibility in font size and style. Code and implement the following enhancements to BannerPrint to give the user more choices:

A. Create a Size menu that will allow the user to select the size of the type to be used for printing. A template for the menu should be placed in the resource file.

B. Create a Style menu that will allow the user to select the style of type to be used for printing (e.g., plain text, boldface, outline, etc.).

NOTE: changes in size and style should also be reflected in the text displayed in the text edit window.
Chapter Objectives

1. To understand the difference between sequential and direct file access
2. To understand the data structures needed to process Macintosh files
3. To learn to create, open, close, read from, and write to both sequential and direct access files
4. To learn to use the Standard File Package to obtain file names and locations for opening and saving files

Introduction

Disk file manipulation is handled by the File Manager. While the routines and their parameter blocks may at first seem rather forbidding, the process is actually much easier than it looks.

When programming in Pascal, you must choose between setting up a file for direct access (with fixed field lengths) or for sequential access (with variable field lengths). As far as the Macintosh is concerned, however, there are only direct access files. That does not mean that a file cannot be processed in a sequential manner.

That last sentence is not double-talk. The terms direct access and sequential access really refer to how a file is processed, not to any physical characteristics of
the file. Direct access means that records are accessed in random order; that capability usually requires that the records are of a fixed length. Sequential access simply means that the records are accessed in order, starting at the beginning of the file. Assuming that one character (generally a carriage return) signifies the end of a record, files written for only sequential access can have variable record lengths. On the Macintosh it is also possible to move backwards when doing sequential processing. Note also that there is no reason that a file with fixed record lengths (i.e., a file that will permit direct access) cannot be processed in a sequential manner. For simplicity, this chapter will often speak of "direct access" and "sequential" files; such terminology applies only to the manner in which the data in the file are read and written.

The Macintosh keeps track of its current position within a file with a pointer called the mark. The mark is always positioned just beyond the last character read or the last character written. In other words, the mark points to the next byte to be read or written. Direct access is therefore achieved by specifying where file operations should occur with respect to either the current position of the mark or the start of the file. The mark is moved whenever read or write activities are performed. There is also a File Manager routine that will set the mark anywhere within a file.

Because file operations are rather slow compared to RAM-based operations, the Macintosh provides the option for an application to execute file operations asynchronously. Asynchronous file calls permit the program to continue with other tasks while the file operation is in progress. Synchronous file operations force the application to wait for the file operation to finish before proceeding. Synchronous execution is the default mode; asynchronous execution can be specified by setting bit 10 of the routine trap word. This chapter assumes that all file operations are to be executed synchronously.

Macintosh files have two parts, known as forks. The resource fork contains resource definitions and the code of application programs; the data fork is used for storing data. Data files created by applications will usually use only the data fork. In fact, although it is possible to open a file's resource fork from the File Manager, the commonly used routines are directed toward the data fork. The discussion that follows applies only to a file's data fork.

The resource and data forks maintain their own marks. They also maintain their own logical and physical end-of-file pointers. A logical end-of-file pointer is always positioned immediately after the last byte in the file. Since file space is allocated in 1024-byte blocks, the physical end-of-file pointer will be positioned just after the byte which ends the nearest block of 1024 bytes.

---

**Data Structures for File Operations**

There are three categories of file manipulation routines: I/O routines, file information routines, and volume information routines. (In this context the term
A volume usually refers to a single floppy disk; a volume may also be a partition on a hard drive.) Each group of routines has a lengthy parameter block. The address of the appropriate parameter block must be loaded into A0 before a File Manager routine is called.

The first eight parameters in the block are common to all File Manager routines:

\[
\text{ParamBlockRec} = \text{RECORD}
\]

\[
\begin{align*}
\text{qLink:} & \quad \text{QElemPtr; next element in file queue} \\
\text{qType:} & \quad \text{INTEGER; queue type} \\
\text{ioTrap:} & \quad \text{INTEGER; routine trap} \\
\text{ioCmdAddr:} & \quad \text{Ptr; routine address} \\
\text{ioCompletion:} & \quad \text{ProcPtr; completion procedure} \\
\text{ioResult:} & \quad \text{OSErr; result code} \\
\text{ioNamePtr:} & \quad \text{StringPtr; volume or file name} \\
\text{ioVRefNum:} & \quad \text{INTEGER; volume reference number or drive number}
\end{align*}
\]

\text{qLink} and \text{qType} refer to the system's file queue. Requests to the File Manager are queued, much like events are queued by the Event Manager. The File Manager, unless told otherwise, processes file activities in the order in which they were entered into the queue. \text{ioTrap} and \text{ioCmdAddr} relate to the particular routine being called. These four parameters are used solely by the File Manager and, for the most part, can be ignored.

\text{ioCompletion} is a pointer to a routine that should be initiated when an asynchronous file operation is completed. If there is no completion routine, \text{ioCompletion} should be set to 0. For synchronous calls, \text{ioCompletion} is automatically set to 0 and can therefore be ignored.

\text{ioResult} contains a File Manager result code (see Table 11.1). The result code also appears in D0 at the completion of all File Manager routines, which means that an assembly language application rarely needs to check this field of the parameter block.

Result codes do more than report errors; they provide important information about the condition of the file system. Consider, for example, the situation where an application needs to create a data file only if one doesn't already exist. The application can simply attempt to create the file. If a file by the same name already exists, the File Manager will return a result code of –48 ($FFFFFFD0). Therefore, a result code of 0 (no error) or –48 means that the application can proceed to open the file, since creating a file does not open it. Any other result code indicates that something unexpected has happened. The application can either interpret the code further (e.g., to determine if the disk is full) or abort the file request.

The contents of \text{ioNamePtr} depends on whether the routine being called is directed toward a single file or toward an entire volume. For routines that operate on volumes, it contains a pointer to the name of the volume. For file operations, the field contains a pointer to a file name. Note that a file name may be prefaced by a volume name. In that case, the volume name is separated from the file name by a colon. For example, \text{tape.index:Annotations} refers to the file called \text{Annotations} on a disk with the name \text{tape.index}.
Table 11.1 File Manager Result Codes

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Decimal Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>No error</td>
</tr>
<tr>
<td>FFFFFFFDF</td>
<td>-33</td>
<td>File directory is full</td>
</tr>
<tr>
<td>FFFFFFFDE</td>
<td>-34</td>
<td>Disk is full (no free 1024 byte allocation blocks)</td>
</tr>
<tr>
<td>FFFFFFFDD</td>
<td>-35</td>
<td>Requested volume is not on-line</td>
</tr>
<tr>
<td>FFFFFFFDC</td>
<td>-36</td>
<td>Unspecific disk I/O error</td>
</tr>
<tr>
<td>FFFFFFFDB</td>
<td>-37</td>
<td>File or volume name is bad</td>
</tr>
<tr>
<td>FFFFFFFDA</td>
<td>-38</td>
<td>File is not open</td>
</tr>
<tr>
<td>FFFFFFFD9</td>
<td>-39</td>
<td>End-of-file encountered when reading</td>
</tr>
<tr>
<td>FFFFFFFD8</td>
<td>-40</td>
<td>Application tried to put mark before start of file</td>
</tr>
<tr>
<td>FFFFFFFD7</td>
<td>-41</td>
<td>No space left in system heap</td>
</tr>
<tr>
<td>FFFFFFFD6</td>
<td>-42</td>
<td>Attempt to open more than 12 access paths</td>
</tr>
<tr>
<td>FFFFFFFD5</td>
<td>-43</td>
<td>File can't be located</td>
</tr>
<tr>
<td>FFFFFFFD4</td>
<td>-44</td>
<td>Volume is hardware locked</td>
</tr>
<tr>
<td>FFFFFFFD3</td>
<td>-45</td>
<td>File is software locked</td>
</tr>
<tr>
<td>FFFFFFFD2</td>
<td>-46</td>
<td>Volume is software locked</td>
</tr>
<tr>
<td>FFFFFFFD1</td>
<td>-47</td>
<td>Some files are open</td>
</tr>
<tr>
<td>FFFFFFFD0</td>
<td>-48</td>
<td>Duplicate file name</td>
</tr>
<tr>
<td>FFFFFFFD</td>
<td>-49</td>
<td>Attempt to open more than one access path/file for writing</td>
</tr>
<tr>
<td>FFFFFFFCE</td>
<td>-50</td>
<td>No volume specified and there is no default volume</td>
</tr>
<tr>
<td>FFFFFFFCD</td>
<td>-51</td>
<td>Access path number does not exist</td>
</tr>
<tr>
<td>FFFFFFFCB</td>
<td>-52</td>
<td>Volume does not exist</td>
</tr>
<tr>
<td>FFFFFFFCA</td>
<td>-53</td>
<td>Access path will not permit writing</td>
</tr>
<tr>
<td>FFFFFFFC9</td>
<td>-54</td>
<td>Attempt to mount an already mounted volume</td>
</tr>
<tr>
<td>FFFFFFFC8</td>
<td>-55</td>
<td>Drive number does not exist</td>
</tr>
<tr>
<td>FFFFFFFC7</td>
<td>-56</td>
<td>Not at Macintosh volume</td>
</tr>
<tr>
<td>FFFFFFFC6</td>
<td>-57</td>
<td>Illegal path reference number</td>
</tr>
<tr>
<td>FFFFFFFC5</td>
<td>-58</td>
<td>Unsuccessful attempt to rename a file</td>
</tr>
<tr>
<td>FFFFFFFC4</td>
<td>-59</td>
<td>Volume must be reinitialized because master directory block is bad</td>
</tr>
<tr>
<td>FFFFFFFC3</td>
<td>-60</td>
<td>Access path will not permit writing</td>
</tr>
<tr>
<td>FFFFFFFC2</td>
<td>-61</td>
<td>Access path will not permit writing</td>
</tr>
<tr>
<td>FFFFFFFC1</td>
<td>-62</td>
<td>Access path will not permit writing</td>
</tr>
</tbody>
</table>

**ioVRefNum** can contain either a reference number to a volume or the drive number that contains a particular volume. Generally, we use the drive number: 1 for the internal drive, 2 for the external drive.

**Specific Fields for I/O Routines**

Calls to I/O routines require nine fields in addition to the eight fields described above:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioRefNum</td>
<td>INTEGER;</td>
<td>path reference number</td>
</tr>
<tr>
<td>ioVersNum</td>
<td>SignedByte;</td>
<td>version number</td>
</tr>
<tr>
<td>ioPermssn</td>
<td>SignedByte;</td>
<td>read/write permission</td>
</tr>
<tr>
<td>ioMisc</td>
<td>Ptr;</td>
<td>depends on the routine</td>
</tr>
<tr>
<td>ioBuffer</td>
<td>Ptr;</td>
<td>pointer to data buffer</td>
</tr>
</tbody>
</table>
ioReqCount  LONGINT;  number of bytes to be transferred
ioActCount  LONGINT;  actual number of bytes transferred
ioPosMode:  INTEGER;  newline character and position of start
of operation relative to the mark
ioPosOffset:  LONGINT;  offset from mark or beginning of file

ioRefNum contains a file's path reference number. When a file is opened, the
system creates an access path to that file. An access path is a description of how
the system should get to a file. Twelve different access paths may be open at one
time. Any given file can therefore support up to 12 access paths, though only one
per file can be used for writing. Each access path has its own mark which moves
independently of the marks in any other access paths to that file. The path
reference number identifies which specific path should be used in a file operation.
The path reference number is returned when a file is opened and passed to
routines which read from and write to files.

ioVersNum was designed to allow the Finder to distinguish between two files
with the same name on the same disk. In practice, though, the Finder ignores this
parameter; the Resource Manager and Segment Loader won't work with files that
have non-zero version numbers. Therefore, the version number should always be 0.

The Macintosh allows an application to specify what kinds of operations are
permitted on a file; this is known as a file's read/write permission. It is stored in the
parameter block in ioPermssn. The possible values for ioPermssn are:

0:  the same as the access path's current permission
1:  read only permission
2:  write only permission
3:  read and write permission

By restricting file activity to reading, for example, an application can protect files
that should not be altered.

The contents of ioMisc varies with the specific File Manager routine.

ioBuffer is a pointer to an I/O buffer. An I/O buffer is an area in RAM from which
data is written to the disk or into which data is read. Its size depends on how much
data is to be transferred. An application does not necessarily need to set aside a
special I/O buffer. For example, since the Video Tape Index program keeps the
entire TapeArray in RAM in a single location while the application is running, that
block of storage can double as the buffer for accepting the information when it is
read from the disk at the beginning of program execution and when it is re-written
just before the program ends.

On the other hand, if data are scattered in RAM, then they must be assembled
into a single storage block before being written to disk. Reading that same data
back into RAM will deposit them into one contiguous block. In that case, an
application must allocate enough storage to hold all the data.
ioReqCount is the number of bytes that are to be transferred. That quantity is passed into read and write routines. ioActCount is returned by routines that transfer data; it contains the number of bytes actually read or written.

ioPosMode contains information to position the mark for data transfer operations and may also contain the ASCII code of a character that indicates the end of a record. The low-order byte of ioPosMode holds the position offset:

0: read and write operations should be at the current position of the mark (ioPosOffset is therefore ignored)
1: the offset contained in ioPosOffset is the offset, in bytes, from the beginning of the file
2: the offset contained in ioPosOffset is the offset, in bytes, from the end of the file
3: the offset contained in ioPosOffset is the offset, in bytes, from the current position of the mark.

The use of ioPosOffset depends on the value of ioPosMode. If ioPosMode is 0, the offset parameter is ignored. For an ioPosMode of 1, the offset will be added to the starting address of the file; the offset must be positive. When ioPosMode is either 2 or 3, the offset will be added to the end of the file or the mark, respectively; the offset may be either positive or negative.

Specific Fields for File Information Routines

The two routines that set and retrieve information about files require 16 parameters in addition to the first eight:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioFRefNum</td>
<td>INTEGER;</td>
<td>path reference number</td>
</tr>
<tr>
<td>ioFVersNum</td>
<td>SignedByte;</td>
<td>version number</td>
</tr>
<tr>
<td>filler1</td>
<td>SignedByte;</td>
<td>unused</td>
</tr>
<tr>
<td>ioFDrlndex</td>
<td>INTEGER;</td>
<td>file number</td>
</tr>
<tr>
<td>ioFlAttrib</td>
<td>SignedByte;</td>
<td>file attributes</td>
</tr>
<tr>
<td>ioFIVersNum</td>
<td>SignedByte;</td>
<td>version number</td>
</tr>
<tr>
<td>ioFIndrInfo</td>
<td>File;</td>
<td>a record including file type</td>
</tr>
<tr>
<td>ioFNum</td>
<td>LONGINT;</td>
<td>file number</td>
</tr>
<tr>
<td>ioFISblk</td>
<td>INTEGER;</td>
<td>first 1024 block of data fork</td>
</tr>
<tr>
<td>ioFILgLen</td>
<td>LONGINT;</td>
<td>logical EOF of data fork</td>
</tr>
<tr>
<td>ioFIPyLen</td>
<td>LONGINT;</td>
<td>physical EOF of data fork</td>
</tr>
<tr>
<td>ioFIRStBlk</td>
<td>INTEGER;</td>
<td>first 1024 block of resource fork</td>
</tr>
<tr>
<td>ioFIRLgLen</td>
<td>LONGINT;</td>
<td>logical EOF of resource fork</td>
</tr>
</tbody>
</table>
This type of parameter block is used primarily when creating a new file; it supplies information to the Finder about a new file. An application will rarely have to check any of its fields directly. Those parameters which must be passed as part of the file creation sequence are discussed later in this chapter.

### Specific Fields for Volume Information Routines

One File Manager routine, GetVolInfo, collects information about a specific disk volume. That routine requests its own parameter block, with 14 fields in addition to the eight common fields:

- **filler2**: LONGINT; unused
- **ioVolIndex**: INTEGER; volume index
- **ioVCrDate**: LONGINT; data & time of initialization
- **ioVLsBkUp**: LONGINT; data & time of last backup
- **ioVAtb**: INTEGER; bit 15 set if volume locked
- **ioVNmFls**: INTEGER; number of files on volume
- **ioVDirSt**: INTEGER; first block of file directory
- **ioVBILen**: INTEGER; number of blocks in directory
- **ioVAlBkSiz**: LONGINT; bytes/allocation block
- **ioVCIpSiz**: LONGINT; number of bytes to allocate
- **ioAIBIST**: INTEGER; first block in volume block map
- **ioVNextFNum**: LONGINT; next free file number
- **ioVFrBlk**: INTEGER; number of free allocation blocks

As with the file information parameters, most of the volume information parameters are used by the system and not directly by an application.

### Storage Space for Parameter Blocks

Storage space for File Manager parameter blocks should be allocated in the application globals area. For example, the following allocates an I/O parameter block:

```
ioParamBlock DS.B ioQEISize
```
ioQEISize is defined in the system equates file. Its value is the total number of bytes (50) in an I/O parameter block. Note that offsets for the fields within all three types of parameter blocks are also contained in the system equates file.

A file information parameter block might be defined as:

\[ \text{fiParamBlock DS.B ioQEISize} \]

and a volume information parameter block as:

\[ \text{vParamBlock DS.B ioVQEISize} \]

How many parameter blocks of each type do you need? That depends on two things: the number of access paths that will be open at any one time and how much parameter shuffling you want to do. Since the Video Tape Index program never has more than one access path open at any given time, only one parameter block of each type is required.

An application that simultaneously maintains more than one access path can handle the parameter block situation in one of two ways. The entire application can use a single set of parameter blocks if data returned by File Manager routines are removed from the parameter blocks and stored elsewhere before the blocks are used by a different access path. On the other hand, each access path can be allocated its own set of parameter blocks. In that case, data is left in the parameter block and doesn't need to be reloaded for subsequent calls to File Manager routines.

It may also be necessary to retrieve information from parameter blocks, even if only one access path is open at a time, if the application makes calls to the Printing Manager. Since the Printing Manager tends to disrupt data in the applications globals area, an application should be careful to at least store the access path reference number elsewhere.

Translating the Pascal syntax for File Manager routines into assembly language is not as straightforward as with the routines of other managers. All of the low-level File Manager routines which must be used from assembly language have the form:

\[ \text{FUNCTION ProcedureName (paramBlock: ParamBlkPtr; \newline async: BOOLEAN): OSErr;} \]

In practical terms, this means that for synchronous operations the address of the appropriate parameter block must be loaded into A0 just before the routine is called, and that OSErr (technically, an "operating system error" code, but in reality, one of the codes in Table 11.1) will be returned in D0. Therefore, the major portion of the setup for a File Manager routine involves loading the necessary information into a parameter block.
Creating a Data File

Creating a new file involves three steps:

1. Create the file (Create)
2. Assemble information about the file that the Finder already has (GetFileInfo)
3. Fill in the remaining information that the Finder needs (SetFileInfo)

The actual creation of a new file requires three pieces of information: the name to be given to the file, its location (usually given as a drive number), and the version number (should always be 0). These parameters must be moved into the I/O parameter block:

```
LEA 'Tape.Master',A0
MOVE.L A0,ioParamBlock + ioFileName(A5) ;address of file name
MOVE #1,ioParamBlock + ioDrvNum(A5) ;drive #
MOVE.B #0,ioParamBlock + ioFileType(A5) ;version #
```

The name selected for a file should be assembled with a length byte. If you wish to avoid worrying about the `STRING__FORMAT` assembler directive, allocate strings with `LEA` rather than defining them as constants. Remember that strings allocated with `LEA` and `PEA` are assembled with length bytes; those allocated with `DC` are assembled without length bytes.

The drive number parameter should be passed as 1 for the internal drive or 2 for the external drive. If you are working with a hard drive, consult the documentation that accompanied the drive to determine the drive's number.

File version number should always be 0.

Once the parameters are loaded into the parameter block, the starting address of the block is loaded into A0. Then the routine can be called:

```
LEA ioParamBlock(A5),A0
__Create
```

Whenever a file is successfully created, the Finder must be supplied with information about that file. Therefore, `Create` should be followed by calls to `GetFileInfo` and `SetFileInfo`.

Both routines use the file information parameter block. To call `GetFileInfo` you must supply:

1. The file's name (`ioFileName`)
2. The drive number where the file is located (`ioDrvNum`)
3. The file's version number (ioFileType)
4. The file's directory index (ioDirIndex)

ioDirIndex contains an integer that tells the File Manager what information should be used to locate the file. If its value is greater than 0, GetFileInfo will use that number to identify the file, assuming that it refers to the file's position in the disk directory. If its value is 0 or negative, GetFileInfo will use the file's name, drive number, and version number to locate the file. In either case, assuming that the requested file exists, the call to GetFileInfo will fill in the remainder of the fields in the file information parameter block with the correct data about that file.

SetFileInfo assures that the Finder has correct information about a given file. It requires the following information:

1. The file's name (ioFileName)
2. The drive number where the file is located (ioDrvNum)
3. The file's version number (ioFileType)
4. The file's type (ioFndrInfo)
5. The file's time and date of creation (ioFICrDat)
6. The file's time and date of last modification (ioFIMdDat)

The last two items listed above are supplied by the call to GetFileInfo. The first three are loaded into the parameter block when setting up for the call to GetFileInfo. The setup for SetFileInfo therefore involves loading the file type and reloading the address of the parameter block into A0.

A file's type is a four-character string. Files with type 'TEXT' can be read by Macintosh text processing programs such as MacWrite and Microsoft Word. Data files created by an application should therefore be type 'TEXT' unless there is some specific reason for preventing the user from viewing and possibly modifying the files.

The Video Tape Index uses the following code to set up and call GetFileInfo and SetFileInfo after creating a new Tape.Master file (Tape.Master holds the information from TapeArray):

```
LEA 'Tape.Master',A0
MOVE.L A0,fiParamBlock + ioFileName(A5) ;file name
MOVE #1,fiParamBlock + ioDrvnum(A5) ;drive number
MOVE.B #0,fiParamBlock + ioFileType(A5) ;version number
MOVE #0,fiParamBlock + ioFDirlndex(A5) ;use name & version number to find file
LEA fiParamBlock(A5),A0
__GetFileInfo
MOVE.L $'TEXT',fiParamBlock + ioFlUsrWds ;start of ioFndrInfo record
```
Opening a File

Before a file can be read from or written to, it must be opened. Creating a file merely creates a disk directory entry that will produce a document icon when the disk's contents are viewed from the Finder; creating does not open a file. The File Manager routine Open will open a file by creating an access path to that file. It returns a reference number to the access path.

The information required by Open is:

1. the name of the file
2. the drive number
3. the file's version number
4. the permission code for this access path (Remember that only one access path to any file can allow writing.)
5. in ioMisc, a pointer to an access path buffer

An access path buffer is a block of RAM that is used as temporary storage by the access path. Either allocate the access path buffer explicitly, in which case it should be defined as 522 bytes long, or instruct the system to use the volume's buffer by passing a value of 0. It is important that all access paths to one file share the same buffer, regardless of whether it is an application-defined buffer or the volume's buffer.

The Video Tape Index uses the following code to open the Annotations file:

```
LEA fiParamBlock(A5),A0
__SetFileInfo

Opening a File

Before a file can be read from or written to, it must be opened. Creating a file merely creates a disk directory entry that will produce a document icon when the disk's contents are viewed from the Finder; creating does not open a file. The File Manager routine Open will open a file by creating an access path to that file. It returns a reference number to the access path.

The information required by Open is:

1. the name of the file
2. the drive number
3. the file's version number
4. the permission code for this access path (Remember that only one access path to any file can allow writing.)
5. in ioMisc, a pointer to an access path buffer

An access path buffer is a block of RAM that is used as temporary storage by the access path. Either allocate the access path buffer explicitly, in which case it should be defined as 522 bytes long, or instruct the system to use the volume's buffer by passing a value of 0. It is important that all access paths to one file share the same buffer, regardless of whether it is an application-defined buffer or the volume's buffer.

The Video Tape Index uses the following code to open the Annotations file:

```
LEA 'Annotations',A0
MOVE.L A0,ioParamBlock + ioFileName(A5) ;file name
MOVE #1,ioParamBlock + iodrvnum(A5) ;drive #
MOVE.B #0,ioParamBlock + ioFileType(A5) ;version #
MOVE.B #3,ioParamBlock + ioPermssn(A5) ;read & write permission
CLR.L ioParamBlock + ioOwnBuf(A5) ;use volume buffer
LEA ioParamBlock(A5),A0 __Open

CMP #0,D0 ;any errors?
BNE FileError ;handle error
LEA fiRefNum,A0 ;save access path
MOVE ioParamBlock + ioRefNum(A5),(A0) ;reference number
```
The final step in the sequence above explicitly retrieves the access path reference number from the parameter block and stores it elsewhere. This is necessary because calls to the Printing Manager disrupt the values in the parameter block and their integrity cannot be ensured. The access path reference number is therefore always reloaded into the parameter block before any further operations are performed on that file.

---

**Writing to Disk Files**

A single File Manager routine, **Write**, performs both sequential and direct access write operations. The difference between the two types of processing is in how an application specifies where writing should begin.

**Writing a Sequential File**

The Video Tape Index stores the data from the RAM-based TapeArray in a sequential file (Tape.Master). This file is read into RAM when the program is launched and rewritten to disk when the user selects Quit from the Options menu. Since the format of the file is exactly the same as the format of TapeArray, the block of storage occupied by TapeArray can be used as the I/O buffer for both read and write operations.

TapeArray has fixed field, and therefore fixed record, lengths. That characteristic makes it easy to do direct access operations on the array while it is in RAM. Nonetheless, do not assume that all sequential files need to have fixed record lengths; on the contrary, they do not. Generally, a carriage return is used to mark the end of a record in a sequential file. The carriage return is not automatically inserted by the system; it must be written explicitly as the last character of each record.

Data are written to disk in 512-byte blocks. If a write operation ends up with a final segment of less than 512 bytes, that data is stored temporarily in the access path buffer until either another write request brings the total number of bytes to 512, or until the application calls a routine that flushes the buffer. **FlushFile** will explicitly write all contents of an access path buffer to disk (requires only the access path reference number in the I/O parameter block). **Close** also flushes the access path buffer. (See the section later in this chapter on closing files.)

**Write** requires the following parameters in the I/O parameter block:

1. access path reference number (**ioRefNum**)
2. starting address of the I/O buffer (**ioBuffer**)
3. number of bytes that should be transferred (**ioReqCount**)
4. the position mode (**PosMode**)
5. the offset (ioPosOffset)

Write returns another result in addition to the error code in D0. ioActCount will contain the actual number of bytes that were written.

To write the entire Tape.Master file, the Video Tape Index uses the following code:

MOVE    fiRefNum,ioParamBlock + ioRefNum(A5) ;access path
LEA     TapeArray(A5),A0
MOVE.L  A0,ioParamBlock + ioBuffer(A5)      ;I/O buffer
MOVE    TotalRecords,D0
MULU    #64,D0                                ;total bytes to move
MOVE.L  D0,ioParamBlock + ioByteCount(A5)
MOVE    #0,ioParamBlock + ioPosMode(A5)       ;(see below)
LEA     ioParamBlock(A5),A0
_Write

The ioPosMode necessary for a sequential write has a code of 0. That indicates that the write operation should begin at the current position of the mark. As mentioned earlier in this chapter, when ioPosMode is zero, the offset parameter is ignored.

To append to a sequential file, use an ioPosMode of 2 and an ioPosOffset of 0. The write will then begin exactly at the logical end-of-file.

If you look at the listing of the Video Tape Index program, you will notice that there is a write operation that occurs before TapeArray is written. Like many sequential files, Tape.Master stores "housekeeping" information in its first record; the term "housekeeping" usually refers to data needed to process the rest of the file, or constants that must be retained from one program run to another. The first four bytes of Tape.Master contain two integers — the total number of records in the file and the last annotation number used. Those bytes are therefore processed separately:

MOVE    TotalRecords,D0 ;retrieve total records
SWAP    D0              ;(see below)
AND.L   #$FFFF0000,D0  ;clear low order byte
MOVE    LastAnnotNumb,D0 ;retrieve last annot. #
MOVE.L  D0,DataBuffer(A5)
MOVE    fiRefNum,ioParamBlock + ioRefNum(A5) ;access path reference #
LEA     DataBuffer(A5),A0
MOVE.L  A0,ioParamBlock + ioBuffer(a5)        ;I/O buffer
MOVE.L  #4,ioParamBlock + ioByteCount(A5)     ;write only four bytes
MOVE #0,ioParamBlock + ioPosMode(A5) ;write at mark
LEA ioParamBlock(A5),A0
__Write

The five first steps in this block of code prepare the total number of records and the last annotation number for writing. When the total number of records is moved into D0, it is stored in the low-order word of the register, since the MOVE was specified as a word-length operation. The next instruction, SWAP inverts the position of the high- and low-order words of registers. In the example above, it puts the total number of records into the high-order word. ANDing D0 with the mask of $FFFFFF0000 preserves whatever is stored in the high-order word (the total number of records) and clears out the low-order word. The second MOVE, since it is also a word-sized instruction, puts the last annotation number in the low-order word of D0 without disturbing the total number of records in the high-order word. Finally, the contents of D0 are transferred to the first four bytes of the storage location set aside as an I/O buffer. (This buffer is 256 bytes long — just enough space for an annotation.)

Writing to a Direct Access File

The only difference between writing to a direct access file and to a sequential file lies in ioPosMode. Generally, a direct access operation occurs with a byte offset relative to the beginning of the file (ioPosMode = 1) or relative to the current position of the mark (ioPosMode = 3).

The Video Tape Index program stores annotations for the tapes in a direct access file. The last field in the TapeArray records is an integer that corresponds to the record number of each tape's annotation. Annotation record numbers are assigned sequentially as new tapes are entered. In other words, the 15th tape entered will have an annotation number of 15, regardless of where the tape's title falls in the alphabetical sequence of tapes. Therefore, an annotation number will generally not correspond to a tape's record number in TapeArray.

The code to write an annotation appears in Listing 11.1. The first step (a) fills the 256-byte I/O buffer with blanks. Though there is nothing that says any given annotation must use all 256 bytes allocated for it, it is essential that the space between the last character of the annotation and the end of the record is padded with blanks. If it is not, and only the exact number of characters in the annotation are written to the file, a subsequent read will transfer garbage characters at the end of the record as well as the annotation itself.

After filling the I/O buffer with blanks, the annotation is moved to the buffer. TEGetText provides a handle to the text (b) and BlockMove transfers the characters (c). The next step is to figure out exactly where this annotation should be placed in the file. This requires the annotation number, which is stored as the last field in the matching TapeArray record.
Listing 11.1 Writing an Annotation to its Direct Access File

LEA AnnotRecMask,A0
LEA DataBuffer(A5),A1
MOVE #256,D0
(a) _BlockMove ;fill buffer with blanks
CLR.L -(SP) ;place for CharsHandle result
MOVE.L AnnotationTextHandle, -(SP)
(b) _GetText ;get handle to text in Annotation record
MOVE.L (SP)+,A2 ;recover CharsHandle
MOVE.L (A2),A0 ;de-referencing handle to get pointer
LEA DataBuffer(A5),A1 ;text goes into disk buffer
MOVE.L AnnotationTextHandle,A3
MOVE.L (A3),A4 ;de-reference again
MOVE teLength(A4),D0 ;number of characters to move
(c) _BlockMove ;puts annotation in disk output buffer
LEA RecordCounter,A0
MOVE (A0),D5
(e) MULU #64,D5
ADD #oAnnotNum,D5 ;offset into tape array
LEA TapeArray(A5),A0
ADD.L D5,A0
(g) MOVE (A0),D0 ;offset into file
MULU #256,D0
LEA DataBuffer(A5),A0
MOVE.L A0,ioParamBlock+ioBuffer(A5)
MOVE.L #256,ioParamBlock+ioByteCount(A5) ;write 256 bytes, blanks and all
(i) MOVE #1,ioParamBlock+ioPosMode(A5) ;offset is relative to beginning of file
MOVE.L D0,ioParamBlock+ioPosOffset(A5) ;offset in bytes
(j) MOVE fiRefNum,ioParamBlock+ioRefNum(A5) ;file reference number
LEA ioParamBlock(A5),A0
_Write

To locate the annotation number, the program gets the TapeArray record number (d) and uses it to compute first an offset into the array (e) and then an offset into the record (f). That address is used to retrieve the annotation number (g). The annotation number is multiplied by 256 (h), the number of characters in each annotation, to produce a byte offset from the beginning of the file.

The setup for the call to Write is exactly the same as that used for a sequential write with two exceptions. ioPosMode is set to 1 to indicate that the offset is relative to the beginning of the file (i) and ioPosOffset (j) is loaded with the computed offset.

Reading From Disk Files

Reading from a file is exactly the same as writing to a file except that the data transfer is in the opposite direction. In a read operation, the I/O buffer is the location
into which data is read. As with writing, the buffer can be a storage location specifically set aside for I/O, or a storage location used for another purpose as well.

The **Read** routine requires precisely the same parameters as a **Write**:

1. the access path reference number (**ioRefNum**)
2. a pointer to the I/O buffer (**ioBuffer**)
3. the number of bytes that should be transferred (**ioByteCount**)
4. the position mode (**ioPosMode**)
5. the offset (**ioPosOffset**)

**Read** returns two results in addition to the error code in D0. **ioActCount** will contain the actual number of bytes that were transferred. **ioPosOffset** will contain the position of the mark after the read is completed.

All read operations transfer data in blocks of 512 bytes. If a read request involves less than 512 bytes, a full 512 bytes will be brought into RAM (into the access path buffer), but the application will receive only those bytes that were specified in the I/O parameter block.

### Reading From Sequential Files

A sequential read is simply a read that begins at the current position of the mark and reads forward. As an example, consider the procedure used by the Video Tape Index to load TapeArray at the beginning of every program run. Assume that the file has just been opened and that the access path reference number is therefore already in the parameter block (since the Tape.Master file is closed after its contents are read into RAM, there is no need to worry about the parameter block being disturbed by the Printing Manager).

First, the housekeeping information is retrieved:

```assembly
LEA    DataBuffer(A5),A0 ;place to receive data
MOVE.L A0,ioParamBlock + ioBuffer(A5)
MOVE   #4,ioParamBlock + ioByteCount(A5) ;read just first 4 bytes
MOVE   #0,ioParamBlock + ioPosMode(A5) ;read from mark
LEA    ioParamBlock(A5),A0
__Read
MOVE.L DataBuffer(A5),D0 ;get 4 bytes just read
LEA    LastAnnotNumb,A0
MOVE   D0,(A0) ;store last annot. #
SWAP   D0
LEA    Total Records,A0
MOVE   D0,(A0) ;store total records
```
After the above read operation, the mark will be positioned at the fifth byte in the file, the starting location of the first record of TapeArray. The program can then load all of TapeArray with one call to **Read**:

```plaintext
LEA TapeArray(A5),A0 ;i/O buffer
MOVE.L A0,ioParamBlock + ioBuffer(A5)
MOVE TotalRecords, D0
MULU #64,D0 ;number of bytes to read
MOVE.L D0,ioParamBlock + ioByteCount(A5)
MOVE #0,ioParamBlock + ioPosMode(A5) ;read from mark
LEA ioParamBlock(A5),A0
___Read
```

## Reading From Direct Access Files

Reading from a direct access file needs an **ioPosMode** of 1, 2, or 3 and an appropriate value for **ioPosOffset**. In all other respects, the process is identical to doing a sequential read. Because the annotations are so long (up to 256 characters), the Video Tape Index program leaves them on disk and reads a single annotation into RAM when needed. The procedure to read a single annotation record is:

```plaintext
LEA DataBuffer(A5),A0 ;place to receive data
MOVE.L A0,ioParamBlock + ioBuffer(A5)
MOVE.L #256,ioParamBlock + ioByteCount(A5)
MOVE #1,ioParamBlock + ioPosMode(A5);relative to beginning of file
MOVE RecordCounter,D5 ;current record #
MULU #64,D5
ADD #0AnnotNum,D5 ;offset into TapeArray
LEA TapeArray(A5),A0
ADD D5,A0 ;address of annot #
MOVE (A0),D0 ;retrieve annot. #
MULU #256,D0 ;offset into file
MOVE.L D0,ioParamBlock + ioPosOffset(A5)
LEA ioParamBlock(A5),A0
___Read
```
Closing a File

An application should explicitly close all files with Close before returning to the Finder. Though files will be closed automatically whenever the system is rebooted, the Close routine flushes the access path buffer, completing any write operations that were temporarily held because they involved less than 512 bytes. Close also deletes the access path. Files that are not closed cannot be deleted by the Finder.

Close requires only one parameter — the access path reference number:

```
MOVE fiRefNum, ioParamBlock + ioRefNum(A5)
LEA ioParamBlock(A5), A0
__Close
```

Timing Out for File I/O

Next to printing, disk I/O is the slowest part of an application. Often the user will have to wait more than a few seconds for some file operation to be completed. For example, as the number of records in Tape. Master grows, the time needed to read and write the file will continue to increase. Applications that adhere to the Macintosh user interface should change the shape of the cursor to the wrist watch (indicating a long wait) for any time-consuming operation.

The shape of the cursor is controlled by the QuickDraw routine SetCursor:

```
PROCEDURE SetCursor (crsr: Cursor);
```

This routine's single parameter is actually a pointer to the location of a resource. The resource is a cursor definition. Four cursors are defined in the system resource file: an I-Beam (resource ID = 1), a cross (resource ID = 2), a plus sign that looks like an outlined cross (resource ID = 3), and the wristwatch (resource ID = 4). A handle to the resource definition is returned by a routine from the ToolBox utilities — GetCursor:

```
FUNCTION GetCursor (cursorID: INTEGER); CursHandle;
```

cursorID refers to the resource ID of one particular cursor.

The following code will set the cursor to the wrist watch:

```
CLR.L - (SP) ;space for cursor handle result
MOVE #4, - (SP) ;resource ID for wristwatch
__GetCursor
```
MOVE.L (SP) + A0 ; retrieve cursor handle
MOVE.L (A0), A0 ; de-reference to get pointer
MOVE.L A0, - (SP) ; put pointer on stack
__SetCursor ; change arrow to wristwatch

The cursor can be returned to the arrow cursor with a call to InitCursor.

Managing Disk Changes and Choosing File Names — the Standard File Package

Many Macintosh applications will have a standard File menu with options that allow a user to open, close and save files. Opening and saving files should allow the user to enter a file name and to change disks and drives if necessary. The application should also take care of initializing disks if an uninitialized disk is inserted. The Standard File Package (package #3) provides routines that collect information from predefined dialog boxes and take care of initializing disks.

The two routines that most programmers will use are SFGetFile (routine #2 in the package) and SFPutFile (routine #1). SFGetFile is used to open files and SFPutFile to save files. Both routines return information to the application in a standard file reply record:

\[
\text{SFReply} = \text{RECORD} \\
\text{good: BOOLEAN; set FALSE if user cancels} \\
\text{copy: BOOLEAN; unused} \\
fType: \text{OSType;} \text{ file type or unused} \\
vRefNum: \text{INTEGER;} \text{ drive number} \\
version: \text{INTEGER;} \text{ file version number} \\
fName: \text{STRING[63]; file name}
\]

The first five fields occupy 10 bytes. Therefore, an application should allocate a total of 74 bytes of space for the file reply record. Offsets for the fields in the reply record are assigned symbolic addresses in the Package equates file, which should be INCLUDEd in the application's source code.

Selecting a File to Open

The dialog box displayed by SFGetFile appears in Figure 11.1. It allows the user to change disks and the default drive, lists all files on the default drive that can be opened, and accepts a command to either open a file or cancel the request.
This is a stack-based routine:

PROCEDURE SFGetFile (where: Point; prompt: Str 255; fileFilter: ProcPtr; numTypes: INTEGER; typelist: SFTypelist; dlgHook: ProcPtr; VAR reply: SFReply);

The parameter where contains, in global coordinates, the location of the upper left-hand corner of the dialog box. The prompt is ignored.

The next three parameters specify what types of files should be presented to the user as candidates for opening. numTypes contains an integer indicating the number of types of files that should be selected. The maximum value is four; a value of -1 will select all files on the default volume. The actual types to be selected are loaded into typelist. typelist is large enough to hold 16 characters (it is a packed array, 16 bytes long). Since it is larger than a longword, a pointer to typelist is pushed onto the stack. fileFilter is a pointer to a function that can perform additional file filtering. For example, files could be filtered by last date of modification. In most cases though, no additional filtering is necessary and fileFilter is set to 0.

dlgHook is a pointer that allows an application to display a dialog box other than the standard seen in Figure 11.1. Normally, it will be set to 0. The final parameter, reply, is a pointer to the reply record.

Figure 11.1 "Get File" Dialog Box from the Standard File Package
SFGetFile requires two data structures, one for the reply record and one for the file type list:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplyRecord</td>
<td>DS.B</td>
<td>74</td>
</tr>
<tr>
<td>TypeList</td>
<td>DS.B</td>
<td>16</td>
</tr>
</tbody>
</table>

The code below will select all TEXT files:

```assembly
MOVE.L # 'TEXT', TypeList(A5) ; load one file type
MOVE #50, -(SP) ; top coordinate
MOVE #50, -(SP) ; left coordinate
CLR.L -(SP) ; place for unused prompt
CLR.L -(SP) ; no filter procedure
MOVE #1, -(SP) ; one file type
PEA TypeList(A5) ; address of type list
CLR.L -(SP) ; use standard dialog
PEA ReplyRecord(A5) ; address of reply record
MOVE #SFGetFile, -(SP) ; routine #
__Pack3 ; invoke the package
```

SFGetFile monitors events and automatically takes care of ejecting disks and changing drives when the user clicks the appropriate button. If an uninitialized disk is inserted, the Standard File Package calls the Disk Initialization Package and handles the entire initialization process. The dialog box is closed when the user chooses Cancel or when a file is selected. A file can be selected by one click on the file name and a second click in the Open button, or by a double-click on the file name.

Once the dialog box has been removed, it is up to the application to retrieve information from the reply record and continue with the file operation. The first action is generally to check the good field to determine whether the file request has been canceled.

### Naming a File

SFPutFile provides a standard dialog box (Figure 11.2) that permits a user to name a file as well as to eject a disk and change the default drive:

```assembly
PROCEDURE SFPutFile (where: Point; prompt: Str 255; origName: Str255; dlgHook: ProcPtr; VAR reply: SFReply);
```

Most of the parameters are the same as those for SFGetFile. In this case, though, prompt has meaning; it is displayed above the window where the file name is entered and usually has a value like ‘Save current file as’. origName determines what will be displayed within the file name window when the dialog box first appears. If the current file has a name that should be assigned to origName;
otherwise, `origName` should be set to the null string. To indicate the null string as a literal, type two single quotes or two double quotes right next to each other.

![Figure 11.2 "Put File" Dialog from the Standard File Package](image)

If we assume that a user has selected Save As from a File menu to save a new text file, the assembly language code would appear as:

```
MOVE   #50, - (SP)            ; top coordinate
MOVE   #50, - (SP)            ; left coordinate
PEA    'Save current file as' ; prompt
PEA    ' '                    ; file name (null string)
CLR.L   - (SP)                ; use standard dialog
PEA    ReplyRecord            ; address of reply record
MOVE   #SFPutFile, - (SP)     ; routine #
_Pack3
```

Pointers to the text of the prompt and the file name can be pushed as literals (as above) or by symbolic addresses. Since their data type is Str255, they must have a length byte. Pushing them as literals will automatically ensure that the length byte is present.

`SFPutFile` will continue to monitor events until the user either:

1. clicks the Cancel button
2. types a file name and clicks the Save button or
3. types a file name and hits the Enter or Return key.

If the Cancel button has been clicked, control returns immediately to the application.

In either of the latter two applications, `SFPutFile` verifies the file name before returning. If the file name already exists, `SFPutFile` displays the alert that asks whether the existing file should be overwritten. If the user clicks the Yes button, control returns to the application with a value of TRUE in `good`. If the user clicks the No button, `good` is set FALSE.

`SFPutFile` also checks to see if a disk is locked, either by hardware or software. If a disk is locked, an alert box informs the user of the situation and cancels the file operation. `good` receives a value of FALSE.

When control is returned to the application, the program must then retrieve the appropriate information (usually the file name and drive number) from the reply record and proceed to save the file with a write operation as described earlier in this chapter.

---

Questions and Problems

1. Assume that an I/O parameter block has been allocated with the statement:

   ```assembly
   ioParamBlock DS ioQEISize
   ```

   where `ioQEISize` is defined in the system equates `file` as equal to the number of bytes in an I/O parameter block. Using the offsets into the parameter block defined in the system equates `file`, write assembly language code to load the following data:

   A. a path reference number that has been stored in the applications globals area under the symbolic address of `PathRefNum`
   B. a version number of 0
   C. an I/O buffer in the applications globals area identified by the symbolic address `MyBuffer`
   D. write only permission
   E. the appropriate values for `ioPosMode` and `ioPosOffset` so that new records will be appended to the end of the file.
2. A. Write assembly language code to create a file named **TextFile.txt** on the internal drive. Remember to collect all the information needed by the Finder as well as simply creating the file. Allocate any data structures your code will use.

B. At the end of your block of code, is **TestFile.txt** ready for read and write operations? Why or why not?

Problems 3 - 8 refer to the **BookStuff** array first introduced in problem 5. The structure of the array is defined as:

<table>
<thead>
<tr>
<th>Field</th>
<th>EQU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>0</td>
</tr>
<tr>
<td>Author</td>
<td>30</td>
</tr>
<tr>
<td>Publisher</td>
<td>50</td>
</tr>
<tr>
<td>Date</td>
<td>75</td>
</tr>
</tbody>
</table>

The total length of a record is 80 bytes. The file **BookStuff.data** holds the same data as the array in RAM.

3. Write assembly language code to open **BookStuff.data** for input and output on the external drive. Be sure to allocate required data structures. Use the RAM array as an I/O buffer.

4. Write assembly language code to write the entire **BookStuff** array sequentially to **BookStuff.data**. Assume that the total number of records in the array is stored in D0. Assume also that the file has just been opened by the code you wrote for problem 3.

5. Write a block of code to perform a direct access write for one record from **BookStuff** to **BookStuff.data**. The record number is stored in D0. Assume the file has been opened by the code you wrote for problem 3.

6. Write a block of code to read the entire **BookStuff.data** file into the **BookStuff** array in main memory. The total number of records in the file is stored in D0. Assume the file has just been opened by the code you wrote for problem 3.

7. Write a block of code to close **BookStuff.data**.

8. Assume now that **BookStuff.data** has been opened with write-only permission and that the data for a single record has been stored in the applications globals area with a symbolic address of **OneRecord**. Write a block of code to **append** the new record to **BookStuff.data**.
9. Assume that a standard file reply record has been defined in the applications globals area with a symbolic address of ReplyRecord. Write a block of code that will display the standard "get file" dialog box and then retrieve the name and drive number of the file the user selects. Anchor the top left corner of the dialog box at 100,80. Display the names of all files of type TEXT and MACA (MacWrite version 4.5) in the dialog box. Allocate any other data structures your code will use.

10. Assume that a standard file reply record has been defined in the applications globals area with a symbolic address of ReplyRecord. Write a block of code that will display the standard "save as" dialog box and then retrieve the file name and drive number from the reply record. The top left corner of the dialog box is at 75, 100. Select an appropriate prompt for the dialog box. Allocate any necessary data structures.
Chapter Objectives

1. To understand the problems associated with numeric I/O
2. To understand the Macintosh's floating point formats
3. To learn to do binary/decimal conversions for integers and floating point numbers
4. To learn to use the Macintosh's arithemetic packages to perform advanced mathematical operations
5. To learn to use separately assembled subroutines
6. To learn to create macros to simplify program code

Introduction

While microprocessor instruction sets contain instructions that perform integer arithmetic, they make no provision for arithmetic with numbers that contain fractional portions. Integer arithmetic is also limited to quantities that will fit into a single register (32 bits). As well as manipulating fractions and very large and very small numbers, it would also be useful to have routines that evaluate logarithmic
and trigonometric functions. Most microcomputers, therefore, have software that provides for a variety of advanced mathematic operations. Because the numbers processed by these routines have decimal points that move, they are referred to as floating point numbers and operations on them as floating point arithmetic. The Macintosh has a powerful floating point arithmetic package called FP68K. Trigonometric, exponential and logarithmic functions are provided by the elementary functions package, ELEMS68K.

The format of floating point numbers closely resembles scientific notation, where a mantissa is multiplied by 10 raised to a power (the exponent). For example, \(3.98 \times 10^{-15}\) is a very small number (.00000000000000398). The mantissa is 3.98; the exponent is \(-15\). The exact format in which floating point numbers are stored by computers differs from machine to machine. The Macintosh format is described below.

Arithmetic, whether it be floating point or integer, presents a significant I/O problem. All input from the keyboard is in ASCII; numbers enter the system as a string of ASCII character codes. That means that the ASCII codes must be converted from a string of decimal characters into a binary quantity before any math can be done. Integer conversion is handled by the Binary-Decimal Conversion Package. Floating point conversion is a two-step process; the ASCII character string must first be put into an intermediate format (the canonical decimal format) which is then used by the decimal-to-binary conversion routines. The Pascal implementation of FP68K provides routines that will convert directly from an ASCII string to binary and back again. Unfortunately, those routines are not available from assembly language. An application must therefore provide the code that puts the ASCII string into the intermediate format. After discussing the integer conversion routines, this chapter will look in detail at a subroutine that will properly reformat strings of decimal characters.

---

**The Binary-Decimal Conversion Package**

The Binary-Decimal Conversion Package contains only two routines: one to translate a string of ASCII decimal characters into a binary integer and another to take a binary integer and convert it to a string.

**NumToString** is the routine that converts a longinteger into a string of characters:

```pascal
PROCEDURE NumToString (theNum: LONGINT; VAR theString: Str255);
```
Since the data type of the string produced by this routine is **Str255**, it will have a length byte.

Before calling this routine, a pointer to a storage location for the string is placed in A0. The number that is to be converted is loaded into D0. **NumTostring** will place the string at the location specified by the address in A0. The string can then be displayed with **Drawstring**, for example, or incorporated into a text edit record.

The code below will convert an integer to a string:

```
MOVE.L #134599,D0
LEA StringStorage(A5),A0
MOVE #0, - (SP) ;select the NumTostring routine
    __Pack7 ;invoke the package
StringStorage     DS.B 20
```

**StringToNum** is the exact opposite of **NumTostring**; it converts a string with the data type Str255 into a longinteger:

```
PROCEDURE StringToNum (theString: Str255; VAR theNum: LONGINT);

A pointer to the string to be converted is loaded into A0. The number will be returned in D0. The system determines the number of characters in the string by examining its length byte:

```
LEA # '123456',A0
MOVE #1, - (SP) ;select the StringToNum routine
    __Pack7 ;invoke the package
```

**StringToNum** does not check to be sure that all characters in the string are digits. The routine is based on the fact that the ASCII codes for the digits are $30$ through $39$. If it looks just at the four low-order bits of the ASCII code, it has the quantity for the digit. This procedure, assuming that the four low-order bits contain the quantity, can be applied to any other character as well as a digit without creating a system error. Therefore, any character checking must be performed by the application.
Floating Point Decimal-to-Binary Conversions

The FP68K decimal-to-binary conversion routines work from a canonical decimal format that is defined as:

```
TYPE
    SigDig = String[20]
    Decimal = record
        sgn: 0..1;
        exp: INTEGER;
        sig: SigDig
    end;
```

The numbers described by the decimal record are stored as a string of up to 20 significant digits that are multiplied by 10 raised to some power (the exponent). For example, in the number $34567 \times 10^{-6}$, $34567$ are the significant digits and $-6$ is the exponent. The decimal point is always directly to the right of the most significant digit: that is, the mantissa is always presented as if it were an integer. The decimal point is not stored in the decimal record but its presence is inferred. The exponent is stored as an integer; the significant digits are a string of ASCII characters preceded by a length byte (their data type is `Str20`). The sign (`sgn`) is stored in bit 8 of the sign word. A value of 0 indicates a positive number; a value of 256 (a 1 in bit 8) indicates a negative number.

There is one major problem with Macintosh's canonical decimal format — user's don't enter data that way. An application must therefore have some way to convert strings of digits with embedded decimal points into that format. Listing 12.1 is an example of a subroutine that will "parse" (break down into constituent parts) character strings and reformat them into the canonical decimal format.

Listing 12.1 Parsing Numeric Strings

```
;---------------------------------------------- Simple Parser ----------------------------------------------
;
; Register Usage
;   A0   starting address of numeric string (load before calling routine)
;   A2   starting address of decimal record (result will go here)
;   D0   starting character position in string
;   D5   exponent
;   D6   number of significant digits
;   D7   total length of string (load before calling routine)

(continued)
```
XDEF Parser

Parser
MOVE D7,D6 ;initialize significant digits
MOVE #0=D0 ;initialize starting character position
MOVE.B (A0),D2 ;get first character
CMP.B #'-',D2 ;negative number?
BNE Positive
MOVE #256,(A2) ;store negative sign
ADDQ #1,D0 ;skip sign
SUBQ #1,D6 ;don't include sign in significant digit count
BRA Parse

Positive
MOVE #0,(A2) ;store positive sign
CMP.B #'+',D2 ;is there a positive sign?
BNE Parse
ADDQ #1,D0 ;skip sign
SUBQ #1,D6 ;don't include sign in significant digit count

Parse
MOVE D0,D3 ;save position of beginning character position

NoDecimal
MOVE.B (A0,D3),D2 ;get a character
CMP.B #'.',D2 ;decimal point found?
BEQ DecimalPoint
ADDQ #1,D3 ;skip to next character
CMP D3,D7 ;past last character?
BGT NoDecimal ;not decimal point or end of string

;This block handles number of the form XXXXXXXX - No decimal point present
;at all (i.e., they're integers)
MOVE #0,D5 ;set exponent (decimal point at right of #)
MOVE.B D6,4(A2) ;load length byte
BRA FillRecord

DecimalPoint
CMP D3,D0 ;is decimal point in first position?
BNE GreaterThanOne

;This block takes care of numbers of the form .XXXXXXXXX
;The next step is to get rid of zeros between the decimal point and the
;first significant digit.
ADDQ #1,D0 ;skip over decimal point

MoreZeros
MOVE.B (A0,D0),D2 ;get character
CMP.B '#0',D2 ;is it a zero?
BNE SetExponent
ADDQ #1,D0
CMP D0,D7 ;at end of string?
BGT MoreZeros

SetExponent
SUBQ #1,D6 ;don't include decimal point in sig. digits
MOVE D6,D5
MULU #1,D5 ;final exponent value

(continued)
Listing 12.1 (continued)

;note - D0 has position of first significant digit

SUB D0,D7 ;number of significant digits
MOVE D7,D6
MOVE.B D6,4(A2) ;load length byte
BRA FillRecord

;This block handles numbers that are greater than 1 and contain a decimal point. They translated to XXXXXXXXX. (note that decimal point is implied and not stored)

GreaterThanOne
SUBO #1,D6 ;don't include decimal point in sig. digits
MOVE D3,D1
MOVE D6,D5 ;move sig. digits to sign register
MOVE (A0),D2
BEO NoAdjustment
SUBO #1,D1

NoAdjustment
SUB D1,D5 ;subtract position of decimal point
MULU #-1,D5 ;make it negative
MOVE D3,D1 ;reload position of decimal point

Shift
ADDQ #1,D1 ;beyond last character?
CMP D7,D1
BGT Done ;beyond last character
BLT Continue ;before last character
MOVE.B (A0,D1),D2 ;decimal point in last position?
CMP.B '#',D2
BEQ Done ;ignore trailing decimal point - otherwise move last digit

Continue
MOVE.B (A0,D0),(A2,D2) ;move one character
ADDQ #1,D3
BRA Shift

Done
MOVE.B D6,4(A2) ;load length byte

FillRecord
MOVE D5,2(A2) ;load exponent
MOVE #Ø,D1 ;initialize loop counter
MOVE #5,D2 ;starting offset of string in decimal record
ADDQ #1,D6 ;include length byte in count

Top
MOVE.B (A0,D0),(A2,D2) ;move one character
ADDQ #1,D1
CMP D1,D6
BEQ Return ;all characters moved
ADDQ #1,D0
ADDQ #1,D2
BRA Top

Return RTS
The Parser

The parser subroutine, like any routine that examines strings and makes decisions based on the characters that are present, works on a set of rules that describe possible character sequences and the actions to be taken when those sequences are found. There are three general formats in which numbers might be entered from the keyboard: XXXX (an integer without a decimal point); .000XXX (a fraction less than one, with or without zeros between the decimal point and the significant digits); and XXXX.XX (a combination of integer and fraction). This simple parser does not handle numbers entered in scientific notation (e.g., 1.345E +06), though it could certainly be expanded to do so.

To better understand the logic of the parser, take a look at the pseudocode in Figure 12.1. This presents an English-like version of the subroutine's logic. The general strategy is to first examine the string for a plus or minus sign in the first character position, at which point the sign of the number can be determined and stored directly into the data structure set aside to hold the canonical decimal format. The second step is to determine which of the three forms described in the previous paragraph (integer, fraction, or combination) fits the character string.

Figure 12.1 Parser Pseudocode

Initialize number of significant digits as equal to total characters in string.

Get the first character in the source string.

If the first character is a minus sign then

   Store value for negative number directly into canonical data structure;

   Decrement the number of significant digits by 1 (sign doesn't count)

Else

   If first character is a plus sign then

      Decrement the number of significant digits by 1;

      Store value for positive number directly into canonical data structure.

Get the "next" character. [will be first character again if no sign was present]

While the character being examined is not a decimal point do

   Get another character.
Figure 12.1 (continued)

If no decimal point is found then

(number is an integer)

Set 0 as the exponent value;

Store number of significant digits in canonical data structure

Else

If the decimal point is in the first position then

(number is all fraction)

Get next character;

While the character being examined is not a 0 do

Get next character;

Decrement number of significant digits by 1 to ignore decimal point;

Compute exponent by subtracting position of first non-zero digit from number of significant digits and multiplying by -1;

Store number of significant digits in canonical data structure

Else

(number has integer and fraction parts)

Decrement number of significant digits by 1 to ignore decimal point;

Compute exponent by subtracting position of decimal point from number of significant digits and multiplying by -1;

Set a pointer to the first character to the right of the decimal point;

While the pointer less than the last character do

Move the character one place to the left; [eliminates decimal point from source string]

Increment the pointer;

If the last character is not a decimal point then

Move the last character;

Store in number of significant digits in the canonical data structure.

Store the exponent in the canonical data structure.

Initialize a counter to 0.

(continued)
While the counter is less than or equal to the number of significant digits do

Move one character from the source string to the canonical data structure;

Increment the counter.

Stop.

Handling integers is straightforward; the decimal point is already in the correct place. Though it is not stored with the number, all integers have implied decimal points directly to the right of the number. The exponent for an integer is always 0.

If the decimal point is in the first position (after the sign, if one is present), then the number is a fraction. Fractions must be converted to integers with no leading zeros and the exponent adjusted accordingly. A fractional exponent will be equal to the number of digits in the original number (including leading zeros) multiplied by $-1$; a negative exponent indicates that the decimal point should be moved to the left. The number of significant digits for the canonical decimal format is determined by finding the left-most non-zero digit. All characters from that point to the end of the string are considered significant digits.

Numbers that contain both integer and fractional portions have an exponent equal to the number of fractional digits multiplied by $-1$. The string must also be adjusted to remove the decimal point; each digit in the fractional portion of the string is moved one position to the left. The number of significant digits is equal to the number of characters in the string less one for the decimal point and one for a sign, if present.

---

**Programming Technique — Using Separately Assembled Subroutines**

The parser just described is designed so that it can be used by many different applications. It appears to an application much like one of Macintosh’s operating system routines in that parameters are passed to it in registers — A0 contains the address of the source string, A2 the address of the destination data structure, and D7 the total number of characters in the source string. An application loads the registers and then does a **JSR** to call the routine.
It would be inefficient to include the code for the parser in each application that needs to do decimal-to-binary conversions. Instead, the parser is assembled separately and kept in its own .Rel file that can be used whenever needed. As you write many applications, you may develop an entire library of utilities like the parser that can be used whenever needed without duplicating their code.

In order to use separately assembled subroutines, three things must happen:

1. The source code of the subroutine must indicate that it will be called by another program (an external definition);
2. The source code of any application calling the subroutine must indicate that the subroutine is not part of the application's code (an external reference);
3. The subroutine must be linked to the application during the linking process.

The assembler directive XDEF (external definition) is used whenever a symbolic address in a piece of code will be referenced by another program. For example, the first line in the parser subroutine is:

```
XDEF Parser
```

which indicates that some other piece of code will be using that symbolic address.

The assembler directive XREF (external reference) alerts the Assembler that a specific symbolic address cannot be found in the code being assembled but can be found in some other program. Any application that uses the parser must therefore include the directive:

```
XREF Parser
```

before it executes a JSR to the code. The XREF will prevent the Assembler from returning an "undefined label" error.

External references are satisfied by the Linker, which provides the actual addresses of all external routines. Therefore, the names of any .Rel files that contain symbolic addresses defined in XREF directives must be included in the Linker control file. If a program called Math uses the parser, its Linker control file will include:

```
Math.Rel
Parser.Rel
```
FP68K has routines to generate six different numeric formats from the canonical decimal format:

1. Extended — an 80-bit floating point number
2. Double — a 64-bit floating point number
3. Single — a 32-bit floating point number
4. Integer — a 16-bit integer
5. Longinteger — a 32-bit integer
6. Computational (also known as Accounting) — a 64-bit integer

Arithmetic operations return their results in the extended format. That format looks somewhat like a binary version of the canonical decimal format. Bit 79 is reserved as a sign bit for the mantissa; it holds 0 for a positive number and 1 for a negative number. The exponent is 15 bits long, stored in bits 64 through 78. Bits 0 through 63 are reserved for the mantissa.

Floating point exponents are stored as binary integers. They are the power to which 2 is raised and then multiplied by the mantissa. One way to store them would be to allocate one exponent bit as a sign bit and use two’s complement notation. The resulting 14-bit exponent would have the range ±16,383. Exponents, though, are stored without a sign bit using a technique known as excess notation.

Excess notation means that some fixed quantity is added to every value of the exponent. The exact value of the excess varies from computer to computer, but it is always enough to make the smallest exponent value 0. The Macintosh uses an excess factor of $3FFF$, or 16,383. That means that the smallest possible exponent that can be stored in 15 bits is $-16,383$ and the largest $+16,384$. The Macintosh can therefore store floating point numbers in the range $2^{-16,383}$ through $2^{+16,384}$. This is an enormous range, well beyond that demanded by all but the most intensive scientific and statistical applications.

The mantissa is also a binary number. The first bit (bit 63) always has the value one. There is an implied decimal point directly between bits 62 and 63; bits 0 through 62 contain the fractional portion of the mantissa.

As an example, consider the decimal number 32. It is passed to FP68K’s decimal to binary conversion routines with a sign of 0, an exponent of 0, and two significant digits (3 and 2). After being converted, it will appear in memory as
$4004 8000 0000 0000 0000. The leftmost word contains the sign and the exponent: $0100 0000 0000 0100. Note that the high-order bit, the sign bit, is 0 to indicate a positive mantissa. The other 15 bits are the exponent. To determine the true value of a Macintosh floating point exponent for a positive number, subtract $3FFF (or subtract $4000 and add 1): $4004 - $3FFF leaves $0005. The mantissa will therefore be multiplied by $2^5. If we expand the first word of the mantissa to binary, we get $1000 0000. Since the decimal place is directly to the right of the high-order bit, the mantissa is actually $1.0000000. The complete value of the number is $1.0000000 * 2^5 or 32.

Any floating point number with a positive exponent and and positive mantissa will have an exponent word with a value between $4000 and $7FFF. It is useful to become accustomed to viewing floating point representations in hexadecimal, since that is how the contents of memory locations are displayed by the debugger.

As a second example, let's include a fraction with the test number and make it 32.5. Like 32, .5 is an even power of 2, $2^{-1}$. The canonical decimal format will contain a 0 for the sign bit, a $-1$ for the exponent, and a 3 for the number of significant digits (3, 2, and 5). FP68K will convert 32.5 to $4004 8200 0000 0000 0000. The exponent is the same as that for the even value 32; it is the mantissa that is different. If we expand the first word of the mantissa to binary, we get $1000 0000 0000 0000 0000 or $1.000 0000 0000 0000 0000 0000 0000 0000 0000 * 2^5. Moving the decimal point in the mantissa five places to the right, produces $1.00000001, which is precisely 32.5.

Negative mantissas produce a change in the value of the exponent word. For example, -32.5 is stored as $C004 8000 0000 0000 0000. In binary, the exponent word is $1100 0000 0000 0100. The high-order bit is set, representing a negative mantissa. To determine the true value of the exponent, first subtract $8000 to strip off the sign bit and then subtract $3FFF to get rid of the excess. For example, $C004 - $8000 = $4004; $4004 - $3FFF = $0005. Numbers with negative mantissas and positive exponents will have exponent word values between $C000 and $FFFF.

Numbers with positive mantissas and negative exponents produce exponent word values in the range $0000 - $3FFF. For example, the quantity .5 generates an exponent of $-1. It is stored by adding $1 to $3FFF, which produces an exponent of $3FFE. Numbers with negative mantissas and negative exponents have exponent word values in the range $8000 - $BFFF; - .5 has an exponent word of $BFFE.

Macintosh's other two floating point formats (with 32 and 64-bit mantissas) are stored exactly like the extended format. They simply have less accuracy in the mantissa, since they store fewer bits.

The 16- and 32-bit integer formats are the same as those manipulated by the integer arithmetic instructions that are part of the 68000 instruction set. The 64-bit integer format can be used to obtain extra accuracy and range when doing computations. It is, however, too long for conversion with the Binary-Decimal Conversion Package.
Executing a Binary-to-Decimal Conversion

The FP68K routines are part of Package 4. Like all packages, its routines are called by pushing the routine identifier onto the stack and then calling the package with __Pack4. The packages we have discussed previously, though, have had a relatively small number of routines, while FP68K has somewhere around 120. For most FP68K routines, the routine identifier is the sum of the operation code (identifying the type of operation the routine will perform) and an operand format code that identifies the format of the source operand.

The operation code for converting from decimal to binary is $0009. To produce an extended floating point result, $0000 is added to the operation code. If the conversion should produce a longinteger result, $2800 is added to the operation code. Each of the six available formats has a unique operand format code.

Most FP68K routines, including decimal to binary conversions, require the following actions:

1. Push a pointer to the source operand onto the stack.
2. Push a pointer to the destination operand onto the stack.
3. Push the routine identifier onto the stack.
4. Invoke the package.

To convert from the canonical decimal format to an extended floating point number, you might use this code:

```
PEA DecimalRecord(A5)
PEA BinaryNumber(A5)
MOVE #$0009, -(SP)
__Pack4
```

Doing the actual conversion is really quite straightforward. The biggest problem facing a programmer is generating the appropriate routine identifier. The Macintosh 68000 Development System has simplified the process by providing a file (SANEMacs.Txt) containing equates and macros.
Programming Technique — Using Macros

A macro is a short block of code that is assigned a name. The name of the macro is then used within an application to represent the entire macro. During assembly, the macro name is replaced by the block of code associated with the macro's name. Note that this is very different from using a subroutine. If a subroutine is called repeatedly, the program merely branches to where the subroutine is located and executes it; the code of the subroutine appears only once in the program. A macro name is a place holder that will be replaced by the body of the macro when the program is assembled; a macro that is used repeatedly in the same program will generate repeated code. Macros are therefore generally short, less than 10 lines of code.

Macros must be defined before they can be used. Macintosh macros can have one to two formats. Either:

```
.MACRO NameOfMacro [ArgumentList]  
{body of macro goes here - any executable code is allowed}
.ENDM
```

or

```
MACRO NameOfMacro [ArgumentList = ]  
{body of macro goes here - any executable code is allowed}
```

The first format is referred to as a "Lisa-style" macro, the second as a "Macintosh-style" macro. Both work equally well with the MDS.

Macros can contain arguments, data that are passed to the macro from the application. Macro arguments work very much like the arguments passed to Pascal functions and procedures. The arguments used in the macro definition are dummy arguments. When the program containing the macro is assembled, the arguments are substituted by position. Consider, for example, a macro that will compute the position of a single field within TapeArray:

```
MACRO AddressCompute  R1,R2,R3 =
MULU #64,[R1]
ADD [R1],[R2]
ADD [R3],[R2]
```

In this particular macro, R1 is a place holder for some register that contains the record number. R2 stands for an address register containing the starting address
of TapeArray. R3 is a constant that stands for the byte offset into a TapeArray record. Each dummy argument is surrounded by braces.

When this macro is used in an application, the programmer will write:

```
MOVE TotalRecords,D0
LEA TapeArray,A0
MOVE #oRating,D1
AddressCompute D0,A0,D1
```

When the program is assembled, this code will be generated:

```
MOVE TotalRecords,D0
LEA TapeArray,A0
MOVE #oRating,D1
MULU #64,D0
ADD D0,A0
ADD D1,A0
```

The arguments specified after the name of the macro in the program code will be substituted by position for the dummy arguments in the macro's argument list.

The file SANEMacs.Txt can be found on MDS2. It contains equates for the FP68K and ELEMS68K operand format codes and operation codes. More importantly, it also contains one macro for each FP68K and ELEMS68K routine. The macros compute the appropriate routine identifier, push it onto the stack, and then invoke the package. SANEMacs.Txt should be INCLUDEd in any application that uses FP68K or ELEMS68K.

The macro for converting from the canonical decimal format to the extended floating point format is:

```
.MACRO
MOVE.W FDEC2X
JSRFP #FFEXT + FOD2B, - (SP)
.ENDM
```

where FFEXT has previously been equated to $0000 and FOD2B to $0009. JSRFP is another macro defined within SANEMacs.Txt. It takes care of invoking the package. If you look at the definition of JSRFP, you will see that the package is invoked with ___FP68K, but if disassembled by the debugger, it appears as ___Pack4. Both have the same trap value and are equivalent. ___ELEMS68K is also equivalent to ___Pack5.

There is an important naming convention to be aware of in the operation code macros. The last character of most of the macro names identifies the type of source operand the operation will handle. For example, FADDX will look for an extended source operand to add to an extended destination operand. FADDD will add a double source operand to an extended destination operand. The suffix S indicates
a single source operand, C a computational, L a longinteger, and I an integer. For each type of operation (e.g., addition, subtraction, comparison, etc.) there are six routines, one for each possible type of source operand.

The binary to decimal conversion can be simplified by using the pre-defined macro:

\[
\begin{align*}
\text{PEA} & \quad \text{DecimalRecord}(A5) \\
\text{PEA} & \quad \text{BinaryNumber}(A5) \\
\text{FDEC2X} &
\end{align*}
\]

The discussion in the rest of this chapter assumes that SANEMacs.txt has been INCLUDED in the application source code and that the pre-defined macros are available.

---

An Overview of the FP68K and ELEMS68K Routines

FP68K routines fall into two major groups — arithmetic routines and those that provide non-arithmetic utility functions.

The Arithmetic Routines

The arithmetic routines include:

1. Addition (one for each type of source operand — FADDX, FADDD, FADDS, FADDI, FADDL, FADDC)
2. Subtraction (one for each type of source operand — FSUB + the letter that identifies the operand type)
3. Multiplication (one for each type of source operand — FMUL + operand type identifier)
4. Division (one for each type of source operand — FDIV + operand type identifier)
5. Square root (FSQRTX — works only with an extended operand)
6. Round to integer (FRINTX — works only with an extended operand)
7. Truncate to integer (FTINTX — works only with an extended operand)
8. Remainder — returns the remainder of a division operation (one for each type of source operand — **FREM** + operand type identifier)

9. Base 2 logarithm — returns the exponent (**FLOBX** — works only with an extended operand)

10. Base 2 exponentiation — the source operand is the power to which 2 is raised and then multiplied by the destination operand (**FSCALBX** — works only with an extended source operand)

Calls to arithmetic routines (with the exception of numbers 4 – 7 and 8 – 10 above) have the following general form:

```
PEA source operand
PEA destination operand
OperationMacroName
```

The result of the operation is placed in the destination operand. That means the original contents of the destination operand is erased by the result. For example, a **FADD** operation has the same effect as the Pascal statement:

```pascal
A := A + B
```

Therefore, if the destination operand must be retained for further use, it should be copied to another storage location before being passed to the FP68K routine.

Code to add a longinteger to an extended floating point number appears as:

```
PEA LongIntegerNumber(A5)
PEA ExtendedNumber(A5)
FADDL
LongIntegerNumber DS.L 1
ExtendedNumber DS 5
```

Note that all operands are passed as pointers to main memory locations where the operands are actually stored.

The other routines, including square root and rounding, require only one operand:

```
PEA source operand
OperationMacroName
```

The result replaces the source operand. In the case of square root, it has the effect of executing the Pascal statement:

```pascal
A := SQRT(A)
```
To actually compute a square root:

```
PEA
FSQRTX
```

ExtendedNumber(A5)

```
ExtendedNumber DS 5
```

The Utility Routines

FP68K non-arithmetic routines include:

1. Negation (FNEGX — works only with an extended operand)
2. Absolute value (FABSX — works only with an extended operand)
3. Conversion from all six formats to extended (FX2X, FD2X, FS2X, FI2X, FL2X, FC2X)
4. Conversion from extended to the other five formats (FX2D, FX2S, FX2I, FX2L, FC2X)
5. Decimal to binary conversion (FDEC2 + operand type identifier, as discussed above)
6. Binary to decimal conversion (F?2DEC, where ? is replaced by the operand type identifier).
7. Comparison between two floating point numbers (FCMP + operand-type identifier or FCPX + operand-type identifier). These comparisons can be used where it makes logical sense to use the CMP instruction.
8. Branching based on the result of floating point comparisons (FBEQ, FBLT, FBLE, etc.). These macros contain instructions that test the condition codes. They assume that the appropriate floating point comparison has been performed. They should be used in place of a Bcc instruction.

Negation and absolute value each require only one operand. For example, to obtain the absolute value of some floating point number:

```
PEA
FABSX
```

SomeNumber(A5)

```
SomeNumber DS 5
```

As with the single operand arithmetic routines, the result of a single operand utility routine will overwrite the source operand.

The internal conversion routines, the decimal-to-binary conversion routines, and the comparison routines require two operands. As discussed earlier, the pointer to the source operand goes deepest in the stack, followed by the pointer to
the destination operand. Note that while the first two sets of routines replace the
destination operand with the result of the operation, the comparison operations do
not affect either operand; they merely set the flags in the status register.

Binary-to-decimal conversions are the only routines that use three operands.
Performing these conversions is discussed later in the chapter.

The floating point branch instructions are used exactly like any other Bcc
instruction. For example:

    FBEQ  NextLabel

assumes that two floating point numbers have just been compared. The program
will branch to NextLabel if the two numbers were equal. Note that FBEQ is not a
new instruction; it is a macro with an argument. Nonetheless, the floating point
branch macros can be used as if they were actual instructions.

The ELEMS68K Routines

ELEMS68K contains a number of advanced logarithmic, trigonometric, and
exponential functions. Most work only with extended operands. The following
routines require one extended operand and replace it with the result:

1. Natural (base e) logarithm (FLNX)
2. Base 2 logarithm (FLOG2X)
3. Natural logarithm of 1 + extended operand (FLNX)
4. Base 2 logarithm of 1 + extended operand (FLOG21X)
5. Raising e to a power specified by the extended operand (FEXPX)
6. Raising 2 to a power specified by the extended operand (FEXP2X)
7. Raising e to a power specified by the extended operand − 1 (FEXP1X)
8. Raising 2 to a power specified by the extended operand − 1 (FEXP21X)
9. Sine (FSIX)
10. Cosine (FCOSX)
11. Tangent (FTANX)
12. Arctangent (FATANX)
13. Random number generator (FRANDX)
For example, to find the sine of a number:

```
PEA
FSINX

ExtendedNumber(A5)
```

The two exponential routines require two operands:

1. Raise an extended operand (the destination operand) to an integer power (the source operand) (**FXPWRRI**)
2. Raise an extended operand (the destination operand) to an extended power (the source operand) (**FXPWRY**)

Note that even when using an integer operand, a pointer to that operand is pushed onto the stack rather than value of the operand itself. For example, to perform an integer exponentiation:

```
PEA
PEA
FXPWRRI

IntegerExponent(A5)
ExtendedNumber(A5)
```

```
IntegerExponent(A5)
ExtendedNumber(A5)
```

```
DS 1
DS 5
```

**ELEMS68K** also contains routines to compute compound interest and annuities. Each requires three extended operands — two source (the interest rate and the number of compounding periods) and one destination (the starting principle). A pointer to the interest rate is deepest in the stack, followed by a pointer to the number of compounding periods and a pointer to the destination operand:

1. Compound interest (**FCOMPOUND**)
2. Annuity (**FANNUITY**)

For example, this code will compute compound interest:

```
PEA
PEA
PEA
FCOMPOUND

InterestRate(A5)
NumbOfPds(A5)
StartPrinc(A5)
```

```
InterestRate(A5)
NumbOfPds(A5)
StartPrinc(A5)
```

```
DS 5
DS 5
DS 5
```
Converting from a binary number back to the canonical decimal format is not precisely the opposite of converting from decimal to binary. There are two possible output formats — floating point and fixed point, both of which are delivered in the canonical decimal format record.

To see the difference, consider the number 32.5. As noted earlier, the extended floating point format of 32.5 is \$4004\ 8200\ 0000\ 0000\ 0000\ 0000. If converted to a floating point number with three significant digits, the canonical decimal format will appear as \$FFFF\ 0333\ 3235\ ...\ or\ 325\ *\ 10^{-1}. A floating point version of the number (assuming that the conversion requests three digits to the right of the decimal point) appears as \$FFFD\ 0533\ 3235\ 3030\ ...\ which is 32500\ *\ 10^{-3}\ or\ 32.500. Floating point numbers are designed to be displayed in the mantissa and exponent format (e.g., 3.25E1) while fixed point numbers have their decimal points embedded in the number itself, as in 32.500.

The output format of a binary-to-decimal routine is controlled by a format record:

```pascal
TYPE DecForm = RECORD
    style : (0, 256); \{0 = float; 256 = fixed\}
    digits : INTEGER
END;
```

The style word stores the flag for the style in bit 8. Therefore, a value of 0 indicates that the number should be converted to a floating point number, while a value of 256 (a 1 in bit 8) indicates that the conversion should be to fixed point.

The meaning of the `digits` field depends on whether the conversion is to float or fixed. For a floating point number, `digits` indicates the total number of significant digits that should be stored in the canonical decimal format. For a fixed point number, the same field contains the number of fractional digits (those to the right of the decimal point) that are to be stored.

### Doing a Binary to Decimal Conversion

The macro for binary to decimal conversions is `F?2DEC`, where ? is replaced by the letter which indicates the format of the source binary number. For example,
**FX2DEC** will convert an extended operand while **Fl2DEC** will convert an integer operand. Note that while the FP68K routines will handle integer and longinteger operands, they can more easily be converted by using the Binary-Decimal Conversion Package.

The binary-to-decimal conversion routines require three operands. A pointer to the format record is deepest in the stack, followed by a pointer to the source operand and finally a pointer to the destination data structure (the canonical decimal format). Assuming that a binary number in extended floating point format is stored in **BinaryNumber**(A5), the following code will convert that binary number to its floating point representation in the canonical decimal format with six significant digits (the number of significant digits in the example is arbitrary):

```
LEA FormatRec(A5), A0
MOVE #0,(A0) ; style = float
MOVE #6,2(A0) ; six significant digits
MOVE.L A0, -(SP) ; put pointer on stack
PEA BinaryNumber(A5) ; pointer to source operand
PEA DecimalRec(A5) ; destination data structure
FX2DEC ; routine macro
```

Converting the same extended binary number to a fixed point format with an arbitrary three digits to the right of the decimal point is only slightly different:

```
LEA FormatRec(A5), A0
MOVE #256,(A0) ; style = fixed
MOVE #3,2(A0) ; three digits to right of decimal point
MOVE.L A0, -(SP) ; pointer to format record
PEA BinaryNumber(A5) ; pointer to source operand
PEA DecimalRec(A5) ; destination data structure
FX2DEC ; routine macro
```

; uses same data structures as example immediately above

The question remains as to when binary numbers should be converted to fixed point and when they should be converted to floating point. The answer lies in how the numbers will be displayed.

**Displaying Numbers from a Canonical Decimal Format**

Numbers that are to be displayed in fixed point format (i.e., with embedded decimal points) should be converted to fixed and numbers that are to be displayed
in floating point format (i.e., with mantissa and exponent) should be converted to float. The decision generally rests on the size of the number; that is, there comes a point where numbers contain too many digits for effective display. For example, 3.78E44 is the digits 378 followed by 42 zeros. Most applications will choose to display such a large number in its floating point form. On the other hand, 37.8 is conveniently displayed as a floating point number and makes more sense to the user than 3.78E1. The actual point at which any given application will switch from fixed to floating point display will vary from application to application.

Whichever format an application chooses to use, there still remains the problem of taking the number from the canonical decimal format and reformatting it into a string of ASCII characters that can be either printed with DrawString or incorporated into a text edit record. The task is more or less the opposite of the function provided by the parser subroutine, which converts strings to the canonical decimal format.

Listing 12.2 contains two subroutines that will convert floating and fixed point numbers to ASCII strings for output. Like the parser, the formatter routines are designed as utility routines to be assembled separately from program code and then called as external references. Each subroutine requires two parameters as input — a pointer to the string containing the number in canonical decimal format (in A1) and a pointer to the output string (in A2). To call either routine (assuming it has been properly linked to the main program code), load the pointers in the address registers and do a JSR to the appropriate symbolic address (FormatFloat or FormatFixed).

*Listing 12.2  Formatting Numeric Strings for Output*

|----------------------- Numeric Output Formatter -------------------|
| Parameters on entry: |
| A1 : pointer to record containing canonical decimal format |
| A2 : pointer to output string |

XDEF FormatFloat
XDEF FormatFixed
.TRAP _Pack7 $A9EE

FormatFloat
MOVE.L #0, D0 ; initialize register
MOVE #1, D3 ; character pointer in output string
MOVE #0, D5 ; starts at one to leave room for length byte
MOVE.B 4(A1), D1 ; initialize character counter
MOVE 2(A1), D2 ; number of significant digits
MOVE D1, D0 ; exponent in canonical decimal format
ADD D2, D0 ; place for output exponent
SUBQ #1, D0 ; final output exponent (exp. + sig. digits -1)

(continued)
Listing 12.2 (continued)

```
MOVE (A1),D6
CMP #0,D6
BEQ NoSignNeeded
MOVE.B #'-',(A2,D3)
ADDQ #1,D3

NoSignNeeded
MOVE #5,D4
MOVE.B (A1,D4),(A2,D3)
ADDQ #1,D3
ADDQ #1,D5
CMP.B D1,D5
BEQ InsertExponent
MOVE.B #'.',(A2,D3)

More Digits
ADDQ #1,D3
ADDQ #1,D4
MOVE.B (A1,D4),(A2,D3)
ADDQ #1,D5
CMP.B D1,D5
BNE More Digits

Insert Exponent
ADDQ #1,D3
MOVE.B #'E',(A2,D3)
ADDQ #1,D3
LEA ExponentString(A5),A0
EXT.L D0 ;propagate sign through high-order word of register
MOVE #0,-(SP) ;convert integer to string
-Pack?
MOVE.B (A0),D1 ;length of exponent string
MOVE #1,D0 ;starting offset into exponent string

More Exponent
MOVE.B (A0,D0),(A2,D3)
CMP.B D1,D0
BEQ SetLength ;all characters moved
ADDQ #1,D0
ADDQ #1,D3
BRA More Exponent

SetLength
MOVE.B D3,(A2) ;install length byte in first position

RTS

Format Fixed
MOVE.B 4(A1),D1 ;number of significant digits
MOVE 2(A1),D2 ;exponent
MOVE.B D1,D5 ;save significant digits to fool with
MOVE D2,D7 ;save exponent to fool with
BGE OK
MULU #1,D7 ;make negative exponent positive
```

(continued)
OK SUB D7,D5
BLE Fraction ;number is a fraction - must handle separately

ADD.B D1,D2 ;number of digits to left of decimal point
MOVE #1,D3 ;position pointer in output string
MOVE #0,D4 ;digit counter
MOVE #5,D5 ;initial offset into canonical decimal record
MOVE (A1),D6 ;get sign
BEQ CopyLoop ;positive number
MOVE.B #",(A2,D3) ;load a negative sign

CopyLoop
CMP.B D2,D4
BNE MoveOne
ADDQ #1,D3
MOVE.B ",,(A2,D3) ;insert decimal point

MoveOne
ADDQ #1,D3
MOVE.B (A1,D5),(A2,D3) ;move one character
ADDQ #1,D4
ADDQ #1,D5
CMP D4,D1
BNE CopyLoop

MOVE.B D3,(A2) ;load length byte
RTS

Fraction
MOVE #1,D3 ;initialize position pointer
MOVE (A1),D6 ;get sign
BEQ None ;positive number
MOVE.B ",,(A2,D3) ;load negative sign
ADDQ #1,D3

None
MOVE.B ",,(A2,D3) ;loading leading zero and decimal point
ADDQ #1,D3 ;must be two steps because of possibility of
MOVE.B ",,(A2,D3) ;uneven starting address

MOVE #0,D0 ;count zeros
MOVE D5,D4 ;this move is just to affect status register
BGT AnotherZero
MULU #1,D5 ;get absolute value

AnotherZero
CMP D0,D5 ;enough 0's added?
BEQ GetDigits
ADDQ #1,D3
MOVE.B ",,(A2,D3)
ADDQ #1,D0
BRA AnotherZero

GetDigits
MOVE #0,D0 ;to count significant digits

(continued)
Listing 12.2 (continued)

MOVE #5,D5 ;offset into canonical decimal format
AnotherDigit
ADDQ #1,D3
MOVE.B (A1,D5),(A2,D3)
ADDQ #1,D5
ADDQ #1,D0
CMP D0,D1
BGT AnotherDigit

MOVE.B D3,(A2) ;load length byte
RTS

ExponentString DS.B 6
END

Formatting Floating Point Numbers

Pseudocode for FormatFloat, the floating point formatter, can be found in Figure 12.2. The routine must first compute the exponent for output. This is different from the exponent stored in the canonical decimal format, since the significant digits are stored as an integer but will be displayed in the form X.XXXX.... In fact, the exponent for output is equal to:

Exponent from canonical decimal format - # sig. digits + 1

This exponent is an integer and must ultimately be converted to a string. FormatFloat uses **NumToString** from the Binary-Decimal Conversion Package for that purpose. **NumToString** is very convenient, since it will insert a minus sign at the head of its output string if the integer being converted is negative.

Figure 12.2 Floating Point Formatter Pseudocode

Initialize pointer to output string \{set to 0 if no length byte; set to 1 if length byte is required\}

Get number of significant digits from canonical decimal format.

Get exponent from canonical decimal format.

Compute exponent for output as Exponent from canonical decimal format - Number of Significant Digits + 1.

Get sign from canonical decimal format.

(continued)
If number is negative then
   Put negative sign in first position of output string;
   Increment pointer to output string.
Move first significant digit from canonical decimal format to output string.
Increment pointer to output string.
Move decimal point to output string.
Increment pointer to output string.
Repeat
   Move one significant digit from the canonical decimal format to the output string;
   Increment pointer to output string
Until no significant digits remain.
Put "E" in output record.
Increment pointer to output string.
Convert integer value of exponent for output into a string.
While exponent characters remain do
   Move one exponent character to the output string;
   Increment pointer to output string.
Load length byte at beginning of output string (equal to pointer to output string)
   {the length byte is optional - leave it out if output string will be incorporated into a text edit record}

FormatFloat checks the first word of the canonical decimal format to determine the sign of the number. If the number is negative, a minus sign is stored in the first position of the output string. Then the first significant digit is moved from the canonical decimal format record to the output record, followed by a decimal point. The remaining significant digits are placed immediately after the decimal point. The next character is an 'E'. Finally, the exponent, as converted by NumToString, is moved to the output string.
FormatFloat also places a length byte at the beginning of the output string. If the string is to be displayed by DrawString, then the length byte is required, but if the output string is to be incorporated into a text edit record, then there should be no
length byte. To modify FormatFloat to format without a length byte, do the following:

1. Initialize the output string position pointer to 0 rather than 1 (register D3)
2. Remove the instruction that loads the lengths byte (MOVE.B D3,(A2))

The same holds true for FormatFixed, the fixed point formatter, since it too was designed to include a length byte.

If you look closely at the assembly language code for FormatFloat in Figure Listing 12.2, you will see one 68000 instruction that we haven't discussed: EXT. EXT stands for "extend"; it takes one operand — a data register. If the instruction is word-sized, it will copy, or extend, the value of bit 7 into bits 8 through 15. A longword-sized operation will copy the value of bit 15 into bits 16 through 31.

Why is EXT important? When the exponent is retrieved from the canonical decimal format it is word-sized. The operations that compute the final exponent are also word-sized. That means that the exponent is stored in D0 as $0000XXXX, where the X's represent the magnitude of the exponent. A problem arises if the word-sized exponent is passed to NumToString. NumToString expects a longword operand. It makes a decision as to the sign of the number on the value in bit 31. The value of bit 31, therefore, also determines whether the number in D0 will be interpreted as true magnitude or two's complement form.

Consider an exponent of −3. In its word-sized form it will be stored as $0000FFFFC. NumToString, though, will pick up the zero in bit 31 and assume that the register contains a positive number. The $FFFFC will be interpreted as the true magnitude of a positive number, or 65535. The solution is to extend the sign bit of the word-sized operand (bit 15) into the high-order word of the register. Assuming that the contents of D0 are $0000FFFFC, the instruction:

`EXT.L D0`

will produce a result of $FFFFFFFC. Since bit 31 is set, NumToString will correctly interpret the contents of D0 as a negative number in two's complement form.

**Formatting Fixed Point Numbers**

Pseudocode for FormatFixed appears in Figure 12.3. Formatting fixed point numbers is slightly more complex than formatting floating point numbers, since numbers that are all fraction must be handled separately from numbers that have both integer and fractional parts. Numbers that are all fraction can be detected by subtracting the absolute value of the exponent in the canonical decimal format from the number of significant digits; any number that has to have its decimal point moved more places to the left than there are significant digits is less than one.

For numbers that have both integer and fractional parts, the first task is to determine how many of the significant digits lie to the right of the decimal point by
summing the exponent and the number of significant digits. Though this procedure may at first seem a bit odd, consider that since the significant digits are stored in the canonical decimal format as if they were an integer, the exponent will always be negative or zero.

**Figure 12.3 Fixed Point Formatter Pseudocode**

Get number of significant digits from canonical decimal format.

Get exponent from canonical decimal format.

Make negative exponent positive. \{need absolute value of exponent\}

Subtract absolute value of exponent from number of significant digits.

If subtraction gives positive result then \{number is integer or integer and fraction\}

Initialize pointer to output string; \{0 for no length byte; 1 for length byte\}

Compute number of digits to left of decimal point by adding exponent to number of significant digits;

Get sign of number from canonical decimal format;

If number is negative then

Put a negative sign in output string;

Increment output string pointer;

While significant digits remain do

If place for decimal point found then

Put decimal point in output string;

increment output string pointer;

Move on significant digit from the canonical decimal format to the output string;

Increment output string pointer;

Load length byte at beginning of output string \{optional\}

Else \{number is all fraction\}

Initialize pointer to output string;

Get sign of number from canonical decimal format; (continued)
Figure 12.3 (continued)

If number is negative then

- Put negative sign in output string;
- Increment output string pointer;
- Put leading zero in output string;
- Increment output string pointer;
- Put decimal point in output string;
- Increment output string pointer;

Compute number of zeros needed between decimal point and first significant digit as absolute value of difference between number of significant digits and exponent

While zeros remain do

- Put zero in output string;
- Increment output string pointer;

Repeat

- Move a significant digit from canonical decimal format to output string;
- Increment output string pointer

Until all significant digits have been moved;

Load length byte as equal to output string pointer. {optional}

FormatFixed then checks the sign of the number by looking at the first word of the canonical decimal format record and moves a minus sign to the output string if appropriate. It then begins to move the significant digits, checking after each digit is moved to determine if the place to insert the decimal point has been found. The decimal point is inserted and the remaining significant digits are transferred.

In order to format a number that is less than one, FormatFixed must determine how many zeros must be inserted between the decimal point and the first significant digit. The number of zeros is equal to the difference between the absolute value of the exponent from the canonical decimal format and the number of significant digits.

As with formatting fixed point numbers greater than one, the procedure for numbers less than one first handles the sign of the number by checking the first word of the canonical decimal format record. A minus sign is moved to the output
string if appropriate. Then a leading zero and a decimal point are inserted in the output string; the leading zero is, of course, not required, but simply creates a nicer-appearing number.

The zeros which come between the decimal point and the significant digits are the next characters that are inserted in the output string. Finally, the significant digits themselves are moved.

Like the parser, the formatters are intended as examples. Feel free to enhance and modify them to suit the needs of your particular application.

---

**Questions and Problems**

1. What will be stored in register D0 after the execution of the following block of code:

   ```
   LEA  # '3000000000',A0
   MOVE #1, -(SP)
   __Pack7
   ;convert to longinteger
   ```

   Hint: consider the maximum quantity that can be stored in a longinteger location and what happens when it overflows.

2. For each floating point number below, indicate the value of the sign, exponent, and significant digits as they would be stored in Macintosh's canonical decimal format.

   a. 84867  
   b. 363.985  
   c. -.00126  
   d. \(-48.88 \times 10^{12}\)  
   e. \(-3.1313 \times 10^{-9}\)  
   f. \(.011927 \times 10^{43}\)

3. Convert the following decimal floating point numbers to Macintosh's 80-bit extended floating point format. Express your answer in hexadecimal.

   a. 32,767  
   b. \(-32,767\)  
   c. \(8.99 \times 10^{38}\)  
   d. \(-10.33 \times 10^{-18}\)  
   e. \(.003 \times 10^{-51}\)  
   f. \(-.0101 \times 10^{67}\)

4. Convert each 80-bit floating point number below to decimal. For base 10 exponents between 4 and \(-4\) produce a fixed-point number; otherwise, produce a decimal floating point number.

   a. \$400A 32A0 0000 0000 0000  
   b. \$6BBB 1246 0000 0000 0000
5. Write a Macintosh-style macro that computes the area of a circle. Pass the radius of the circle to the macro in a data register. Return the answer in a different data register.

6. Write a Macintosh-style macro that compares two characters and returns whichever character is alphabetically greater. Pass the characters to the macro in D0 and D1. Return the result in D2.

7. Using the macros defined in SANEMacs.Txt, write code to perform the floating point conversions below. Assume the canonical decimal format is stored in the applications globals area under DecimalFormat; storage for the result has been allocated as ConvertedNumber.
   a. canonical decimal format to double precision (64-bit) floating point
   b. canonical decimal format to long integer
   c. canonical decimal format to computational

8. Using the macros defined in SANEMacs.Txt, write code to perform the floating point operations below. A destination operand in extended floating point format is stored in the applications globals area as DesExtended. Source operands are stored as DoubleSource (64-bit floating point), SingleSource (32-bit floating point), IntSource (integer), LongintSource (long integer) and CompSource (computational).
   a. add a double-precision source operand to the destination operand
   b. multiply an integer source operand by the destination operand
   c. round the destination operand to an integer
   d. invert the sign of the destination operand
   e. find the cosine of the destination operand
   f. generate a random number

9. Write a block of code that compares two operands in extended floating point format and then puts a pointer to the larger operand in A0 and a pointer to the smaller operand in A1. Use the macros defined in SANEMacs.Txt and allocate any necessary data structures.

10. Write a block of code that will create a format record for a binary-to-decimal conversion that will produce a floating point number with eight significant digits.
11. Write pseudocode that summarizes the procedure for doing a binary to decimal conversion, assuming that you are starting with a string of ASCII characters.

12. Using the subroutines Parser and FormatFloat, write a subroutine that:
   A. accepts a pointer to a source operand string in A0 and a pointer to a destination operand string in A1;
   B. converts both operands to the extended floating point format;
   C. subtracts the source operand from the destination operand; and
   D. returns the result as a floating point number properly formatted for output
The Video Tape Index Program is a specialized database program that maintains listings and annotations of tapes. It can also be used to handle, for example, audio tapes, records, and video discs.

To run the program, double-click on its icon from the Finder. It will automatically create and open any necessary files. You may then select from the Options menu to update the database (entering, changing, deleting), do on-screen data display, or print the database.

The program does have some limitations. Those limitations and suggested modifications to overcome them are:

1. The program will support up to 100 titles. To increase this, increase the size of the TapeArray storage area. A 512K Mac will support as many as 500 titles, though if that much memory is used for the RAM array, there may not be enough left to image a spooled print file. In that case, you must always draft print.

2. Search hits are based only on equality of the search data with data in TapeArray. To implement searches on equalities, allow the user to enter a symbol for the inequality (e.g., <, > =, etc.) at the beginning of the search text. Then add code to the binary and sequential search routines that identifies the search criteria and, based on that criteria, makes the appropriate comparisons.

3. All TapeArray data are stored in a file named TAPE.MASTER. All annotations are stored in ANNOTATIONS. To allow the user to have multiple sets of master and annotation files, use the standard "get file" dialog box twice, once to select a tape file and once to select an annotation file.

4. The program will print only the entire file, either with or without annotations. To print only selected records, allow the user to enter selection criteria and verify each record against that criteria before printing.
5. The program will print only in tape-name order. It can be easily sorted to change that sequencing. Since the file must be maintained in tape-name order for the binary search to work, it is probably best to sort the array to a copy in RAM. The same straight-insertion sort that inserts new records can be used for that purpose.

6. Deleting a record from the tape master file does not delete its annotation. Deleting annotations requires a routine that completely re-writes the annotation file. It should read sequentially through TapeArray, retrieving each record's annotation and writing it out to the new annotation file.

Listing A.1 Source Code of the Video Tape Index Program

#include MaoTraps.D ;includes addresses of Toolbox routines
#include ToolEqu.D ;includes the Toolbox equates
#include SysEqu.D ;includes the System equates
#include QuickEqu.D ;the QuickDraw equates
#include PrEqu.Txt ;printer equates

;----------------------------- EQUATES -----------------------------
;------- (must go at the top or program won't assemble!) -------

oTapeName EQU .0' ;offsets in tape record
oProducer EQU 30
oReleaseDate EQU 50
oRating EQU 54
oTapeNumber EQU 58
oAnnotNum EQU 62

; --------------------- Initialize managers ---------------------
PEA -4(A5)
_InitGraf ;Initializes QuickDraw
_InitFonts ;Initializes the Font Manager
_InitWindows ;Initializes the Window Manager
_InitMenus ;Initializes the Menu Manager
CLR.L -(SP) ;no restart procedure
_InitDialogs ;initializes dialog manager
_TEInit ;Initializes Text Edit

;This section gets all eight menus from the resource file and makes them
;available to the program through their handles
CLR.L -(SP) ;Clear space for menu handle
MOVE #1, -(SP) ;This will be menu 1
_GetRMenu ;Apple menu comes in from resource file

LEA AppleHandle, A0 ;Get address for handle
MOVE.L (SP)+, A1 ;Pull handle off stack
MOVE.L A1,(A0) ;Store handle

(continued)
Listing A.1 (continued)

LEA AppleHandle,A1
MOVE.L (A1),(SP) ;Put handle back on stack
MOVE.L *DRVR-(SP) ;Identify desk accessories
…AddResMenu ;Get desk accessories from system

CLR.L -(SP) ;Clear space for handle
MOVE #2,-(SP) ;menu #2
…GetRMenu ;Edit menu

LEA EditHandle,AB ;Get address for handle
MOVE.L (SP)+,A1 ;Pull handle off stack
MOVE.L A1,(AB) ;Store handle

CLR.L -(SP) ;Clear space for handle
MOVE #3,-(SP) ;menu #3
…GetRMenu ;Options menu

LEA OptionsHandle,AB ;Get address for handle
MOVE.L (SP)+,A1 ;Pull handle off stack
MOVE.L A1,(AB) ;Store handle

CLR.L -(SP) ;Clear space for handle
MOVE #4,-(SP) ;Enter menu
…GetRMenu ;Change menu

LEA EnterHandle,AB
MOVE.L (SP)+,A1
MOVE.L A1,(AB)

CLR.L -(SP)
MOVE #5,-(SP)
…GetRMenu ;Delete menu

LEA ChangeHandle,AB
MOVE.L (SP)+,A1
MOVE.L A1,(AB)

CLR.L -(SP)
MOVE #6,-(SP)
…GetRMenu ;Select menu

LEA DeleteHandle,AB
MOVE.L (SP)+,A1
MOVE.L A1,(AB)

CLR.L -(SP)
MOVE #7,-(SP)
…GetRMenu ;Select menu

LEA SelectHandle,AB
MOVE.L (SP)+,A1
MOVE.L A1,(AB)

(continued)
; This section gets the seven windows from the resource file and allocates
; their storage. They are invisible at this point.

CLR.L -(SP) ; space for window handle
MOVE *8,-(SP) ; GetMenu

LEA PrintHandle, A0
MOVEL (SP)+, A1
MOVEL A1, (A0)

; Annotation window
PEA AnnotationWindowStorage(A5) ; address for window record
MOVEL *-1,-(SP) ; put window in front

; Tape Number window
PEA TapeNumberWindowStorage(A5) ; address for window record
MOVEL *-1,-(SP) ; put window in front

; Number window
PEA NumberWindowStorage(A5) ; address for window record
MOVEL *-1,-(SP) ; put window in front

; Rating window
PEA RatingWindowStorage(A5)
MOVEL *-1,-(SP)

; Date window
PEA DateWindowStorage(A5)
MOVEL *-1,-(SP)

(continued)
Listing A.1 (continued)

MOVE  #3,-(SP) ;Producer window
LEA  ProducerWindowStorage(A5)
MOVE.L  #1,-(SP)
...GetNewWindow

LEA  ProducerWindowPtr,A0
MOVE.L  (SP)+,A1
MOVE.L  A1,(A0)

CLR.L  -(SP)
MOVE  #2,-(SP) ;Tape Name window
LEA  NameWindowStorage(A5)
MOVE.L  #1,-(SP)
...GetNewWindow

LEA  NameWindowPtr,A0
MOVE.L  (SP)+,A1
MOVE.L  A1,(A0)

CLR.L  -(SP)
MOVE  #1,-(SP) ;this is window #1
LEA  MainWindowStorage(A5) ;address for window record storage
MOVE.L  #1,-(SP)
...GetNewWindow ;get window definition from resource file

LEA  MainWindowPtr,A0 ;load destination address for pointer
MOVE.L  (SP)+,A1 ;get pointer from stack
MOVE.L  A1,(A0) ;put pointer into WindowPtr

;----------------- Allocate TextEdit Records -----------------------------

MOVE.L  NameWindowPtr,-(SP)
...SetPort
CLR.L  -(SP) ;clear space for text handle
LEA  NameDestRect
LEA  NameViewRect
...TENew ;allocate text record
LEA  NameTextHandle,A0 ;get address for text handle
MOVE.L  (SP)+,(A0) ;take handle from stack and store

MOVE.L  ProducerWindowPtr,-(SP)
...SetPort
CLR.L  -(SP)
LEA  ProducerDestRect
LEA  ProducerViewRect
...TENew
LEA  ProducerTextHandle,A0
MOVE.L  (SP)+,(A0)

MOVE.L  DateWindowPtr,-(SP)
...SetPort
CLR.L  -(SP)
LEA  DateDestRect

(continued)
PEA DateViewRect
_TENew
LEA DateTextHandle,AØ
MOVE.L (SP)+,(AØ)

MOVE.L RatingWindowPtr,-(SP)
_SetPort
CLR.L -(SP)
PEA RatingDestRect
PEA RatingViewRect
_TENew
LEA RatingTextHandle,AØ
MOVE.L (SP)+,(AØ)

MOVE.L NumberWindowPtr,-(SP)
_SetPort
CLR.L -(SP)
PEA NumberDestRect
PEA NumberViewRect
_TENew
LEA NumberTextHandle,AØ
MOVE.L (SP)+,(AØ)

MOVE.L AnnotationWindowPtr,-(SP)
_SetPort
CLR.L -(SP)
PEA AnnotationDestRect
PEA AnnotationViewRect
_TENew
LEA AnnotationTextHandle,AØ
MOVE.L (SP)+,(AØ)

;------------------------ Change cursor to watch for file operations ------------------------
CLR.L -(SP) ;space for cursor handle result
MOVE #4,-(SP) ;indicates the watch cursor for long wait
_GetCursor ;get handle to cursor definition

MOVE.L (SP)+,AØ
MOVE.L (AØ),AØ ;de-reference the handle to get pointer
MOVE.L AØ,-(SP) ;put pointer on stack
_SetCursor ;set cursor to watch

;------------------------ Load TapeArray or create new file ------------------------
LEA 'Tape.Master',AØ ;file name
MOVE.L AØ,ioParamBlock+ioFileName(A5)
MOVE #1,ioParamBlock+ioDrvNum(A5) ;on drive 1
MOVE.B #0,ioParamBlock+ioFileType(A5) ;version number 0
LEA ioParamBlock(A5),AØ ;Create
_Create ;attempt to create file

CMP #-48,DØ ;duplicate file name
BEQ OpenTapeFile
Listing A.1 (continued)

```assembly
CMP   #0,0
BEQ   TapeFileInfo
JMP   FileError

TapeFileInfo
LEA   'Tape.Master',A0
MOVE.L A0,fiParamBlock+ioFileName(A5) ;name
MOVE   #1,fiParamBlock+ioDrvNum(A5) ;drive
MOVE.B #0,fiParamBlock+ioFileType(A5) ;version
MOVE   #0,fiParamBlock+ioDirIndex(A5) ;use name and drive to find file
LEA   fiParamBlock(A5),A0
...GetFileInfo

MOVE.L '#TEXT',fiParamBlock+ioFlUsrWds(A5) ;file type
LEA   fiParamBlock(A5),A0
...SetFileInfo

LEA   TotalRecords,A0
MOVE   #0,(A0)
BRA   CloseTapeFile

OpenTapeFile
LEA   'Tape Master',A0
MOVE.L A0,ioParamBlock+ioFileName(A5)
MOVE   #1,ioParamBlock+ioDrvNum(A5)
MOVE.B #0,ioParamBlock+ioFileType(A5)
MOVE.B #1,ioParamBlock+ioPermssn(A5) ;read only permission
CLR.L ioParamBlock+ioOwnBuf(A5)
LEA   ioParamBlock(A5),A0
...Open

CMP   #0,0
BNE   FileError

LEA   DataBuffer(A5),A0
MOVE.L A0,ioParamBlock+ioBuffer(A5)
MOVE.L #4,ioParamBlock+ioByteCount(A5) ;just get tape & annot. totals
MOVE   #0,ioParamBlock+ioPosMode(A5) ;read from mark
LEA   ioParamBlock(A5),A0
...Read

MOVE.L DataBuffer(A5),DB ;get numbers just read
LEA   LastAnnotNumb,A0
MOVE   DB,(A0) ;recover last annotation number
SWAP   DB
LEA   TotalRecords,A0
MOVE   DB,(A0) ;put total records in lower half
LEA   TapeArray(A5),A0 ;destination for tape records
MOVE.L A0,ioParamBlock+ioBuffer(A5)
MOVE   TotalRecords,DB ;recover total records
MJULU   #64,0
...number of bytes to read
```

(continued)
MOVE.L DB, ioParamBlock+ioByteCount(A5)
MOVE #B, ioParamBlock+ioPosMode(A5) ; read from mark
LEA ioParamBlock(A5), AB
_Read

CloseTapeFile
LEA ioParamBlock(A5), AB
_Close

;----------------------- Open Annotations file or create new file -----------------------

LEA 'Annotations', AB
MOVE.L AB, ioParamBlock+ioFileName(A5) ; file name
MOVE #1, ioParamBlock+ioDrvNum(A5) ; on drive 1
MOVE.B #0, ioParamBlock+ioFileType(A5) ; version number of B
LEA ioParamBlock(A5), AB ; point to parameter block
_Create

CMP #-4B, DB ; duplicate file name
BEQ OpenAnnotFile
CMP #0, DB ; file successfully created
BEQ AnnotFileInfo
JMP FileError

AnnotFileInfo
LEA 'Annotations', AB
MOVE.L AB, fiParamBlock+ioFileName(A5)
MOVE #1, fiParamBlock+ioDrvNum(A5)
MOVE #0, fiParamBlock+ioFileType(A5)
MOVE #0, fiParamBlock+ioFDirIndex(A5)
LEA fiParamBlock(A5), AB
_GetFileInfo

MOVE.L '#TEXT', fiParamBlock+ioF1UsrWds(A5)
LEA fiParamBlock(A5), AB
_SetFileInfo

LEA LastAnnotNumb, AB
MOVE #-1, (A.B)

OpenAnnotFile
LEA 'Annotations', AB
MOVE.L AB, ioParamBlock+ioFileName(A5) ; load file name
MOVE #1, ioParamBlock+ioDrvNum(A5) ; load drive number
MOVE.B #0, ioParamBlock+ioFileType(A5) ; a version number of B
MOVE.B #3, ioParamBlock+ioPermssn(A5) ; allow reading and writing
CLR.L ioParamBlock+ioOwnBuf(A5) ; use volume access path buffer
LEA ioParamBlock(A5), AB ; point to parameter block
_Open

CMP #B, DB ; result code in DB
BNE FileError

(continued)
Listing A.1 (continued)

LEA fiRefNum, A0
MOVE ioParamBlock+ioRefNum(A5),(A0) ;save reference number
BRA BeginProgram ;file open and all's well

FileError
CLR -(SP)
MOVE #6, -(SP)
CLR.L -(SP)
 ALERT
MOVE (SP)+, D0

RTS ;returns to Finder

;---------------------- Make main window visible and bring to front ----------------------
BeginProgram

MOVE.L MainWindowPtr, -(SP)
.SelectWindow

MOVE.L MainWindowPtr, -(SP)
 SetPort
_InitCursor ;set the cursor to the arrow

MOVE.L everyEvent, D0 ;Mask to select all events
.FlushEvents ;Clear the event queue

JSR MainMenuBar ;Set up and draw main menu bar

;------------- Main Event Loop ------------------------
Event .SystemTask ;update desk accessories

CLR -(SP) ;Space for boolean result
MOVE #1, -(SP) ;Mask for keyboard - select all events
PEA EventRecord ;Place to receive event info
.GetNextEvent ;Get next event from queue

MOVE (SP)+, D0 ;Recover event result
CMP *D0
BEQ Event ;If no event, branch to look again

MOVE What,D0 ;Recover event ID
CMP *mButDwnEvt,D0 ;Was mouse button pressed?
BEQ MouseEvent

CMP *keyDwnEvt,D0 ;Was key pressed?
BEQ KeyEvent

BRA Event ;Look for another event

(continued)
KeyEvent

```
MOVE.B Modify, D0 ;Recover modifier bytes
CMP.B #$B1, D0 ;Was command key pressed
BEQ KeyboardCommand
BRA Event
```

KeyboardCommand

```
CLR.L -(SP) ;space for menu item selection
MOVE Message+2, -(SP) ;put character pressed on stack
(MenuKey) ;figure out what key was pressed
BRA Selections
```

MouseEvent

```
CLR -(SP) ;Place for “what” result
MOVE.L Point, -(SP) ;Point = mouse coordinates
PEA WhichWindowPtr ;Push place for window handle of window
(FindWindow) ;Where was button pushed?
MOVE (SP)+, D0 ;Recover FindWindow result
CMP *inMenuBar, D0 ;Was mouse clicked in menu bar?
BEQ MenuBar ;Mouse clicked in menu bar
CMP *inSysWindow, D0 ;Was mouse clicked in system window?
BEQ SysEvent ;Mouse clicked in system window
BRA Event ;go back to check for another event
```

SysEvent

```
PEA EventRecord ;Event record goes on stack
MOVE.L WhichWindowPtr, -(SP) ;Window pointer goes on stack, too
(SystemClick) ;System handles it
BRA Event
```

MenuBar

```
CLR.L -(SP) ;Place for menu ID and Menu item
MOVE.L Point, -(SP) ;Push mouse coordinates
(MenuSelect) ;Find out exactly where mouse was clicked
```

Selections

```
MOVE.L (SP)+, D2 ;Recover result
LEA WhichMenu, A0 ;get address for high-order byte of result
MOVE.L D2, (A0) ;Store result
CLR -(SP) ;get set to unhighlight all menus
(HiliteMenu) ;Unhighlight the menus
MOVE WhichMenu, D0 ;Put menu number in D0
CMP #1, D0 ;In apple menu?
```
Listing A.1 (continued)

BNE Menu2
JSR AppleMenu

Menu2 CMP #2,D.0 ;In edit menu?
BNE Menu3
JSR EditMenu

Menu3 CMP #3,D.0 ;In options menu?
BNE NoMenu
JSR Options

NoMenu BRA Event ;Return to look for another event

AppleMenu
LEA AppleHandle,A.0 ;Get address of menu handle
MOVE.L (A.0),-(SP) ;Put menu handle of stack
MOVE WhatItem,D.0
MOVE D.0,-(SP) ;Put ID# of item clicked on stack
PEA DeskAccName ;Put address where desk acc. name should go
.GETItem ;Figure out which one was selected
CLR -(SP) ;Leave space for reference number
PEA DeskAccName ;Put address of name on stack
_OpenDeskAcc ;Open the desk accessory
MOVE (SP)+,D.0 ;Pull reference number off stack
CLR -(SP) ;Unhighlight the menu title
/appleMenu
MOVE #9,D.0
RTS

EditMenu
MOVE WhatItem,D.0 ;Figure out which command is selected
SUBQ #1,D.0 ;Adjust number to pass to SysEdit
CLR -(SP) ;Space for result if there's a problem
MOVE D.0,-(SP) ;Let system know what item was chosen
_SYSEdit ;Let the system handle the edit
MOVE (SP)+,D.1 ;Clear problem result from stack
MOVE #9,D.0
RTS

Options MOVE WhatItem,D.0 ;Move item selected to D.0

CMP #1,D.0
BNE Item2
JSR Enter ;Enter new tapes

Item2 CMP #2,D.0
BNE Item3
JSR Change ;Modify existing tapes

(continued)
CMP #3, DH
BNE Item4
JSR Delete

Item4
CMP #4, DH
BNE Item5
JSR Select

Item5
CMP #5, DH
BNE Item6
JSR Print

Item6
CMP #6, DH
BEQ Quit
JSR Delete ;Delete tapes

JSR Select ;Retrieve info
CMPL #5, oH
BNE Item6
JSR Print ;Print lists

CMP #6, DH
BEQ Quit ;Exit the program

MOVE L MainWindowPtr, -(SP)
..SelectWindow

MOVE L MainWindowPtr, -(SP)
..SetPort

PEA MainWindowRect
..EraseRect ;clears out text window prompts

JSR ReDrawMainMenu ;put options menu back in menu bar

MOVE L EditHandle, -(SP)
MOVE *t, -(SP) ;"UnDo"
..Enable Item ;highlight "UnDo", since systems windows use it

MOVE L MainWindowPtr, -(SP)
PEA 'Video Tape Index'
..SetTitle

RTS
..Return to main program

Quit
CLR L -(SP) ;space for cursor handle
MOVE #4, -(SP) ;ID for watch cursor

..GetCursor
MOVE (SP)+, AØ ;recover handle
MOVE (AØ), AØ ;de-reference handle to get pointer
MOVE L AØ, -(SP) ;pointer to cursor definition
..SetCursor ;set watch cursor for file operations

MOVE fiRefNum, ioParamBlock+ioRefNum(A5)
LEA ioParamBlock(A5), AØ
..Close ;close the annotations file

LEA 'Tape.Master', AØ
MOVE L AØ, ioParamBlock+ioFileName(A5)
MOVE *1, ioParamBlock+ioDrvNum(A5)

(continued)
Listing A.1 (continued)

```
MOVE.B *$0, ioParamBlock+ioFileType(A5)  
MOVE.B *$2, ioParamBlock+ioPermssn(A5)  ;write only permission
CLR.L ioParamBlock+ioOwnBuf(A5)  
LEA ioParamBlock(A5), A5

CMP *$0, D0
BNE FileError

MOVE TotalRecords, D0  ;put total records in high order bits
SWAP D0
AND.L *$00000000, D0  ;clear out low order bits

MOVE LastAnnotNumb, D0

MOVE L, ioPermssn(A5)

MOVE ioParamBlock(A5), A5

MOV L ioParamBlock(A5), A5  ;tape array location doubles as buffer

MOVE A5, ioParamBlock+ioBuffer(A5)

MOVE TotalRecords, D0  ;total number of bytes to move

MULU *$64, D0

MOVE L, ioPermssn(A5)

MOVE ioParamBlock+ioPosMode(A5)  ;sequential write

LEA ioParamBlock(A5), A5

_Write

LEA TapeArray(A5), A5  ;tape array location doubles as buffer

MOVE A5, ioParamBlock+ioBuffer(A5)

MOVE TotalRecords, D0

MULU *$64, D0

MOVE L, ioPermssn(A5)

MOVE ioParamBlock+ioPosMode(A5)  ;sequential write

LEA ioParamBlock(A5), A5

_Write

_CLOSE

_Close  ;close the tape master file

_InitCursor  ;re-set to arrow cursor

MOVE (SP)+, D0  ;pop subroutine return address off stack

RTS  ;This return goes back to the Finder
```

;------------------------- Enter New Titles -------------------------

Enter

MOVE L MainWindowPtr, -(SP)

PEA 'Enter New Titles and Annotations'

_SetWindowTitle

J sr DisplayPrompts  ;make text window prompts visible

J sr DisplayWindows  ;make the text entry windows visible

MOVE *$3, -(SP)

_DeleteMenu  ;Remove Options menu from menu list

LEA EnterHandle, A5  ;get address for Enter menu’s handle

MOVE A5, -(SP)  ;put handle on stack

(continued)
CLR -(SP) ;this menu will go at the end of the list
_InsertMenu

_DrawMenuBar ;Re-draw the menu bar

MOVE L EditHandle,-(SP)
MOVE #1,-(SP) ;"UnDo" is not supported
_DisableItem ;make "UnDo" appear dimmed

_EnterEvent
MOVE L ActiveTextHandle,-(SP)
_SystemTask ;update desk accessories
_CLR -(SP) ;space for boolean result
MOVE *,-(SP) ;mask to select all events
PEA EventRecord ;place to accept event
_GetNextEvent ;get next event from queue

MOVE (SP)+,DØ ;recover event result
CMP #B, DØ
_BEQ EnterEvent ;no event encountered - keep checking

MOVE What,DØ ;recover event ID
CMP #mButDwnEvt,DØ ;mouse button pressed?
_BEQ EnterMouseEvent

CMP #keyDwnEvt,DØ ;was key pressed?
_BEQ EnterKeyEvent

CMP #activateEvt,DØ ;activate event posted?
_BEQ EnterActivateEvent

CMP #updateEvt,DØ ;text window needs updating?
_BEQ EnterUpdateEvent

_BRA EnterEvent ;look for another event

_EnterActivateEvent
JSR ActivateTextWindow

BRA EnterEvent

_EnterUpdateEvent
JSR UpdateTextWindows

BRA EnterEvent

_EnterKeyEvent
MOVE B Modify,DØ ;recover modifier byte

(continued)
Listing A.1 (continued)

CMP.B #$01,D0 ;was command key pressed?
BEQ EnterKeyboardCommand

MOVE Message+2,-(SP) ;character that was pressed
MOVE.L ActiveTextHandle,-(SP) ;move handle to current text record
_TEKey ;insert the character

BRA EnterEvent

EnterKeyboardCommand

CLR.L -(SP) ;place for menu item selection
MOVE Message+2,-(SP) ;put character pressed on stack
_MENUKey
BRA EnterSelections

EnterMouseEvent

CLR -(SP) ;space for "what" result
MOVE Point,-(SP) ;put mouse coordinates on stack
PEA WhichWindowPtr ;push pointer to window record
_FindWindow ;where was button pressed?

MOVE (SP)+,D0 ;recover FindWindow result

CMP *inMenuBar,D0 ;was mouse clicked in menu bar?
BEQ EnterMenuBar ;mouse clicked in menu bar

CMP *inSysWindow,D0 ;was mouse clicked in a desk accessory?
BEQ EnterSysEvent

CMP *inContent,D0 ;mouse clicked in content area of user window?
BEQ EnterInWindow

BRA EnterEvent

EnterSysEvent

PEA EventRecord ;pointer to event record goes on stack
MOVE.L WhichWindowPtr,-(SP) ;window pointer on stack, too
_SystemClick ;let the system handle it

BRA EnterEvent

EnterMenuBar

CLR.L -(SP) ;place for menu ID and menu item
MOVE.L Point,-(SP) ;push mouse coordinates
_MENUSelect ;which menu?

EnterSelections

MOVE.L (SP)+,D2 ;recover result

LEA WhichMenu,A0 ;get address for high-order byte of result
MOVE.L D2,(A0) ;store result

(continued)
CLR -(SP) ;mask to indicate all menus
.HILiteMenu ;unhighlight the menus

MOVE WhichMenu,DØ ;put menu number in DØ

CMP #1,DØ ;in Apple menu?
BNE EnterMenu2
JSR AppleMenu

EnterMenu2

CMP #2,DØ ;in edit menu?
BNE EnterMenu4
JSR EditMenu ;was edit command in system window?
CMP *Ø,D1
BNE EnterEvent ;edit was in system window and system handled it
JSR DoEditing
BRA EnterEvent

EnterMenu4

CMP #4,DØ
BNE EnterEvent

EnterMenuOptions

MOVE WhatItem,DØ ;Move item selected to DØ

CMP #1,DØ ;Add a new item?
BNE OtherOne
JSR AddNewTitle

OtherOne

CMP #2,DØ ;Quit?
BNE EnterEvent ;not what we want - look for another event

MOVE #4 -(SP)
.DeleteMenu ;remove enter menu

MOVE #9,DØ ;return to options block
RTS

EnterInWindow

CLR L -(SP) ;make room for pointer as result
.FRontWindow ;find out which window is in front (i.e., active)

MOVE L (SP)+,AØ ;recover FrontWindow result

CMP L WhichWindowPtr,AØ ;is front window same as clicked window?
BNE MustActivate ;window is inactive

PEA Point ;place where mouse button was clicked
_GlobalToLocal ;convert coordinates to local system

(continued)
Listing A.1 (continued)

MOVE.L Point,-(SP) ;coordinates now local
BTST  #shiftKey,Modify ;shift key bit set?
SNE   D0 ;set true if shift key was held down
MOVE.B D0,-(SP) ;moving byte puts boolean in high order byte
MOVE.L ActiveTextHandle,-(SP) ;this will be currently active window
_TEClock

BRA    EnterEvent

MustActivate
JSR    SelectTextWindow
BRA    EnterEvent

AddNewTitle

; ---------------- assemble an input record -------------------------

JSR    ClearNewRecord
JSR    MoveName
JSR    MoveProducer
JSR    MoveDate
JSR    MoveRating
JSR    MoveNumber

MOVE    LastAnnotNumb,D0
ADDQ    #1,D0
MOVE    D0,NewRecord+AnnotNum(A5)
LEA     LastAnnotNumb,A0
MOVE    D0,(A0)

;----------------- Straight-Insertion Sort -------------------

MOVE    TotalRecords,D1
LEA     TapeArray(A5),A2
CMP     #8,D1
BEQ     InsertNew ;if first record, insert immediately
SUBQ    #1,D1 ;otherwise, adjust for record # 0's beginning with 0

Checking

JSR    ComputeAddress1 ;Address returned in A3
MOVE.L D1,-(SP) ;save D1 on stack
CLR.W   -(SP) ;space for result
MOVE.L A3,-(SP) ;pointer to record in array
PEA     NewRecord(A5) ;pointer to new record
MOVE.W #50,-(SP) ;characters to look at in first string
MOVE.W #30,-(SP) ;characters to look at in second string
MOVE.W #10,-(SP) ;ID for IUMagString
_Pack6
MOVE.W (SP)+,D0 ;recover result
MOVE.L (SP)+,D1 ;recover former contents of D1

CMP     #8,D0
BLE     JustBeforeInsert ;found place to insert record
BGT     MoveOld ;move existing record down

(continued)
MoveOld

MOVE  D1,D5
ADDQ  #1,D5           ;record # to move to
JSR   ComputeAddress1 ;offset returned in A3
JSR   ComputeAddress2 ;offset returned in A4

MOVEL A3,A0          ;source pointer for block move
MOVEL A4,A1          ;destination pointer for block move
MOVEL #64,D0         ;64 bytes will be moved
 BlockMove             ;move an entire record

SUBQ  #1,D1           ;move back a record
CMP   #1,D1
BEQ   JustBeforeInsert
BRA   Checking

JustBeforeInsert
ADDQ  #1,D1           ;insert just below where comparing

InsertNew

MOVE  D1,D5
JSR   ComputeAddress2

LEA   NewRecord(A5),A0         ;pointer to source (the new record)
MOVEL A4,A1                   ;pointer to destination
MOVEL #64,D0                   ;number of bytes to move
 BlockMove                     ;move a record

LEA   TotalRecords,A0          ;increment number of records
ADDQ  #1,(A0)                 ;increment number of records

;----------------- Write the annotation directly to the Annotation file -------

LEA   AnnotRecMask,A0
LEA   DataBuffer(A5),A1
MOVEL #256,D0                 ;fill first half of buffer with blanks

CLRL   -(SP)                  ;place for CharsHandle result
MOVEL AnnotationTextHandle,-(SP)
_TGetText ;get handle to text in Annotation record
MOVEL (SP)+,A2                ;recover CharsHandle
MOVEL (A2),A0                 ;de-referencing handle to get pointer
LEA   DataBuffer(A5),A1       ;text goes into disk buffer
MOVEL AnnotationTextHandle,A3
MOVEL (A3),A4                 ;de-reference again
MOVE   telLength(A4),D0       ;number of characters to move
 BlockMove                     ;puts annotation in disk output buffer

MOVE   #256,D0                ;characters per annotation record
MULU   LastAnnotNumb,D0       ;offset into annotations file

(continued)
Listing A.1 (continued)

```assembly
LEA DataBuffer(A5),A0
MOVE.L A0,ioParamBlock+ioBuffer(A5)
MOVE.L *256,ioParamBlock+ioByteCount(A5) ;write 256 bytes, blanks and all
MOVE *1,ioParamBlock+ioPosMode(A5) ;offset is relative to beginning of file
MOVE.D 0,ioParamBlock+ioPosOffset(A5) ;offset in bytes
MOVE fiRefNum,ioParamBlock+ioRefNum(A5) ;file reference number
LEA ioParamBlock(A5),A0
_Write

JSR DisplayWindows ;clear windows and text edit records
RTS ;return

ComputeAddress1
  MOVE.L D1,D6 ;offset = record # * 64 bytes
  MULU *64,D6
  MOVE.L A2,A3
  ADDA.L D6,A3
  RTS

ComputeAddress2
  MOVE.L D5,D7
  MULU *64,D7
  MOVE.L A2,A4
  ADDA.L D7,A4
  RTS

;------------------------ Change Existing Data ------------------------

Change MOVE.L MainWindowPtr,-(SP)
PDA 'Change Existing Titles and Annotations'
  _SetTitle

  JSR DisplayPrompts
  JSR DisplayWindows

  MOVE *3,-(SP) ;remove Options menu from menu list
  _DeleteMenu

LEA ChangeHandle,A0;get address for Change menu's handle
MOVE.L (A0),-(SP) ;put handle on stack
CLR -(SP) ;this menu goes at the end
  _InsertMenu

  _DrawMenuBar

MOVE.L EditHandle,-(SP)
MOVE *1,-(SP)
  _DisableItem
```

(continued)
ChangeEvent

MOVE.L ActiveTextHandle, -(SP)
_TEIdle

_SystemTask ; update desk accessories
CLR -(SP) ; space for boolean result
MOVE #1,-(SP) ; mask to select all events
PEA EventRecord ; place to accept event
_GetNextEvent ; get an event from the queue

MOVE (SP)+,D0 ; recover event record
CMP *g,D0
BEQ ChangeEvent ; no event - keep looking

MOVE what,D0 ; recover event ID
CMP #mButDwnEvt,D0 ; mouse button pressed?
BEQ ChangeMouseEvent

CMP #keyDwnEvt,D0 ; key pressed?
BEQ ChangeKeyEvent

CMP #activateEvt,D0 ; activate event posted?
BEQ ChangeActivateEvent

CMP #updateEvt,D0 ; text window needs updating?
BEQ ChangeUpdateEvent

BRA ChangeEvent

ChangeActivateEvent

JSR ActivateTextWindow
BRA ChangeEvent

ChangeUpdateEvent

JSR UpdateTextWindows
BRA ChangeEvent

ChangeKeyEvent

MOVE.B Modify,D0 ; recover modifier byte
CMP.B #1,D0 ; was command key pressed?
BEQ ChangeKeyboardCommand

MOVE Message+2,-(SP)
MOVE.L ActiveTextHandle, -(SP)
_TEKey

BRA ChangeEvent

ChangeKeyboardCommand

CLR.L -(SP) ; place for menu item selection

(continued)
Listing A.1 (continued)

MOVE Message+2,-(SP); put character pressed on stack
  _MenuKey
BRA ChangeSelections

ChangeMouseEvent

CLR -(SP); space for "What" result
MOVE.L Point,-(SP); put mouse coordinates on stack
PEA WhichWindowPtr; push pointer to window record
  _FindWindow; where was mouse button pushed?

MOVE (SP)+,D0; recover FindWindow result

CMP #inMenuBar,D0; was mouse clicked in menu bar?
BEQ ChangeMenuBar

CMP #inSysWindow,D0; was mouse clicked in a desk accessory?
BEQ ChangeSysEvent

CMP #inContent,D0
BEQ ChangeInWindow

BRA ChangeEvent

ChangeSysEvent

PEA EventRecord; pointer to event record goes on stack
MOVE.L WhichWindowPtr,-(SP); window pointer goes on stack, too
  _SystemClick; let the system handle it

BRA ChangeEvent

ChangeMenuBar

CLR.L -(SP); place for menu ID and menu item
MOVE.L Point,-(SP); push mouse coordinates
  _MenuSelect; which menu? which item?

ChangeSelections

MOVE.L (SP)+,D2; recover result

LEA WhichMenu,A0; get address for high order byte of result
MOVE.L D2,(A0); store result

CLR -(SP); mask to indicate all menus
  _HiLiteMenu; unhighlight all menus

MOVE WhichMenu,D0; put menu number in D0

CMP #1,D0; in Apple menu?
BNF ChangeMenu2
JSR AppleMenu

(continued)
ChangeMenu2
  CMP  *2,00
  BNE  ChangeMenu5 ;in Edit menu?
  JSR  EditMenu
  CMP  *0,01
  BNE  ChangeEvent ;was edit request in system window?
  JSR  DoEditing
  BRA  ChangeEvent

ChangeMenu5
  CMP  *5,00
  BNE  ChangeEvent ;in Change menu?

ChangeMenuOptions
  MOVE  WhatItem,00
  MOVE  *1,00
  BNE  ChangeItem2 ;find a record?
  MOVE  *1,ReturnFlag(A5) ;set return flag to show origin of call
  JSR  SelectOneTitle ;note: record number returned in RecordCounter(A5)
  BRA  ChangeEvent

ChangeItem2
  CMP  *2,00
  BNE  ChangeItem3 ;save a change?
  JSR  ChangeSave
  BRA  ChangeEvent

ChangeItem3
  CMP  *3,00
  BNE  ChangeItem4 ;abandon a change?
  JSR  DisplayWindows ;clear text windows
  BRA  ChangeEvent

ChangeItem4
  CMP  *4,00
  BNE  ChangeEvent ;quit?
  MOVE  #5,-(SP)
  _DeleteMenu ;remove Change menu
  MOVE  #9,00
  RTS ;return to options block

ChangeInWindow
  CLR.L -(SP)
  _FrontWindow
  MOVE.L (SP)+,A0
  CMP.L WhichWindowPtr,A0

(continued)
Listing A.1 (continued)

BNE ChangeMustActivate ;window is inactive
PEA Point
  _GlobalToLocal

MOVE.L Point,-(SP)
BTST *shiftkey,Modify
SNE D0
MOVE.B D0,-(SP)
MOVE.L ActiveTextHandle,-(SP)
  _TECNext ;re-position the cursor

BRA ChangeEvent

ChangeMustActivate
JSR SelectTextWindow
BRA ChangeEvent

ChangeSave

JSR ClearNewRecord
JSR MoveName
JSR MoveProducer
JSR MoveDate
JSR MoveRating
JSR MoveNumber

LEA TapeArray(A5),A2 ;start of Tape Array
LEA RecordCounter,A6
MOVE (A6),DS ;record number
JSR ComputeAddress2 ;get address of record - returned in A4

LEA NewRecord(A5),A6 ;source of data
MOVE.L A4,A1 ;destination of data
MOVE.L *62,D6 ;move only 62 bytes so annotation * isn't disturbed
  _BlockMove

`;------------------- re-write the annotation -------------------------------`
LEA AnnotRecMask,A6
LEA DataBuffer(A5),A1
MOVE *256,D6
  _BlockMove ;fill first half of buffer with blanks
CLR.L -(SP) ;place for CharsHandle result
MOVE.L AnnotationTextHandle,-(SP)
  _TEGetText ;get handle to text in Annotation record
MOVE.L (SP)+,A2 ;recover CharsHandle
MOVE.L (A2),A6 ;de-referencing handle to get pointer
LEA DataBuffer(A5),A1 ;text goes into disk buffer
MOVE.L AnnotationTextHandle,A3
MOVE.L (A3),A4 ;de-reference again

(continued)
MOYE teLength(A5),D0
_BLOCKMove

LEA RecordCounter,A0
MOVE (A0),D0
MULU #64,D0
ADD #oAnnotNum,D0 ;offset into tape array
LEA TapeArray(A5),A0
ADD L D5,A0
MOVE (A0),D0
MULU #256,D0 ;offset into file

LEA DataBuffer(A5),A0
MOVE.L A0,ioParamBlock+ioBuffer(A5)
MOVE.L #256,ioParamBlock+ioByteCount(A5) ;write 256 bytes, blanks and all
MOVE *1,ioParamBlock+ioPosMode(A5) ;offset is relative to beginning of file
MOVE D0,ioParamBlock+ioPosOffset(A5) ;offset in bytes
MOVE fileRefNum,ioParamBlock+ioRefNum(A5) ;file reference number
LEA ioParamBlock(A5),A0
_WRITE

JSR DisplayWindows

RTS

;----------------------------- Delete Titles -----------------------------

Delete

MOVE.L MainWindowPtr,-(SP)
PEA 'Delete Existing Titles'
_SetWTTitle

JSR DisplayPrompts
JSR DisplayWindows

MOVE *3,-(SP)
_DeleteMenu ;remove options menu from menu list

LEA DeleteHandle,A1 ;get address for Delete menu's handle
MOVE.L (A1),- (SP) ;put handle on stack
CLR -(SP) ;this menu will go at the end of the list
_InsertMenu ;put Delete menu into menu list

_DrawMenuBar ;re-draw the menu bar

MOVE.L EditHandle,-(SP)
MOVE *1,-(SP)
_DisableItem

DeleteEvent

MOVE.L ActiveTextHandle,-(SP)
_TEIdle

(continued)
Listing A.1 (continued)

setSystemTask ;update desk accessories

CLR -(SP)
MOVE #1,-(SP)
PEA EventRecord

-GetNextEvent ;get next event from queue

MOVE (SP)+,D0
CMP #0,D0
BEQ DeleteEvent ;no event encountered - keep looking

MOVE What,D0
CMP #mButDwnEvt,D0
BEQ DeleteMouseEvent ;mouse button pressed

CMP #keyDwnEvt,D0
BEQ DeleteKeyEvent ;key pressed

CMP #activateEvt,D0
BEQ DeleteActivateEvent

CMP #updateEvt,D0
BEQ DeleteUpdateEvent

BRA DeleteEvent ;look for another event

DeleteActivateEvent
    JSR ActivateTextWindow
    BRA DeleteEvent

DeleteUpdateEvent
    JSR UpdateTextWindows
    BRA DeleteEvent

DeleteKeyEvent
    MOVE.B Modify,D0
    CMP.B #1,D0
    BEQ DeleteKeyboardCommand ;command key was held down

    MOVE Message+2,-(SP)
    MOVE.L ActiveTextHandle,-(SP)
    _TEKey
    BRA DeleteEvent

DeleteKeyboardCommand
    CLR.L -(SP)
    MOVE Message+2,-(SP)
    _MenuKey ;figure out what key was pressed
    BRA DeleteSelections

(continued)
DeleteMouseEvent
CLR -(SP)
MOVE L Point,-(SP)
PEA WhichWindowPtr
.FindWindow ;where was mouse button pushed?
MOVE (SP)+,D0
CMP *inMenuBar,D0
BEQ DeleteMenuBar ;mouse button pushed in the menu bar
CMP *inSysWindow,D0
BEQ DeleteSysEvent ;mouse button pushed in desk accessory
CMP *inContentView,D0
BEQ DeleteContentView
BRA DeleteEvent

DeleteSysEvent
PEA EventRecord
MOVE L WhichWindowPtr,-(SP)
.SystemClick ;let the system handle it
BRA DeleteEvent

DeleteMenuBar
CLR L -(SP)
MOVE L Point,-(SP)
.MenuSelect ;which menu?

DeleteSelections
MOVE L (SP)+,D2
LEA WhichMenu,A0
MOVE L D2,(A0)
CLR -(SP)
.HiLiteMenu ;unhighlight the menus
MOVE WhichMenu,D0
CMP *1,D0 ;in Apple menu?
BNE DeleteMenu2
JSR AppleMenu

DeleteMenu2
CMP *2,D0 ;in Edit menu?
BNE DeleteMenu6
JSR EditMenu
CMP *0,D1

(continued)
Listing A.1 (continued)

BNE DeleteEvent ; system edit - has already been handled
JSR DoEditing
BRA DeleteEvent

DeleteMenu6
CMP #6, D0 ; in Delete menu?
BNE DeleteEvent ; get another event

DeleteMenuOptions
MOVE WhatItem, D0
CMP #1, D0 ; Find a title?
BNE DeleteOption2
MOVE #1, ReturnFlag(A5)
JSR SelectOneTitle
BRA DeleteEvent

DeleteOption2
CMP #2, D0 ; Delete a title?
BNE DeleteOption3
JSR DoTheDelete
BRA DeleteEvent

DeleteOption3
CMP #3, D0 ; Cancel a delete?
BNE DeleteOption4
JSR DisplayWindows
BRA DeleteEvent

DeleteOption4
CMP #4, D0 ; Quit?
BNE DeleteEvent

MOVE #6, -(SP)
_DeleteMenu ; remove Delete menu from menu list
MOVE #9, D0
RTS

DeleteInWindow
CLR.L -(SP)
_FrontWindow
MOVE.L (SP)+, A5
CMP.L WhichWindowPtr, A5
BNE DeleteMustActivate ; window is inactive

PEA Point
_GlobalToLocal

MOVE.L Point, -(SP)
BTST #shiftkey, Modify
SNE D0

(continued)
MOVE.B D8,-(SP)
MOVE.L ActiveTextHandle,-(SP)
annyHandle ;replication the cursor
BRA DeleteEvent

DeleteMustActivate
JSR SelectTextWindow
BRA DeleteEvent

DoTheDelete
LEA TapeArray(A5),A2 ;start of TapeArray
LEA RecordCounter,AB
MOVE (AB),D5 ;number of record to be deleted
ADDQ #1,D5 ;record number of source
JSR ComputeAddress2 ;get address of source
MOVE.L A4,AB

LEA RecordCounter,AB
MOVE (AB),D5
JSR ComputeAddress2 ;address of destination of move
MOVE.L A4,A1

MOVE TotalRecords,0
LEA RecordCounter,AB
SUB (AB),D0
MULU *64,D0 ;number of bytes to move
 rockingMove
LEA TotalRecords,AB
SUBQ #1,(AB)
JSR DisplayWindows

RTS

;------------------------ Select Titles -------------------------------------

Select MOVE.L MainWindowPtr,-(SP)
PENL 'Select Titles and Annotations'
_SetWTitle

JSR DisplayPrompts
JSR DisplayWindows

MOVE #3,-(SP)
_DeleteMenu ;remove Options menu from list

LEA SelectHandle,A1
MOVE.L (A1),-SP
CLR -(SP)
_InsertMenu ;put Select menu after all others

(continued)
Listing A.1 (continued)

`...DrawMenuBar ;re-draw menu bar
MOVE.L EditHandle,-(SP)
MOVE *1,-(SP)
...Disable Item`

`SelectEvent
MOVE.L ActiveTextHandle,-(SP)
...Idle

`...System Task ;update desk accessories
CLR -(SP)
MOVE *-1,-(SP)
PEA EventRecord
...GetNextEvent ;get next event from queue
MOVE (SP)+,DB
CMP *0,0
BEQ SelectEvent ;no event encountered

MOVE What,0
CMP #mButDwnEvt,0
BEQ SelectMouseEvent
CMP #keyDwnEvt,0
BEQ SelectKeyEvent
CMP #activateEvt,0
BEQ SelectActivateEvent ;text window needs activating
CMP #updateEvt,0
BEQ SelectUpdateEvent ;window needs updating
BRA SelectEvent`

`SelectActivateEvent
JSR ActivateTextWindow
BRA SelectEvent`

`SelectUpdateEvent
JSR UpdateTextWindows
BRA SelectEvent`

`SelectKeyEvent
MOVE.B Modify,0
CMP.B *1,0
BEQ SelectKeyboardCommand ;command key pressed

MOVE Message+2,-(SP)
MOVE.L ActiveTextHandle,-(SP)

(continued)
_TEKey

BRA SelectEvent

SelectKeyboardCommand

CLRL -(SP)
MOVE Message+2,-(SP)
...MenuKey ;find out what key was pressed
BRA SelectSelections

SelectMouseEvent

CLR -(SP)
MOVE L Point,-(SP)
PEA WhichWindowPtr
...FindWindow ;where was mouse button pressed?
MOVE (SP)+,DØ

CMP *inMenuBar,DØ
BEQ SelectMenuBar ;mouse button pressed in menu bar

CMP *inSysWindow,DØ
BEQ SelectSysEvent ;mouse button pressed in desk accessory

CMP *inContent,DØ ;mouse button pressed in content area of text window?
BEQ SelectInWindow

BRA SelectEvent

SelectSysEvent

PEA EventRecord
MOVE L WhichWindowPtr,-(SP)
...SystemClick ;let the system handle it

BRA SelectEvent

SelectMenuBar

CLRL -(SP)
MOVE L Point,-(SP)
...MenuSelect ;which menu?

SelectSelections

MOVE (SP)+,D2

LEA WhichMenu,AØ
MOVE L D2,(AØ)

CLR -(SP)
...HiliteMenu ;unhighlight all menus

MOVE WhichMenu,DØ

CMP #1,DØ

(continued)
Listing A.1 (continued)

BNE SelectMenu2
JSR AppleMenu ; in Apple menu

SelectMenu2
  CMP #2,DL
  BNE SelectMenu7
  JSR EditMenu ; in Edit menu

SelectMenu7
  CMP #7,DL ; in Select menu?
  BNE SelectEvent

SelectMenuOptions
  MOVE WhatItem,DL
  CMP #1,DL ; Display all?
  BNE SelectOptions2
  MOVE #1,D4 ; flag says “display annotations”
  JSR SelectAll

SelectOptions2
  CMP #2,DL ; Display all titles?
  BNE SelectOptions3
  MOVE #0,D4 ; flag says “don’t print annotations”
  JSR SelectAll

SelectOptions3
  CMP #3,DL ; Display one title?
  BNE SelectOptions4
  MOVE #0,ReturnFlag(A3) ; return flag (call is from Select)
  JSR SelectOneTitle

SelectOptions4
  CMP #4,DL ; Select by producer
  BNE SelectOptions5
  MOVE.L ProducerTextHandle,A1
  MOVE.L (A1),A2
  MOVE teLength(A2),DL
  CMP #0,DL
  BEQ SelectGoof
  MOVE #0Producer,D4 ; offset into record
  MOVE #2B,D6 ; number of characters in field
  JSR ClearNewRecord
  JSR MoveProducer
  JSR SequentialSearch

SelectOptions5
  CMP #5,DL ; Select by date
  BNE SelectOptions6
  MOVE.L DateTextHandle,A1
  MOVE.L (A1),A2

(continued)
MOVE teLength(A2),D0
CMP #0,D0
BEQ SelectGoof
MOVE #0ReleaseDate,D4
MOVE #4,D6
JSR ClearNewRecord
JSR MoveDate
JSR SequentialSearch

SelectOptions6
CMP #6,D0 ;Select by rating
BNE SelectOptions7
MOVEL. RatingTextHandle,A1
MOVEL (A1),A2
MOVE teLength(A2),D0
CMP #0,D0
BEQ SelectGoof
MOVE #0Rating,D4
MOVE #4,D6
JSR ClearNewRecord
JSR MoveRating
JSR SequentialSearch

SelectOptions7
CMP #7,D0 ;Select by tape number
BNE SelectOptions8
MOVEL. NumberTextHandle,A1
MOVEL (A1),A2
MOVE teLength(A2),D0
CMP #0,D0
BEQ SelectGoof
MOVE #0TapeNumber,D4
MOVE #4,D6
JSR ClearNewRecord
JSR MoveNumber
JSR SequentialSearch

SelectOptions8
CMP #8,D0 ;Quit
BNE SelectEvent

MOVE #7,-(SP)
_DeleteMenu

MOVE #9,D0
RTS

SelectlnWindow
CLRL -(SP)
_FrontWindow
MOVEL (SP)+,A0

(continued)
Listing A.1 (continued)

```
CMP.L WhichWindowPtr, A0
BNE SelectMustActivate

PEA Point
  ...GlobalToLocal

MOVE.L Point,-(SP)
BTST *shiftKey, Modify
SNE D0
MOVE.B D0,-(SP)
MOVE.L ActiveTextHandle,-(SP)
  _TEClick

BRA SelectEvent

SelectMustActivate
  JSR SelectTextWindow
  BRA SelectEvent

SelectAll
  LEA RecordCounter, A0
  MOVE *D, (A0)
  MOVE TotalRecords, StopNumber(A5)

AllLoop
  JSR DisplayOneRecord
  CMP *1, D4
  BNE Box
  JSR Display Annotation

Box
  LEA RecordCounter, A0
  ADDQ *1, (A0)
  MOVE StopNumber(A5), D0
  CMP (A0), D0
  BEQ AlmostDone
  JSR DisplayDialog2
  CMP *2, D7
  BEQ Cancelled
  BRA AllLoop

AlmostDone
  JSR DisplayDialog3

Cancelled
  JSR DisplayWindows
  MOVE *9, D0
  RTS

SelectOneTitle
  MOVE.L NameTextHandle, A1
  MOVE.L (A1), A2
  MOVE teLength(A2), D0
  CMP *D, D0
  BEQ SelectGoof
  ;if text length is D, no selection criteria
```

(continued)
JSR ClearNewRecord
JSR MoveName; put selected tape name into NewRecord

;----------------------------- Binary Search -----------------------------

LEA TapeArray(A5),A2 ; start of tape array
MOVE TotalRecords,D1
SUBQ #1,D1 ; bottom pointer
MOVE D1,D3
SUBQ #1,D3 ; save last record-1 # for future reference
MOVE #*D,D2 ; top pointer

MidPoint
MOVE D2,D5 ; find middle record *
ADD D1,D5
DIVU #*D,D5
AND.L #$000000FF,D5 ; mask off remainder
CMP #*D,D5
BLE TopRec ; handle first two records
CMP D5,D3
BLE BottomRec ; handle last two records

JSR ComputeAddress2
MOVEM.L D1-D5/A1-A2,-(SP) ; save registers
CLR.W -(SP) ; space for result
MOVEM.L A4,-(SP) ; pointer to record in tape array
PEA NewRecord(A5) ; pointer to search string
MOVE.W #*D,-(SP) ; number of characters to compare
MOVE.W #*D,-(SP) ; number of characters to compare
MOVE.W #*D,-(SP)
.Pack6 ; invoke the package
MOVE.W (SP)+,D6 ; recover result
MOVEM.L (SP)+,D1-D5/A1-A2 ; restore registers

CMP #*D,D6 ; check result of string compare
BGT TopHalf ; array greater than search string
BLT BottomHalf ; array less than search string

LEA RecordCounter,A0
MOVE D5,(A0)
JSR DisplayOneRecord ; must be equal - record has been found
MOVE ReturnFlag(A5),D6
CMP #*D,D6 ; which module called this routine?
BEQ KeepGoing ; call was from Select
RTS ; call was from Change or Delete

KeepGoing
JSR DisplayDialog3 ; display find & wait dialog box
JSR DisplayWindows ; clear text edit windows
RTS ; return to Select menu

(continued)
Listing A.1 (continued)

BottomHalf
  MOVE D5,D2 ;move top pointer down
  BRA NoFindCheck
TopHalf
  MOVE D5,D1 ;move bottom pointer up
NoFindCheck
  CMP D2,D1
  BMI NoFind ;pointers have crossed
  BRA MidPoint ;find new middle record and go again
NoFind
  JSR DisplayDialog1 ;displays “none found” dialog box
  JSR DisplayWindows ;clear screen and text edit records
  RTS ;return to Select menu
TopRec
  MOVE *0,05
  JSR OneCheck
  MOVE *1,05
  JSR OneCheck
  BRA NoFind
BottomRec
  MOVE D3,D5
  JSR OneCheck
  ADDQ *1,03
  MOVE D3,D5
  JSR OneCheck
  BRA NoFind
OneCheck
  JSR ComputeAddress2
  MOYEL D1-D5/A1-A2,-(SP)
  CLR.W -(SP) ;space for result
  MOVE.L A4,-(SP) ;pointer to array
  PEA Ne'A'Record(A5) ;pointer to search string
  MOVE.W *30,-(SP) ;number of characters to compare
  MOVE.W *30,-(SP) ;number of characters to compare
  MOVE.W *10,-(SP)
  ...Pack6 ;invoke the package
  MOVE.W (SP)+,D0 ;recover result
  MOYEL (SP)+,D1-D5/A1-A2
  CMP *0,D0
  BNE WrongOne ;correct record not found
  LEA RecordCounter,A0
  MOVE D5,A0
  JSR DisplayOneRecord
  MOVE ReturnFlag(A5),D0
  CMP *0,D0 ;where does this call originate?
  BEQ OneCheckContinues ;call comes from Select
(continued)
MOVE.L (SP)+,D0' ;pull extra subroutine return address from stack
RTS ;call comes from Change or Delete

OneCheckContinues
JSR DisplayDialog3
JSR DisplayWindows
MOVE *9,D0'
MOVE.L (SP)+,D7 ;pop subroutine return address off stack
RTS ;return directly to "Select" routine
WrongOne
MOVE *9,D0' ;return to Top or Bottom
RTS

SelectGoof
JSR NoSelectionCriteria
MOVE *9,D0'
RTS ;------- Sequential Search for equality on Producer, Rating, Date, or Number ------

SequentialSearch
LEA TapeArray(A5),A2
LEA NewRecord(A5),A1
ADD.L D4,A1 ;adds offset into NewRecord
MOVE TotalRecords,D1
SUBQ #1,D1 ;number of last records
LEA RecordCounter,A0'
MOVE *B,(A0') ;initialize record counter

SequentialSearch1
LEA RecordCounter,A0'
MOVE (A0),D5
JSR ComputeAddress2 ;finds start of TapeArray record
ADD.L D4,A4 ;adds offset into TapeArray record

MOVEM.L D1/A1/A2,-(SP);save critical registers
CLR.W -(SP)
MOVE.L A4,-(SP)
MOVE.L A1,-(SP)
MOVE.W D6,-(SP) ;characters to compare
MOVE.W D6,-(SP)
MOVE.W #1B,-(SP) ;IUMagString
.Move6
MOVE.W (SP)+,D0'
MOVEM.L (SP)+,D1/A1/A2 ;restore critical registers

CMP *B,D0'
BEQ SequentialDisplay
LEA RecordCounter,A0'
CMP (A0),D1

(continued)
Listing A.1 (continued)

BEQ EndofArray
ADDQ $1, (A0)

SequentialDisplay
MOVEM.L D1/A1/A2,-(SP)
JSR DisplayOneRecord
MOVEM.L (SP)+,D1/A1/A2

LEA RecordCounter, A0
CMP (A0), D1 ; last record?
BNE SequentialDisplay2
JSR DisplayDialog3
JSR DisplayWindows
MOVE #9, D0
RTS

SequentialDisplay2
MOVEM.L D1/A1/A2,-(SP)
JSR DisplayDialog2
MOVEM.L (SP)+,D1/A1/A2
LEA RecordCounter, A0
ADDQ $1, (A0)
BRA SequentialSearch1

EndofArray
JSR DisplayWindows
JSR DisplayDialog1
MOVE #9, D0
RTS

;--------------------- Print Lists -------------------------------

Print MOVE.L MainWindowPtr,-(SP)
PEA 'Print Titles and Annotations'
_SetWTitle

MOVE #3,-(SP)
_DeleteMenu ; remove Options menu from list

LEA PrintHandle, A1
MOVE.L (A1),-(SP)
CLR -(SP)
_InsertMenu ; put Print menu in list

MOVE.L EditHandle,-(SP)
MOVE #0,-(SP)
_DisableItem ; disable entire edit menu

_DrawMenuBar ; re-draw menu bar

(continued)
PrintEvent
   .SystemTask ;update desk accessories
   CLR -(SP)
   MOVE *-1,(SP)
   PEA EventRecord
   _GetNextEvent ;get next event from queue
   MOVE (SP)+,D0
   CMP *D0
   BEQ PrintEvent ;no event encountered - keep checking
   MOVE What,D0
   CMP *mButDownEvt,D0 ;mouse button pressed?
   BEQ PrintMouseEvent
   CMP *keyDownEvt,D0 ;key pressed?
   BEQ PrintKeyEvent
   BRA PrintEvent ;look for another event

PrintKeyEvent
   MOVE.B Modify,D0
   CMP.B *1,D0 ;command key pressed?
   BEQ PrintKeyboardCommand
   BRA PrintEvent

PrintKeyboardCommand
   CLR.L -(SP)
   MOVE Message+2,-(SP)
   _MenuKey ;what key was pressed?
   BRA PrintSelections

PrintMouseEvent
   CLR -(SP)
   MOVE.L Point,-(SP)
   PEA WhichWindowPtr
   _FindWindow ;where was mouse button pressed?
   MOVE (SP)+,D0
   CMP *inMenuBar,D0 ;pressed in menu bar?
   BEQ PrintMenuBar
   CMP *inSysWindow,D0 ;pressed in desk accessory?
   BEQ PrintSysEvent
   BRA PrintEvent

PrintSysEvent
   PEA EventRecord

(continued)
Listing A.1 (continued)

MOVE.L WhichWindowPtr,-(SP)
_:SystemClick ;let the system handle it

BRA PrintEvent

PrintMenuBar
  CLR.L -(SP)
  MOVE.L Point,-(SP)
  ._MenuSelect ;which menu?

PrintSelections
  MOVE.L (SP)+,D2
  LEA WhichMenu,AB
  MOVE.L D2,(AB)
  CLR -(SP)
  ._HiliteMenu ;unlightlight all menus

  MOVE WhichMenu,D0

  CMP #1,D0 ;in Apple menu?
  BNE PrintMenu2
  JSR AppleMenu

PrintMenu2
  CMP #2,D0 ;in Edit menu?
  BNE PrintMenu8
  JSR EditMenu

PrintMenu8
  CMP #8,D0 ;in Print menu?
  BNE PrintEvent
  JSR PrintOptions

PrintOptions
  MOVE WhichItem,D0

  CMP #1,D0 ;Print all?
  BNE PrintOption2
  JSR PrintAll

PrintOption2
  CMP #2,D0 ;Print all titles?
  BNE PrintOption3
  JSR PrintAllTitles

PrintOption3
  CMP #3,D0 ;quit?
  BNE PrintEvent

  MOVE #8,-(SP)
  ._DeleteMenu ;remove Print menu

(continued)
MOVE.L EditHandle,-(SP)       ;Enable edit menu
MOVE *9,-(SP)                  ;enable entire edit menu

PrintAll
JSR PrOpen                     ;open printing manager
MOVE.L *PrintSize,0           ;size of print record
_LEahandle                   ;allocate heap space for print record
LEA PrintRecordHandle,A2      ;store handle to print record
MOVE.L A2,-(SP)               ;handle back on stack
JSR PrintDefault              ;fill default info into print record
CLR -(SP)                     ;space for boolean result
LEA PrintRecordHandle,A2      ;allocate heap space for print record
MOVE.L (A2),-(SP)             ;store handle to print record
JSR PrJobDialog                ;draft or spooled?
MOVE (SP)+,0                  ;remove result
BEQ PrintFinish               ;user clicked cancel - close up shop
JSR PrintAlert                 ;tell user to ready the printer
CLR -(SP)                     ;space for pointer to printer port
LEA PrintRecordHandle,A2      ;allocate heap space for print record
MOVE.L (A2),-(SP)             ;store handle to print record
CLR -(SP)                     ;no scaling
JSR PrOpenPage                 ;open a new page
MOVEM.L (SP)+,02/D7            ;restore record counters

AnnotAnotherPage
JSR AnnotPrintOnePage
CMP D2,D7                      ;no scaling
BLT AnnotAnotherPage
MOVE.L PrPortPtr(A5),-(SP)     ;pointer to printer port
JSR PrCloseDoc                 ;close the document
BRA PrintFinish                ;all done

AnnotPrintOnePage
MOVEM.L D2/D7,-(SP)            ;save record counter
MOVEM.L PrPortPtr(A5),-(SP)    ;pointer to printer port
CLR -(SP)                      ;no scaling
JSR PrOpenPage                 ;open a new page
MOVEM.L (SP)+,D2/D7            ;restore record counters

(continued)
Listing A.1 (continued)

MOVE *monaco,-(SP)
  _TextFont

MOVE *12,-(SP)
  _TextSize

MOVEM.L D2/D7,-(SP)
PEA FontInfoStorage(A5)
  _GetFontInfo
MOVEM.L (SP)+,D2/D7
MOVE FontInfoStorage+ascent(A5),D4
ADD FontInfoStorage+descent(A5),D4
ADD FontInfoStorage+leading(A5),D4   ;height of line

LEA PrintRecordHandle,A2
MOVEM.L (A2),A8
MOVEM.L (A8),A8                   ;de-reference to get pointer
MOVE printInfo+pageNumber(A8),D6   ;page bottom coordinate
SUB FontInfoStorage+descent(A5),D6

MOVE D4,D3

JSR ClearPrintLine
ADD D4,D3    ;one blank line
ADD D4,D3    ;another blank line

MOVEM.L D2/D7,-(SP)
JSR PrintHeadings
MOVEM.L (SP)+,D2/D7

AnnotRecordPrint

MOVE D7,D8
MULU *64,D8
ADD *oAnnotNum,D8             ;offset into tape array
LEA TapeArray(A5),A8
ADD D8,A8             ;start of record in array
MOVE (A8),D8        ;retrieve annotation number

MOVEL *0,-(SP)
MOVEL *256,-(SP)
MOVEL AnnotationTextHandle,-(SP)
  _TSetSelect
MOVEL AnnotationTextHandle,-(SP)
  _TSet
LEA fiRefNum,A8
MOVE (A8),ioParamBlock+ioRefNum(A5) ;somehow Printing Manager trashes param block
LEA DataBuffer(A5),A8
MOVEL A8,ioParamBlock+ioBuffer(A5)
MOVEL *256,ioParamBlock+ioByteCount(A5)
MOVEL *1,ioParamBlock+ioPosMode(A5) ;read relative to start of file

(continued)
MOVE   D7,D5          ;number of current record
MULU   #64,D5         ;offset into tape array
ADD    #oAnnotNum,D5
LEA    TapeArray(A5),AØ
ADD.L  D5,AØ          ;AØ has location of annot. number
MOVE   (AØ),D6        ;retrieve annot. number
MULU   #256,D6        ;offset into file
MOVE.L D6,ioParamBlock+ioPosOffset(A5)

LEA    ioParamBlock(A5),AØ
_Read

LEA    DataBuffer(A5),AØ
MOVE.L AØ,-(SP)
MOVE.L #256,-(SP)
MOVE.L AnnotationTextHandle,-(SP)
_TEInsert

CLR.L  -(SP)
MOVE.L AnnotationTextHandle,-(SP)
_TEGetText
MOVE.L (SP)+,A6
MOVE.L (A6),A6
;de-reference to get pointer

MOVE   #4,D1
LEA    AnnotationTextHandle,AØ
MOVE.L (AØ),AØ        ;get handle
MOVE.L (AØ),AØ        ;de-reference to get pointer
MOVE teNLines(AØ),DØ  ;number of lines of text
ADD    DØ,D1          ;total number of lines in this entry

MULU   D4,D1
ADD    D3,D1
;where you will end up if this is printed
CMP     D6,D1         ;will this one fit on the page?
BLT     EnoughRoom
BRA     PageFinish

EnoughRoom
MOVEM.L D2/D7,-(SP)
JSR    PrintOneRecord
JSR    ClearPrintLine
ADD     D4,D3          ;get a blank line
MOVEM.L (SP)+,D2/D7

LEA    AnnotationTextHandle,A2
MOVE.L (A2),A2
MOVE.L (A2),A2
MOVE teNLines(A2),DØ  ;get number of lines again
MOVE   #Ø,D1
Listing A.1 (continued)

AnotherLine

```
MOVEL.L D2/D4,-(SP) ;save D4 (line height) & D2 (total records)
ADDQ *1,D1 ;look at next line
CMP  D1,D6 ;at last line?
BEQ  LastLine
SUBQ *1,D1 ;restore current line #
MOVE *2,D4 ;line starts are stored as integers
MOVE telines(A2,D4),D2 ;line start of this line
ADDQ *2,D4
MOVE telines(A2,D4),D5 ;start of next line
SUB  D2,D5 ;D5 has number of bytes
MOVE *20,-(SP) ;annotation is indented 20 pixels
MOVE D3,-(SP)

MOVEL.L D0/D1/D7/A2/A6,-(SP)
MOVEL A6,-(SP) ;pointer to text
MOVE D2,-(SP) ;starting position
MOVE D5,-(SP) ;number of bytes to print

.DrawText
MOVEL.L (SP)+,D8/D1/D7/A2/A6
MOVEL.L (SP)+,D2/D4
ADDQ *1,D1 ;increment line counter
ADD  D4,D3 ;space to next line
BRA AnotherLine

LastLine

SUBQ *1,D1 ;restore current line #
MOVEL.L D1/D3/D7/A2/A6,-(SP)
MULU  *2,D1
MOVE telines(A2,D1),D5 ;start of last line
MOVE *257,D6 ;total characters + 1
SUB  D5,D6 ;characters left to print

MOVE *20,-(SP)
MOVE D3,-(SP)

.MoveTo

MOVEL A6,-(SP)
MOVEL D5,-(SP)
MOVEL D0,-(SP)

.DrawText
MOVEL.L (SP)+,D1/D3/D7/A2/A6
MOVEL.L (SP)+,D2/D4

ADD  D4,D3

JSR  ClearPrintLine
ADD  D4,D3 ;one blank line
ADD  D4,D3 ;another blank line
```

(continued)
ADDQ #1,D7
CMP D2,D7
BEQ PageFinish ;all records printed
BRA AnnotRecordPrint

------------------- print without annotations -------------------------

PrintAllTitles
JSR PrOpen ;open printing manager (on disk - not in ROM)
MOVE.L #PrintSize,D8 ;size of print record
NEWHandle ;allocate heap space for print record
LEA PrintRecordHandle,A2
MOVE.L A8,(A2) ;store handle to print record

MOVE.L A8,-(SP) ;put handle on stack
JSR PrintDefault ;fill default info into print record
CLR -(SP) ;space for boolean result
LEA PrintRecordHandle,A2
MOVE.L (A2),-(SP)
JSR PrJobDialog ;draft or spooled?
MOVE (SP)+,D8 ;remove result
BEQ PrintFinish ;user clicked CANCEL - must close up shop

JSR PrintAlert ;tell user to ready the printer
CLR.L -(SP) ;space for pointer to printer port
LEA PrintRecordHandle,A2
MOVE.L (A2),-(SP)
CLR.L -(SP) ;let system allocate new port
CLR.L -(SP) ;use system I/O buffer
JSR PrOpenDoc ;allocate custom printer port
MOVE.L (SP)+,PrPortPtr(A5) ;retrieve pointer

MOVE.L #8,D7 ;initialize a record counter
MOVE.L #8,D2 ;clear out register
LEA TotalRecords,A8
MOVE (A8),D2

AnotherPage
JSR PrintOnePage
CMP D2,D7
BLT AnotherPage
MOVE.L PrPortPtr(A5),-(SP)
JSR PrCloseDoc ;close the document

PrintFinish
MOVE.L #8,D8 ;clear out register
LEA PrintRecordHandle,A2
MOVE.L (A2),A8
MOVE.L (A8),A8 ;get pointer
MOVE.B prJob+buDocLoop(A8),D8
BEQ Closeup ;draft printing was done

APPENDIX A 423
Listing A.1 (continued)

LEA PrintRecordHandle, A2
MOVE.L (A2), -(SP)
CLR.L -(SP) ; let spooler set up its own printing port
CLR.L -(SP) ; let spooler use its own buffer
CLR.L -(SP) ; let spooler use its own device buffer
PEA PrinterStatusRec(A5)
JSR PrPioFile ; image and print spool file

Closeup

MOVE.L PrintRecordHandle, AØ
 DisposHandle ; free space taken by print record
JSR PrClose ; close print manager
MOVE #9, DØ
RTS

PrintOnePage

MOVEM.L D2/D7, -(SP) ; save record counter
MOVE.L PrPortPtr(A5), -(SP) ; pointer to printer port
CLR.L -(SP) ; no scaling
JSR PrOpenPage ; begin a new page
MOVEM.L (SP)+, D2/D7 ; restore record counter
MOVE *monaco, -(SP) ; printing will be in monaco font

MOVEM.L D2/D7, -(SP)

PEA FontInfoStorage(A5) ; pointer to font info record
GetFontInfo ; font characteristics needed to calculate end of page
MOVEM.L (SP)+, D2/D7

MOVE FontInfoStorage+ascent(A5), D4
ADD FontInfoStorage+descent(A5), D4
ADD FontInfoStorage+leading(A5), D4 ; calculates height of line

LEA PrintRecordHandle, A2
MOVE.L (A2), AØ

MOVE.L (AØ), AØ ; get pointer from handle
MOVE prInfo+Page+bottom(AØ), D6 ; page bottom coordinate
SUB FontInfoStorage+descent(A5), D6 ; adjust for font descent

MOVE D4, D3 ; initial vertical position

JSR ClearPrintLine
ADD D4, D3 ; blank line
ADD D4, D3 ; blank line

MOVEM.L D2/D7, -(SP)
JSR PrintHeadings
MOVEM.L (SP)+, D2/D7

(continued)
RecordPrint

    MOVEM.L D2/D7,-(SP)      ;save record counter
    JSR PrintOneRecord
    MOVEM.L (SP)+,D2/D7
    ADDQ #1,D7
    CMP D2,D7
    BEQ PageFinish    ;all records printed - close up shop
    CMP D6,D3          ;at bottom of page?
    BLT RecordPrint    ;not at bottom - print another record

PageFinish

    MOVEM.L D2/D7,-(SP)      ;save record counter
    MOVEM.L PrPortPtr(A5),-(SP)  
    JSR PrClosePage
    MOVEM.L (SP)+,D2/D7      ;restore record counter
    RTS

ClearPrintline   ;fill print line with blanks

    LEA PrintLineMask,A0
    LEA PrintLine(A5),A1
    MOVE #182,D0
    _BlockMove

    MOVE.B #188,PrintLine(A5)    ;set length of print line

    RTS

PrintHeadings

    LEA PageHead,A0
    LEA PrintLine+40(A5),A1
    MOVE #11,D0
    _BlockMove
    MOVE #0,-(SP)
    MOVE D3,-(SP)
    _MoveTo
    MOVEM.L D1/D2/D7,-(SP)
    PEA PrintLine(A5)
    _DrawString
    MOVEM.L (SP)+,D1/D2/D7
    ADD D4,D3

    JSR ClearPrintLine
    ADD D4,D3      ;blank line
    ADD D4,D3      ;blank line
    MOVE #4,-(SP)  ;underline the column headings
        _TextFace
    LEA TitleHead,A0
    LEA PrintLine+12(A5),A1
    MOVE #5,D0
    _BlockMove

(continued)
Listing A.1 (continued)

LEA ProducerHead,A0
LEA PrintLine+44(A5),A1
MOVE *8,D0
LEA DateHead,A0
LEA PrintLine+66(A5),A1
MOVE *4,D0

LEA RatingHead,A0
LEA PrintLine+72(A5),A1
MOVE *4,D0

LEA NumberHead,A0
LEA PrintLine+78(A5),A1
MOVE *4,D0

MOVE *8,-(SP)
MOVE D3,-(SP)

MOVEM.L D1/D2/D7,-(SP)
PEA PrintLine(A5)
_DrawString
MOVEM.L (SP)+,D1/D2/D7

ADD D4,D3

MOVE *8,-(SP) ;back to normal
_TextFace
JSR ClearPrintLine
ADD D4,D3 ;blank line

RTS

PrintOneRecord
JSR ClearPrintLine
LEA TapeArray(A5),A2

MOVEM.D D7,-(SP) ;save record counter
MOVE D7,D5
JSR ComputeAddress2 ;address returned in A4
MOVEM.L (SP)+,D7 ;restore record counter

MOVE.L A4,A0 ;start of record
LEA PrintLine+12(A5),A1
MOVE *30,D0

MOVE.L A4,A0
ADD.L *oProducer,A0
LEA    PrintLine+44(A5),A1
MOVE   #20,D0
       _BlockMove                ;moves Producer
MOVE.L A4,A0
ADD.L #0ReleaseDate,A0
LEA    PrintLine+66(A5),A1
MOVE   *4,D0
       _BlockMove                ;moves Date
MOVE.L A4,A0
ADD.L #0Rating,A0
LEA    PrintLine+72(A5),A1
MOVE   *4,D0
       _BlockMove                ;moves Rating
MOVE.L A4,A0
ADD.L #0TapeNumber,A0
LEA    PrintLine+78(A5),A1
MOVE   *4,D0
       _BlockMove                ;moves Tape Number
MOVE   #0,-(SP)
MOVE   D3,-(SP)
       _MoveTo
MOVEM.L D1/D2/D7,-(SP)
PEA    PrintLine(A5)
       _DrawString
MOVEM.L (SP)+,D1/D2/D7
ADD    D4,D3

RTS

PrintAlert
  CLR    -(SP)        ;space for integer result
  MOVE   #5,-(SP)     ;alert ID
  CLR.L  -(SP)       ;use standard filter procedure
  _Alert
  MOVE   (SP)+,D0     ;pop result
  RTS

; ------------------------ Set up the main menu ------------------------

MainMenuBar
LEA    AppleHandle,A1
MOVE.L (A1),-(SP)     ;Put handle on stack again
CLR    -(SP)          ;shows that this menu is after all others
       _InsertMenu     ;Puts menu in list
LEA    EditHandle,A1
MOVE.L (A1),-(SP)     ;Put handle on stack again
Listing A.1 (continued)

CLR -(SP) ;Put menu after the first one
_InsertMenu ;Put menu in list

RedrawMainMenu
LEA OptionsHandle, A1
MOVE.L (A1), -(SP) ;Put handle on stack again
CLR -(SP) ;This menu is after the other two
_InsertMenu ;Put menu in list

_DrawMenuBar ;Draw the menu bar
RTS

; ------------ Make the text windows visible --------------------------

DisplayWindows
MOVE.L AnnotationWindowPtr, -(SP)
_SelectWindow
MOVE.L AnnotationWindowPtr, -(SP)
_SetPort
MOVE.L @0, -(SP)
MOVE.L #256, -(SP)
MOVE.L AnnotationTextHandle, -(SP)
_TESetSelect ;select all the text in the window
MOVE.L AnnotationTextHandle, -(SP)
_TECut ;cut out text from previous use

MOVE.L AnnotationWindowPtr, -(SP)
SF -(SP)
_HiliteWindow ;get rid of highlighting in this window

MOVE.L NumberWindowPtr, -(SP)
_SelectWindow
MOVE.L NumberWindowPtr, -(SP)
_SetPort
MOVE.L @0, -(SP)
MOVE.L 2@0, -(SP)
MOVE.L NumberTextHandle, -(SP)
_TESetSelect
MOVE.L NumberTextHandle, -(SP)
_TECut

MOVE.L RatingWindowPtr, -(SP)
_SelectWindow
MOVE.L RatingWindowPtr, -(SP)
_SetPort
MOVE.L @0, -(SP)
MOVE.L #4, -(SP)
MOVE.L RatingTextHandle, -(SP)
_TESetSelect
MOVE.L RatingTextHandle, -(SP)
_TECut

(continued)
MOVE.L DateWindowPtr,-(SP)  
.SelectWindow  
MOVE.L DateWindowPtr,-(SP)  
.SetPort  
MOVE.L #8,-(SP)  
MOVE.L #5,-(SP)  
MOVE.L DateTextHandle,-(SP)  
.TESetSelect  
MOVE.L DateTextHandle,-(SP)  
.TECut  

MOVE.L ProducerWindowPtr,-(SP)  
.SelectWindow  
MOVE.L ProducerWindowPtr,-(SP)  
.SetPort  
MOVE.L #8,-(SP)  
MOVE.L #22,-(SP)  
MOVE.L ProducerTextHandle,-(SP)  
.TESetSelect  
MOVE.L ProducerTextHandle,-(SP)  
.TECut  

MOVE.L $00000100,0  ;mask to remove activate events  
.FlushEvents  

MOVE.L NameWindowPtr,-(SP)  
.SelectWindow  
MOVE.L NameWindowPtr,-(SP)  
.SetPort  
MOVE.L #8,-(SP)  
MOVE.L #32,-(SP)  
MOVE.L NameTextHandle,-(SP)  
.TESetSelect  
MOVE.L NameTextHandle,-(SP)  
.TECut  

LEA ActiveTextHandle,A0  
MOVE.L NameTextHandle,(A0)  ;for TEIdle  

RTS  

;------------------ Select the appropriate text window ------------------
.SelectTextWindow  

LEA ActiveTextHandle,A1  
MOVE.L WhichWindowPtr,A0  

CMP.L NameWindowPtr,A0  ;check to identify specific window  
BNE Select1  
MOVE.L NameTextHandle,(A1)  ;pass appropriate handle to TEIdle  
BRA Select6  

(continued)
Listing A.1 (continued)

```assembly
Select1 CMP.L ProducerWindowPtr,AØ
BNE Select2
MOVE.L ProducerTextHandle,(A1)
BRA Select6

Select2 CMP.L DateWindowPtr,AØ
BNE Select3
MOVE.L DateTextHandle,(A1)
BRA Select6

Select3 CMP.L RatingWindowPtr,AØ
BNE Select4
MOVE.L RatingTextHandle,(A1)
BRA Select6

Select4 CMP.L NumberWindowPtr,AØ
BNE Select5
MOVE.L NumberTextHandle,(A1)
BRA Select6

Select5 CMP.L AnnotationWindowPtr,AØ
BNE Select7 ;not a text window
MOVE.L AnnotationTextHandle,(A1)

Select6 MOVE.L WhichWindowPtr,-(SP)
.Select5

Select7 RTS

; --------- Handle activate events in text windows -------------------
ActivateTextWindow

    MOVE.L Message,AØ ;get pointer to window which posted event
    MOVE Modify,DØ
    BTST "activeFlag,DØ ;activate bit set?
    BEQ DeActivate ;if not set, window was deactivated

Activate1

    CMP.L NameWindowPtr,AØ ;name window event?
    BNE Activate2

Activate2

    CMP.L ProducerWindowPtr,AØ
    BNE Activate3

Activate3

    CMP.L DateWindowPtr,AØ
    BNE Activate4
```

(continued)
MOVE.L DateTextHandle, -(SP)
_THA Activate
BRA Activate99

Activate4
CMI.L RatingWindowPtr, AØ
BNE Activate5
MOVE.L RatingTextHandle, -(SP)
_THA Activate
BRA Activate99

Activate5
CMI.L NumberWindowPtr, AØ
BNE Activate6
MOVE.L NumberTextHandle, -(SP)
_THA Activate
BRA Activate99

Activate6
CMI.L AnnotationWindowPtr, AØ
BNE Activate98 ; not one of our text windows
MOVE.L AnnotationTextHandle, -(SP)
_THA Activate

Activate99
MOVE.L Message, -(SP) ; make this the current grafport
_SETPort

Activate98
RTS

DeActivate
CMI.L NameWindowPtr, AØ
BNE DeActivate1
MOVE.L NameTextHandle, -(SP)
_THA DeActivate
RTS

DeActivate1
CMI.L ProducerWindowPtr, AØ
BNE DeActivate2
MOVE.L ProducerTextHandle, -(SP)
_THA DeActivate
RTS

DeActivate2
CMI.L DateWindowPtr, AØ
BNE DeActivate3
MOVE.L DateTextHandle, -(SP)
_THA DeActivate
RTS

(continued)
Listing A.1 (continued)

DeActivate3
    CMP.L RatingWindowPtr, AØ
    BNE DeActivate4
    MOVE.L RatingTextHandle,-(SP)
    _TeDeActivate
    RTS

DeActivate4
    CMP.L NumberWindowPtr, AØ
    BNE DeActivate5
    MOVE.L NumberTextHandle,-(SP)
    _TeDeActivate
    RTS

DeActivate5
    CMP.L AnnotationWindowPtr, AØ
    BNE DeActivate6 ; not a text window
    MOVE.L AnnotationTextHandle,-(SP)
    _TeDeActivate
    RTS

DeActivate6
    RTS

; ------------------------ Update Text Windows ------------------------
; This updates all windows, regardless of which one was active
UpdateTextWindows

    MOVE.L MainWindowPtr,-(SP)
    _BeginUpdate
    MOVE.L MainWindowPtr,-(SP)
    _SetPort
    PEA MainWindowRect
    _EraseRect
    JSR DisplayPrompts ; re-draw window's contents
    MOVE.L MainWindowPtr,-(SP)
    _EndUpdate

    MOVE.L NameWindowPtr,-(SP)
    _BeginUpdate
    MOVE.L NameWindowPtr,-(SP)
    _SetPort
    PEA NameViewRect
    _EraseRect
    PEA NameViewRect
    MOVE.L NameTextHandle,-(SP)
    _TeUpdate
    MOVE.L NameWindowPtr,-(SP)
    _EndUpdate

    MOVE.L ProducerWindowPtr,-(SP)
    _BeginUpdate

(continued)
MOVE.L ProducerWindowPtr,-(SP)
_SetPort
PEA ProducerViewRect
_EraseRect
PEA ProducerViewRect
MOVE.L ProducerTextHandle,-(SP)
_TEUpdate
MOVE.L ProducerWindowPtr,-(SP)
_EndUpdate

MOVE.L DateWindowPtr,-(SP)
_BeginUpdate
MOVE.L DateWindowPtr,-(SP)
_SetPort
PEA DateViewRect
_EraseRect
PEA DateViewRect
MOVE.L DateTextHandle,-(SP)
_TEUpdate
MOVE.L DateWindowPtr,-(SP)
_EndUpdate

MOVE.L RatingWindowPtr,-(SP)
_BeginUpdate
MOVE.L RatingWindowPtr,-(SP)
_SetPort
PEA RatingViewRect
_EraseRect
PEA RatingViewRect
MOVE.L RatingTextHandle,-(SP)
_TEUpdate
MOVE.L RatingWindowPtr,-(SP)
_EndUpdate

MOVE.L NumberWindowPtr,-(SP)
_BeginUpdate
MOVE.L NumberWindowPtr,-(SP)
_SetPort
PEA NumberViewRect
_EraseRect
PEA NumberViewRect
MOVE.L NumberTextHandle,-(SP)
_TEUpdate
MOVE.L NumberWindowPtr,-(SP)
_EndUpdate

MOVE.L AnnotationWindowPtr,-(SP)
_BeginUpdate
MOVE.L AnnotationWindowPtr,-(SP)
_SetPort
PEA AnnotationViewRect
_EraseRect

(continued)
Listing A.1 (continued)

```
PEA AnnotationViewRect
MOVE.L AnnotationTextHandle,-(SP)
_TEUpdate
MOVE.L AnnotationWindowPtr,-(SP)
_EndUpdate

RTS

;;;;;;;;;;;;;; Display prompts for text entry windows ;;;;;;;;;;;;;;;
DisplayPrompts
MOVE #sysFont,-(SP)
_TextFont

PEA NameTitle
MOVE.L #11,-(SP)
PEA NamePromptBox
MOVE L #1,-(SP)
_TextBox

PEA ProducerTitle
MOVE.L #22,-(SP)
PEA ProducerPromptBox
MOVE L #1,-(SP)
_TextBox

PEA DateTitle
MOVE.L #17,-(SP)
PEA DatePromptBox
MOVE L #1,-(SP)
_TextBox

PEA RatingTitle
MOVE.L #8,-(SP)
PEA RatingPromptBox
MOVE L #1,-(SP)
_TextBox

PEA NumberTitle
MOVE.L #13,-(SP)
PEA NumberPromptBox
MOVE L #1,-(SP)
_TextBox

RTS

;;;;;;;;;;;;;; Do the text edit functions ;;;;;;;;;;;;;;;
DoEditing
MOVE WhatItem,D.0 ;get set to figure out what was selected
CMP #3,D.0 ;cut?
BNE DoEditing1
MOVE.L ActiveTextHandle,-(SP)
```

(continued)
DoEditing1
CMP #4, D0 ; copy?
BNE DoEditing2
MOVE.L ActiveTextHandle, -(SP)
_TECopy
RTS

DoEditing2
CMP #5, D0 ; paste?
BNE DoEditing3
MOVE.L ActiveTextHandle, -(SP)
_TEPaste
RTS

DoEditing3
CMP #6, D0 ; clear?
BNE DoEditing4
; not a recognizable item
MOVE.L ActiveTextHandle, -(SP)
_TEDelete
RTS

; ----------------------------- Clear the NewRecord storage area -----------------------------
ClearNewRecord
LEA NewRecordMask, A0 ; all blanks
LEA NewRecord(A5), A1
MOVE.L #64, D0
_BlockMove
RTS

; ------------------------ Move data from text edit records to data record -----------------
MoveName
CLR.L -(SP) ; space for CharsHandle result
MOVE.L NameTextHandle, -(SP)
_TEGetText ; get handle to text in Name edit record
MOVE.L (SP)+, A2 ; recover CharsHandle
MOVE.L (A2), A0 ; source pointer for block move
LEA NewRecord+oTapeName(A5), A1 ; destination of block move
MOVE.L NameTextHandle, A3
MOVE.L (A3), A4
MOVE telLength(A4), D0 ; number of characters to move
_BlockMove
RTS

MoveProducer
CLR.L -(SP)
MOVE.L ProducerTextHandle, -(SP)
_TEGetText

(continued)
Listing A.1 (continued)

MOVE.L (SP)+,A2
MOVE.L (A2),A0
LEA NewRecord+oProducer(A5),A1
MOVE.L ProducerTextHandle,A3
MOVE.L (A3),A4
MOVE teLength(A4),D0
_BlockMove
RTS

MoveDate
CLR.L -(SP)
MOVE.L DateTextHandle, -(SP)
_TEGetText
MOVE.L (SP)+,A2
MOVE.L (A2),A0
LEA NewRecord+oReleaseDate(A5),A1
MOVE.L DateTextHandle,A3
MOVE.L (A3),A4
MOVE teLength(A4),D0
_BlockMove
RTS

MoveRating
CLR.L -(SP)
MOVE.L RatingTextHandle, -(SP)
_TEGetText
MOVE.L (SP)+,A2
MOVE.L (A2),A0
LEA NewRecord+oRating(A5),A1
MOVE.L RatingTextHandle,A3
MOVE.L (A3),A4
MOVE teLength(A4),D0
_BlockMove
RTS

MoveNumber
CLR.L -(SP)
MOVE.L NumberTextHandle, -(SP)
_TEGetText
MOVE.L (SP)+,A2
MOVE.L (A2),A0
LEA NewRecord+oTapeNumber(A5),A1
MOVE.L NumberTextHandle,A3
MOVE.L (A3),A4
MOVE teLength(A4),D0
_BlockMove
RTS

;------------------ Alert box processing for no selection criteria ------------------
NoSelectionCriteria
CLR. -(SP) ;space for alert item result
MOVE #4,-(SP) ;alert item ID

(continued)
MOVE.L #0,-(SP) ; use standard filter procedure
_Alert
MOVE (SP)+,D8 ; pull result from stack
RTS

; -------------- Display one record from array ----------------------
DisplayOneRecord
JSR DisplayWindows ; clears out text edit records
LEA RecordCounter,A8
MOVE (A8),D5
MULU #64,D5

MOVE.L NameWindowPtr,-(SP)
_SetPort
LEA TapeArray(A5),A8
ADD D5,A8
MOVE.L A8,-(SP) ; pointer to text
MOVE.L #38,-(SP) ; # of characters to get
MOVE.L NameTextHandle,-(SP) ; edit record which will get characters
_TEInsert ; incorporate text into record

MOVE.L ProducerWindowPtr,-(SP)
_SetPort
LEA TapeArray(A5),A8
ADD D5,A8
ADD.L #6Producer,A8
MOVE.L A8,-(SP)
MOVE.L #28,-(SP)
MOVE.L ProducerTextHandle,-(SP)
_TEInsert

MOVE.L DateWindowPtr,-(SP)
_SetPort
LEA TapeArray(A5),A8
ADD D5,A8
ADD.L #6ReleaseDate,A8
MOVE.L A8,-(SP)
MOVE.L #4,-(SP)
MOVE.L DateTextHandle,-(SP)
_TEInsert

MOVE.L RatingWindowPtr,-(SP)
_SetPort
LEA TapeArray(A5),A8
ADD D5,A8
ADD.L #6Rating,A8
MOVE.L A8,-(SP)
MOVE.L #4,-(SP)
MOVE.L RatingTextHandle,-(SP)
_TEInsert

(continued)
Listing A.1 (continued)

    MOVE.L NumberWindowPtr,-(SP)  
    _SetPort
    LEA    TapeArray(A5),A0
    ADD    D5,A0
    ADD.L  #0TapeNumber,A0
    MOVE.L A0,-(SP)
    MOVE.L #4,-(SP)
    MOVE.L NumberTextHandle,-(SP)
    _TInsert
    RTS

; ------------ Retrieve and display an annotation ------------------------

Display Annotation

    LEA    DataBuffer(A5),A0
    MOVE.L A0,ioParamBlock+ioBuffer(A5)
    MOVE.L #256,ioParamBlock+ioByteCount(A5)
    MOVE   #1,ioParamBlock+ioPosMode(A5)  ;read relative to start of file

    LEA    RecordCounter,A0
    MOVE   (A0),D5                   ;number of current record
    MULU   #64,D5                    ;offset into tape array
    ADD    #0AnnotNum,D5
    LEA    TapeArray(A5),A0
    ADD.L  D5,A0
    MOVE   (A0),D0                   ;A0 has location of annot. number
    MULU   #256,D0                   ;offset into file
    MOVE.L D0,ioParamBlock+ioPosOffset(A5)
    LEA    ioParamBlock(A5),A0
    _Read

    MOVE.L AnnotationWindowPtr,-(SP)  
    _SetPort
    LEA    DataBuffer(A5),A0
    MOVE.L A0,-(SP)
    MOVE.L #256,-(SP)
    MOVE.L AnnotationTextHandle,-(SP)
    _TInsert
    RTS

; ------------ Display and handle "Find and Wait" dialog box ------------

DisplayDialog

    CLR.L  -(SP)  ;space for dialog pointer
    MOVE   #3,-(SP)  ;dialog ID
    PEA    DialogWindRec(A5)  ;storage for dialog record
    MOVE   #1,-(SP)  ;put this dialog box in front
    _SetNewDialog

    MOVE.L (SP)+,DialogWindPtr(A5)  ;recover dialog pointer

(continued)
MOVE L DialogWndPtr(A5), -(SP)

_SetPort

Dialog3a
MOVE L #0,-(SP) ;use standard event filter
PEA _HatItem ;place to put number of item selected
.Move L Dialog ;let system monitor dialog box

Move L _HatItem,D0
Cmp #okButton,D0 ;OK button pressed?
Bne Dialog3a

.Move L DialogWndPtr(A5), -(SP) ;put dialog pointer on stack
_Move L CloseDialog ;remove dialog

RTS

; -------------- Display and handle "Find More?" dialog box --------------

DisplayDialog2
Clr L -(SP) ;space for dialog pointer
.Move L #2,-(SP) ;for dialog box #2
P ea DialogWndRec(A5) ;storage for dialog record
.Move L #1,-(SP) ;put dialog box in front
_Move L GetNewDialog

.Move L (SP)+,DialogWndPtr(A5) ;recover pointer

.Move L DialogWndPtr(A5), -(SP) ;put back on stack
_Move L SetPort

Dialog2a
Move L #0,-(SP) ;use standard filter procedure
P ea _HatItem ;space for item that was pressed
.Move L ModalDialog

Move _HatItem,D7
Cmp #okButton,D7
Beq Dialog2b

Cmp #cancelButton,D7
Bne Dialog2a

Dialog2b
.Move L DialogWndPtr(A5), -(SP)
_Move L CloseDialog

RTS ;if cancelled, returns to Select menu control

; ----------- Display and handle "No Find" dialog box -----------

DisplayDialog1
Clr L -(SP) ;space for dialog pointer
.Move L #1,-(SP) ;this is dialog box 1

(continued)
Listing A.1 (continued)

```
PEA DialogWindRec(A5) ; storage for dialog record
MOVE.L #1,-(SP) ; put dialog box in front
GetNewDialog ; get the dialog box

MOVE.L (SP)+,DialogWindPtr(A5) ; recover pointer
MOVE.L DialogWindPtr(A5),-(SP) ; put back on stack
SetPort

Dialog1a
MOVE.L #0,-(SP) ; use standard filter procedure
PEA WhatItem ; space for item that was pressed
ModalDialog

MOVE WhatItem,D0
CMP #okButton,D0
BNE Dialog1a

MOVE.L DialogWindPtr(A5),-(SP)
CloseDialog

RTS

; ------------ Pointers and storage for the seven window records --------------

MainWindowPtr DCL Ø
NameWindowPtr DCL Ø
ProducerWindowPtr DCL Ø
DateWindowPtr DCL Ø
RatingWindowPtr DCL Ø
NumberWindowPtr DCL Ø
AnnotationWindowPtr DCL Ø

MainWindowStorage DSWindowSize
NameWindowStorage DSWindowSize
ProducerWindowStorage DSWindowSize
DateWindowStorage DSWindowSize
RatingWindowStorage DSWindowSize
NumberWindowStorage DSWindowSize
AnnotationWindowStorage DSWindowSize

WhichWindowPtr DCL Ø ; place for FindWindow result

; --------------- Data Structures for TextEdit ------------------------

NameViewRect DC 1,1,19,249
NameDestRect DC 1,1,19,249
NameTextHandle DCL Ø
NamePromptBox DC 12,10,32,200
NameTitle DC 'Tape Name:'

(continued)
ProducerViewRect DC 1,1,19,175
ProducerDestRect DC 1,1,19,175
ProducerTextHandle DCL $0
ProducerPromptBox DC 37,10,57,200
ProducerTitle DC ‘Producer/Distributor:’

DateViewRect DC 1,1,19,42
DateDestRect DC 1,1,19,42
DateTimeHandle DCL $0
DatePromptBox DC 62,18,82,200
DateTitle DC ‘Date of Release:’

RatingViewRect DC 1,1,19,28
RatingDestRect DC 1,1,19,28
RatingTextHandle DCL $0
RatingPromptBox DC 87,18,107,200
RatingTitle DC ‘Rating:’

NumberViewRect DC 1,1,19,35
NumberDestRect DC 1,1,19,35
NumberTextHandle DCL $0
NumberPromptBox DC 112,10,132,200
NumberTitle DC ‘Tape Number:’

AnnotationViewRect DC 4,3,72,46
AnnotationDestRect DC 4,3,72,46
AnnotationTextHandle DCL $0

ActiveTextHandle DCL $0 ; holds text handle of active text window for TElde
MainWindowRect DC $0,248,490 ; for EraseRect

;------------------ Definitions for trapping events ------------------

everyEvent DCL $8088FFFE
EventRecord ; where GetNextEvent Puts its result
What DC $0
Message DCL $0
When DCL $0
Point DCL $0
Modify DC $0

;---------- These are the handles for the eight menus ----------

AppleHandle DCL $0
EditHandle DCL $0
OptionsHandle DCL $0
EnterHandle DCL $0
ChangeHandle DCL $0
DeleteHandle DCL $0
SelectHandle DCL $0
PrintHandle DCL $0

(continued)
Listing A.1 (continued)

WhichMenu DC Ø
WhatItem DC Ø

DeskAccName DCW 16,Ø
TapeArray DS.B 64ØØ
NewRecord DS.B 64
NewRecordMask DC.B 64,""
TotalRecords DC Ø
AnnotRecMask DC.B 256,""
LastAnnotNumb DC Ø
RecordCounter DC Ø

StopNumber DS.W 1

DialogWindRec DS dWindLen
DialogWindPtr DS.L 1

; -------------------------------- Data structures and storage for file operations --------------------------------
DataBuffer DS.B 256 ;need maximum 256 bytes for annotation
ioParamBlock DS.B ioQEISize ;I/O parameter blocks are 50 bytes
fiParamBlock DS.B ioFQEISize ;file info parameter blocks are 80 bytes
vParamBlock DS.B ioVQEISize ;volume parameter blocks are 64 bytes

ReturnFlag DS 1 ;source of call to SelectOneTitle
fiRefNum DC Ø ;place for file reference number

; -------------------------------- Data structures and constants for printing --------------------------------
PrintRecordHandle DC.L Ø
PrinterStatusRec DS.B iPrStatSize ;printer status record
PrintLine DS.B 182
PrintLineMask DC.B 1ØØ,""

PrPortPtr DS.L 1
FontInfoStorage DS.W 4 ;place to put font info for printing

PageHead DC 'Video Tapes'
TitleHead DC 'Title'
ProducerHead DC 'Producer'
DateHead DC 'Date'
RatingHead DC 'Ratg'
NumberHead DC 'Numb'

END
Listing A.2 Resource File for the Video Tape Index Program

tape.index:TapesRs.rc.REL

TYPE MENU

1

14

2

Edit

Undo/Z

(-)

Cut/X

Copy/C

Paste/V

Clear

3

Options

Enter

Change

Delete

Select

Print

Quit/Q

4

Enter

Add/A

Quit/Q

5

Change

Find Record/F

Save Change/S

Abandon Change/A

Quit/Q

6

Delete

Find Record/F

Delete/D

Cancel/C

Quit/Q

7

Select

Display All

Display All Titles

Select One Title

Select by Producer

(continued)
Listing A.2 (continued)

Select by Date
Select by Rating
Select by Tape Number
Quit/Q

; ; Print menu (print a couple of lists)

Print
Print All
Print All Titles
Quit/Q

TYPE WIND
; ; Main window

,1
Video Tape Index
48 10 300 500
visible NoGoAway
Ø
Ø

,2
Tape Name
50 240 70 490
visible NoGoAway
2
Ø

,3
Producer
75 240 95 415
visible NoGoAway
2
Ø

,4
Date
160 240 120 283
visible NoGoAway
2
Ø

,5
Rating
125 240 145 269
visible NoGoAway
2
Ø

; ; Tape name window

; ; Producer /Distributor window

; ; Date window

; ; Rating window

(continued)
; Tape Number
150 240 170 276 visible NoGoAway
2 Ø

; Annotation window
Annotation
285 28 28 490 visible NoGoAway Ø Ø

TYPE DLOG
; None Found dialog box
1 Dialog box for "None Found" condition
180 380 170 490 Visible NoGoAway
2 Ø 1

; Find More dialog box
2 Dialog box for "One Found/Find More?" condition
180 380 170 490 Visible NoGoAway
2 Ø 2

; One Found dialog box
3 Dialog box for "One Found" condition
180 380 170 490 Visible NoGoAway
2 Ø 3

TYPE ALRT
; No Selection Criteria alert
4 180 380 170 490 4 7765

; Ready Printer alert
5 50 140 120 390 5 4444

(continued)
Listing A.2 (continued)

```
6
5B 14B 12B 39B
6
5555

TYPE DITL
1
2

button
4B 11B 6B 17B

OK

static Text
1B 41 3B 149
None Found

2
3

button
4B 11B 6B 17B

OK

button
4B 2B 6B 8B
Cancel

static Text
1B 41 3B 149
Find More?

3
1

button
4B 11B 6B 17B
OK

4
2

button
4B 11B 6B 17B
OK

static Text
1B 5 3B 185
Selection criteria?
```

;; File Error alert

;; Item list for None Found dialog box

;; Item list for Find More dialog box

;; Item list for One Found dialog box

;; Item list for No Selection Criteria alert

(continued)
Turn on printer. Press "Enter".

; Item list for Ready Printer alert

Turn on printer. Press "Enter".

; Item list for File Error Alert

Unexpected file error!
The names of ToolBox and operating system routines discussed in this book are presented below, grouped by function to help when you know what you want to do but not what you need to do it with. Each routine is followed by a short description of what it does. Once you know the name of the routine you wish to use, the quickest way to locate details about it is to look in the index under the name of the routine.

1. INITIALIZING THE SYSTEM

Calls to the initialization routines should be made at the beginning of every program in the order in which they are listed below:

- **InitGraf**: Initializes QuickDraw. (Chapter 7)
- **InitFonts**: Initializes the Font Manager. (Chapter 9)
- **FlushEvents**: Flushes events from the event queue. (Chapter 8)
- **InitWindows**: Initializes the window manager. (Chapter 7)
- **InitMenus**: Initializes the menu manager. (Chapter 7)
- **InitDialogs**: Initializes the Dialog Manager. (Chapter 9)
- **TEInit**: Initializes Text Edit. (Chapter 9)
- **InitCursor**: Initializes the arrow cursor.
1. a Managing the Cursor

GetCursor: Retrieves the 16 x 16 pixel image of a cursor from the system resource file. (Chapter 11)

SetCursor: Changes the shape of the cursor. (Chapter 11)

2. USING A RESOURCE FILE

OpenResFile: Opens a resource file for program use. This routine is needed when resource definitions are kept in a file separate from the application's source code. (Chapter 7)

3. CREATING WINDOWS

GetNewWindow: Creates a new window from parameters contained in a resource file template. (Chapter 7)

NewWindow: Creates a new window from parameters contained in the function call. (Chapter 7)

DrawGrowlcon: Draws a grow icon and the outline of scroll bars in a standard document window. (Chapter 7)

GetNewControl: Defines a control for one particular window, using parameters from a resource file. This routine is used to define scroll bars. (Chapter 7)

4. MANIPULATING WINDOWS

4. a Activating Windows

SelectWindow: Activates a window, making it the frontmost window on the screen. This is the preferred way to activate a window. (Chapter 7)

SetPort: Makes a window the current grafport. This is an essential routine, since the Macintosh can only draw in the current grafport. (Chapter 8)
4.b Manipulating Window Position in the Plane

**BringToFront:** Changes the position of a window in the plane, making it the frontmost window on the screen. This routine does not affect whether or not a window is visible and does not make it active. (Chapter 7)

**SendBehind:** Changes a window's position in the plane, sending it either behind all other windows on the screen or some other specific window on the screen. This routine does not affect whether or not a window is visible. (Chapter 7)

4.c Manipulating Window Appearance

**SetWTitle:** Changes the title of a window. (Chapter 7)

**ShowWindow:** Makes a previously invisible window visible. If the window is already visible, the routine has no effect. (Chapter 7)

**HideWindow:** Makes a previously visible window invisible. If the window is already invisible, the routine has no effect. (Chapter 7)

4.d Manipulating Window Regions

**InvalRect:** Incorporates a part of a window into the update region, indicating that something has disturbed the appearance of that part of the window and that it must be redrawn. (Chapter 8)

4.e Manipulating Controls

**ShowControl:** Makes a control visible. (Chapter 8)

**HideControl:** Makes a control invisible. (Chapter 8)

5. CLOSING WINDOWS

**CloseWindow:** Removes the window from the screen and deletes it from the application's window list. Should be used when storage for the window record was allocated by the application. (Chapter 7)
**DisposWindow**: Removes the window from the screen and deletes it from the application's window list. Should be used when storage for the window record was placed on the application heap. (Chapter 7)

---

**6. CREATING MENUS**

**GetRMenu**: Defines a menu, using parameters from a resource file. (Chapter 7)

**AddResMenu**: Adds resources of a given type to a menu. This routine is used primarily to add the desk accessories to an "Apple" menu. (Chapter 7)

---

**7. MANIPULATING MENUS**

**7.a Managing the Menu Bar**

**InsertMenu**: Inserts a menu into the menu list, but does not re-draw the menu bar. (Chapter 7)

**DeleteMenu**: Removes a menu from the menu list, but does not re-draw the menu bar. (Chapter 7)

**DrawMenuBar**: Draws the menu bar, displaying the titles of all menus currently in the menu list. (Chapter 7)

**7.b Manipulating Menu Appearance**

**DisableItem**: Disables either one menu item or an entire menu. Disabled items appear immediately, but the menu bar must be re-drawn before a disabled menu will appear with its title dimmed. (Chapter 7)

**EnableItem**: Enables either one menu item or an entire menu. Enabled items appear immediately, but the menu bar must be re-drawn before a newly enabled menu will appear with its title in boldface. (Chapter 7)

**HiLiteMenu**: Removes highlighting from a menu title. (Chapter 8)
8. IDENTIFYING EVENTS

GetNextEvent: Retrieves an event from the event queue. (Chapter 8)

8.a Mouse Down Events

FindWindow: Returns a code indicating the general location of where a mouse down event occurred. If the mouse down event was in a window, it also returns a pointer to that window. (Chapter 8)
MenuSelect: Returns the menu ID and the item number of a menu selection made with the mouse. (Chapter 8)
GetItem: Returns the text of a selected menu item. (Chapter 8)
FrontWindow: Returns a pointer to the frontmost window on the screen. (Chapter 8)
FindControl: Identifies which control, if any, was the site of a mouse down event. This routine also returns the part of the control that posted the event. (Chapter 8)

8.b Key Down Events

MenuKey: Returns the menu ID and menu item selection by the keyboard equivalent of a menu item. (Chapter 8)

8.c Update Events

EraseRect: Erases the contents of a rectangle. Can be used to clear the contents of a window before re-drawing them during the update process. (Chapter 8)
BeginUpdate: Called at the beginning of any code that updates a window. (Chapter 8)
EndUpdate: Called at the end of any code that updates a window. (Chapter 8)
9. HANDLING EVENTS

9.a The Desk Accessories

**SystemTask:** Updates the desk accessories. This routine must be called repeatedly and is therefore generally part of a main event loop. (Chapter 8)

**OpenDeskAcc:** Opens a desk accessory and turns its execution over to the system. (Chapter 8)

**SysEdit:** Handles editing requests in system windows, and in particular, the desk accessories. It should be called whenever an application detects an edit request. If the system cannot process the edit (i.e., the request wasn't for a system window), the function will return a result of false. In that case, the application can process the edit. (Chapter 8)

**SystemClick:** Handles any type of mouse down event in a system window (i.e., a desk accessory). (Chapter 8)

9.b Controls

**TrackControl:** Used to process mouse down events in scroll bars. If the mouse down event has occurred in the thumb of a scroll bar, this routine will continue to drag that thumb as long as the mouse button is held down. If the mouse button was pressed in the up or down arrow, the routine will highlight the arrow until the mouse button is released. Returns a code for the part of the control posting the event. (Chapter 8)

9.c GoAway Boxes

**TrackGoAway:** Highlights the GoAway box as long as the mouse button is depressed in the box. Should be called whenever a mouse down event is detected in a GoAway box. (Chapter 8)
9.d Drag Regions

DragWindow: Drags an outline of a window around the screen until the mouse button is released. The window will be redrawn in its new location. Should be called whenever an application detects a mouse down event in a drag region. (Chapter 8)

9.e Grow Regions

GrowWindow: Drags an outline of the window about the screen as long as the mouse button is held down in the grow icon. Returns the coordinates of the new bottom right of the window. (Chapter 8)
SizeWindow: Re-draws a window with a new size, using the bottom right coordinates returned by GrowWindow. This routine only re-draws the outline of a window; it does not take care of controls or other window contents. (Chapter 8).
MoveControl: Moves a control to a new location in its window. (Chapter 8)
SizeControl: Changes the size of a control. (Chapter 8)

10. HANDLING TEXT

10.a Establishing a Text Edit Record

TENew: Creates a new text edit record. This routine attaches the text edit record to whatever window is the current grafport. (Chapter 9)

10.b Managing Text Edit Windows

TEIdle: Makes the straight-line cursor blink in the active text edit window. Must be called repeatedly for the cursor to blink regularly and should therefore be part of an event loop. (Chapter 9)
TActiv: Activates a text edit window, making the straight-line cursor appear. (Chapter 9)
**TEDeActivate:** Deactivates a text edit window, removing the straight-line cursor. (Chapter 9)

**TEUpdate:** Re-draws the text specified by a boundary rectangle, generally the text edit window's view rectangle. (Chapter 9)

### 10.c Setting the Selection Range

**TEClick:** Positions the straight-line cursor in a text edit window based on the location of a mouse down event. The routine also takes care of extended selections made by dragging the mouse across text or by shift-clicking. (Chapter 9)

**TESetSelect:** Establishes the selection range in a text edit record based on starting and ending character positions passed to the routine as parameters. (Chapter 9)

### 10.d Character Display

**TEKey:** Inserts one character into a text edit record at the current insertion point and displays it on the screen. The character to be inserted generally comes from the keyboard. Therefore, this routine is called in response to a key down event that was not a keyboard equivalent for a menu selection. (Chapter 9)

**TEInsert:** Inserts one or more characters into a text edit record at the current insertion point and displays the new text on the screen. This routine is used, for example, to display text that has been read in from a disk file. (Chapter 9)

**TESetJust:** Sets the justification of the text in the current text edit record. The text edit window should be updated after changing the justification to re-draw the text. (Chapter 9)

### 10.e Editing

**TECut:** Deletes the text in the current selection range and copies it to the Clipboard. (Chapter 9)

**TEDelete:** Deletes the text in the current selection range. The text is not copied to the Clipboard. (Chapter 9)

**TECopy:** Copies the contents of the current selection range onto the Clipboard without deleting it from the text edit record. (Chapter 9)
**TEPaste**: Inserts the current contents of the Clipboard into a text edit record at the current selection point. (Chapter 9)

**10. Scrolling**

**TEScroll**:Scrolls the text in a text edit window. (Chapter 9)

---

**11. DIALOG BOXES**

**GetNewDialog**: Creates a dialog box and displays it on the screen, using a template and an item list from a resource file. (Chapter 9)

**CloseDialog**: Removes a dialog box from the screen and deletes its data structures from memory. (Chapter 9)

**ModalDialog**: Monitors and handles events in modal dialog boxes. (Chapter 9)

---

**12. ALERTS**

**Alert**: Creates an alert from a template and item list in a resource file, monitors and handles events in the alert, and removes the alert from the screen when the user clicks on a push button. (Chapter 9)

---

**13. PRINTING**

**NewHandle**: Returns a handle to a block of memory in the application heap. This routine is used to allocate space for a print record. (Chapter 10)

**DisposHandle**: Releases the block of memory referenced by a handle. This routine is used to delete a printer record. (Chapter 10)

**ProOpen**: Opens the printer resource file. This call must be issued once, before any other Printing Manager calls. (Chapter 10)
PrClose: Closes the printer resource file. This call is issued once, at the end of all printing activity. (Chapter 10)

PrintDefault: Fills a printer record with default information stored in the printer resource file. (Chapter 10)

PrStlDialog: Displays the standard Style dialog box, allows the user to make selections within the dialog box, and fills the printer record with that information. Data from the dialog box is also stored in the printer resource file. (Chapter 10)

PrJobDialog: Displays the standard Job dialog box, allows the user to make selections within the dialog box, and fills the printer record with that information. Data from the dialog box is also stored in the printer resource file. (Chapter 10)

PrOpenDoc: Opens a printing port and makes it the current grafport. This routine is called once before beginning to print a document. (Chapter 10)

PrCloseDoc: Closes a printing port. If draft printing, it issues a form feed to the printer. If spool printing, it closes the spool file. This routine is called once at the end of printing a single document. (Chapter 10)

PrOpenPage: Opens a single page for printing. This routine is called before printing one page. (Chapter 10)

PrClosePage: Closes a single page. If draft printing, the routine issues a form feed to the printer. If printing with single sheets, it prompts the user to insert another sheet of paper. (Chapter 10)

PrPicFile: Images and prints a spool file. (Chapter 10)

---

14. MANAGING COORDINATES

GlobalToLocal: Translates a set of global screen coordinates into coordinates in the local coordinate system of the current grafport. (Chapter 8)

LocalToGlobal: Translates a set of coordinates expressed in the local coordinate system of the current grafport into global screen coordinates. (Chapter 9)

---

15. DRAWING

MoveTo: Moves the pen in the current grafport. If the application is printing, this routine affects the print head. (Chapter 10)
**DrawChar**: Draws a single character on the screen at the current pen position. (Chapter 6)

**DrawString**: Draws a string of characters, beginning at the current pen position and moving to the right. This routine does no text formatting. (Chapter 10)

**DrawText**: Draws a block of text that is stored in main memory, beginning at the current pen position and moving to the right. This routine does no text formatting. (Chapter 10)

**TextBox**: Draws a line of static text in a window. Though the text can be justified in its boundary rectangle, it is not stored in a text edit record and therefore cannot be edited. (Chapter 9)

---

**16. MOVING TEXT**

**BlockMove**: Moves a block of text stored in main memory to another main memory location. (Chapter 6)

---

**17. STRING COMPARISON**

**IUMagString**: Compares two strings of ASCII characters and returns a 0 if the two strings are equal, a -1 if the first string is less than the second and a +1 if the first string is greater than the second. (Chapter 6)

---

**18. FONT CHARACTERISTICS**

**TextFont**: Sets the text font. (Chapter 9)

**TextFace**: Sets the text style (e.g., boldface, underlined, etc.). (Chapter 9)

**TextSize**: Sets the size of the current text font. (Chapter 9)

**GetFontInfo**: Returns information about the current font in the current grafport. (Chapter 10)
19. FILE PROCESSING

**Create:** Creates a new disk file. This routine does not open a file. (Chapter 11)

**GetFileInfo:** Retrieves information stored by the Finder about a specific file. This routine is always called immediately after creating a file. (Chapter 11)

**SetFileInfo:** Sets information about a file for the Finder. The routine is generally called during the file creation sequence, immediately after GetFileInfo. (Chapter 11)

**Write:** Writes data from a data buffer in RAM onto a disk file. (Chapter 11)

**Read:** Reads data from a disk file into a data buffer in RAM. (Chapter 11)

**Close:** Closes a file. (Chapter 11)

**SFGetFile:** Displays the standard "get file" dialog box and allows the user to choose between the files listed. The user can also change disks and drives. The entire process is handled by this routine until the user selects "OK" or "Cancel". (Chapter 11)

**SFPutFile:** Displays the standard "save as" dialog box and allows the user to enter a file name. The user can also change disks and drives. The entire process is handled by this routine until the user selects "OK" or "Cancel." (Chapter 11)

**FlushFile:** Forces the contents of the access path buffer to be written to disk. (Chapter 11)

20. ARITHMETIC

(All routines can be found in Chapter 12)

20.a Integer Binary/Decimal Conversions

**NumToString:** Converts an integer or longinteger into a string of ASCII characters.

**StringToNum:** Converts a string of ASCII characters in an integer or longinteger.

20.b Floating Point

The names of the FP68K and ELEMS68K routines are presented below as the macros defined for them in SANEMacs.TXT.
20.b.1 Addition (A := A + B)

FADDX: Add an extended source operand to an extended destination operand.
FADDD: Add a double precision source operand to an extended destination operand.
FADDS: Add a single precision source operand to an extended destination operand.
FADDC: Add a 64-bit (computational) integer source operand to an extended destination operand.
FADDI: Add an integer source operand to an extended destination operand.
F addl: Add a longinteger source operand to an extended destination operand.

20.b.2 Subtraction (A := A - B)

FSUBX: Subtract an extended source operand from an extended destination operand.
FSUBD: Subtract a double precision source operand from an extended destination operand.
FSUBS: Subtract a single precision source operand from an extended destination operand.
FSUBC: Subtract a 64-bit integer source operand from an extended destination operand.
FSUBI: Subtract an integer source operand from an extended destination operand.
FSUBL: Subtract a longinteger source operand from an extended destination operand.

20.b.3 Multiplication (A := A * B)

FMULX: Multiply an extended source operand by an extended destination operand.
FMULD: Multiply a double precision source operand by an extended destination operand.
FMULS: Multiply a single precision source operand by an extended destination operand.
FMULC: Multiply a 64-bit integer source operand by an extended destination operand.
FMULI: Multiply an integer source operand by an extended destination operand.
**FMULL:** Multiply a longinteger source operand by an extended destination operand.

**20.b.4 Division (A := A / B)**

**FDIVX:** Divide an extended destination operand by an extended source operand.
**FDIVD:** Divide an extended destination operand by a double precision source operand.
**FDIVS:** Divide an extended destination operand by a single precision source operand.
**FDIVC:** Divide an extended destination operand by a 64-bit integer source operand.
**FDIVI:** Divide an extended destination operand by an integer source operand.
**FDIVL:** Divide an extended destination operand by a longinteger source operand.

**20.b.5 Remainder (A := A mod B)**

**FREMX:** Find the remainder of the division of an extended destination operand by an extended source operand.
**FREMD:** Find the remainder of the division of an extended destination operand by a double precision source operand.
**FREMS:** Find the remainder of the division of an extended destination operand by a single precision source operand.
**FREMC:** Find the remainder of the division of an extended destination operand by a 64-bit integer source operand.
**FREMI:** Find the remainder of the division of an extended destination operand by an integer source operand.
**FREML:** Find the remainder of the division of an extended destination operand by a longinteger source operand.

**20.b.6 Rounding**

**FRINTX:** Round an extended operand to an integer.
**FTINTX:** Truncate an extended operand to an integer.

**20.b.7 Arithmetic functions**

**FSQRTX:** Find the square root of an extended operand. \((A := \sqrt{A})\)
**FLOGBX:** Find the base 10 logarithm of an extended operand. \( A := \log_{10}A \)

**FSCALEBX:** Multiply an extended destination operand by 2 raised to an integer power. \( A := A \times 2^B \)

**FCPYSGNX:** Replace an extended operand with the sign of the operand. \( A := \text{sign of } A \)

**FNEGX:** Negate an extended operand \( A := -A \)

**FABSX:** Take the absolute value of an extended operand. \( A := |A| \)

### 20.b.8 Internal type conversion and arithmetic assignment \( A := B \)

**FX2X:** Move an extended source operand to an extended destination operand.

**FD2X:** Move a double precision source operand to an extended destination operand.

**FS2X:** Move a single precision source operand to an extended destination operand.

**FI2X:** Move an integer source operand to an extended destination operand.

**FL2X:** Move a long integer source operand to an extended destination operand.

**FC2X:** Move a 64-bit integer source operand to an extended destination operand.

**FX2D:** Move an extended source operand to a double precision destination operand.

**FX2S:** Move an extended source operand to a single precision destination operand.

**FX2I:** Move an extended source operand to an integer destination operand.

**FX2L:** Move an extended source operand to a long integer destination operand.

**FX2C:** Move an extended source operand to a 64-bit integer destination operand.

### 20.b.9 Binary to decimal conversions \( A := B \)

**FX2DEC:** Convert an extended operand to the canonical decimal format.

**FD2DEC:** Convert a double precision operand to the canonical decimal format.

**FS2DEC:** Convert a single precision operand to the canonical decimal format.

**FC2DEC:** Convert a 64-bit integer operand to the canonical decimal format.

**FI2DEC:** Convert an integer operand to the canonical decimal format.

**FL2DEC:** Convert a long integer operand to the canonical decimal format.
20.b.10 Decimal to binary conversions \((A := B)\)

**FDEC2X:** Convert from the canonical decimal format to an extended operand.

**FDEC2D:** Convert from the canonical decimal format to a double precision operand.

**FDEC2S:** Convert from the canonical decimal format to a single precision operand.

**FDEC2C:** Convert from the canonical decimal format to a 64-bit integer operand.

**FDEC2I:** Convert from the canonical decimal format to an integer operand.

**FDEC2L:** Convert from the canonical decimal format to a long integer operand.

20.b.11 Comparisons (use in place of CMP)

**FCMPX** and **FCPXX:** Compare two extended operands and set the condition codes.

**FCMPE** and **FCPXD:** Compare an extended operand with a double precision operand and set the condition codes.

**FCMPS** and **FCPXS:** Compare an extended operand with a single precision operand and set the condition codes.

**FCMPC** and **FCPXC:** Compare an extended operand with a 64-bit integer operand and set the condition codes.

**FCMPI** and **FCPXI:** Compare an extended operand with an integer operand and set the condition codes.

**FCMPL** and **FCPXL:** Compare an extended operand with a long integer operand and set the condition codes.

20.b.12 Branch on condition codes (use in place of Bcc instructions)

**FBEQ** and **FBEQS:** Branch if equal.

**FBLT** and **FBLTS:** Branch if less than.

**FBLE** and **FIBLES:** Branch if less than or equal.

**FBGT** and **FBGTS:** Branch if greater than.

**FBGE** and **FBGES:** Branch if greater than or equal.

**FBNE** and **FBNES:** Branch if not equal.
20.b.13 Elementary functions

**FLNX**: Find the natural logarithm of an extended operand. \( A := \ln A \)

**FLOG2X**: Find the base 2 logarithm of an extended operand. \( A := \log_2 A \)

**FLN1X**: Find the natural logarithm of an extended operand plus 1. \( A := \ln (1 + A) \)

**FLOG21X**: Find the base 2 logarithm of an extended operand plus 1. \( A := \log_2 (1 + A) \)

**FEXPX**: Raise \( e \) to an extended operand power. \( A := e^A \)

**FEXP2X**: Raise 2 to an extended operand power. \( A := 2^A \)

**FEXP1X**: Raise \( e \) to an extended operand power and subtract 1. \( A := e^A - 1 \)

**FEXP21X**: Raise 2 to an extended operand power and subtract 1. \( A := 2^A - 1 \)

**FXPWRI**: Raise an extended operand to an integer operand power. \( A := A^B \)

**FXPWRY**: Raise an extended operand to an extended operand power. \( A := A^B \)

**FCOMPOUND**: Use extended operands to compute compound interest. \( A := (1 + \text{Rate})^{\#\text{Periods}} \)

**FANNUITY**: Use extended operands to compute an annuity. \( A := (1 - (1 + \text{Rate})^{-\#\text{Periods}})/\text{Rate} \)

**FSINX**: Find the sine of an extended operand. \( A := \sin(A) \)

**FCOSX**: Find the cosine of an extended operand. \( A := \cos(A) \)

**FTANX**: Find the tangent of an extended operand. \( A := \tan(A) \)

**FATANX**: Find the arctangent of an extended operand. \( A := \arctan(A) \)

**FRANDX**: Find the next random number, using an extended operand as a seed. \( A := \text{rand}(A) \)
Absolute Address: A main memory address specified by its numeric address. For example, $001A is an absolute address. While an application can only work from absolute addresses, programmers can use symbolic addresses in their source code, leaving the translation to absolute addresses to the assembler and linker.

Access path: A data structure describing how the Macintosh should find a disk file. An access path is created every time a file is opened. The Macintosh will support 12 access paths at any one time, though only one access path per file can be open for writing.

Access path buffer: A RAM buffer that is used as temporary storage by the access path.

Active window: The front-most window on the screen. An application can only work in an active window. Active windows are highlighted in some way.

Address: The location of a byte in a computer's main memory. The bytes in a computer's main memory are numbered sequentially beginning with 0. Each byte therefore has a unique number known as its address.

Address register: A general purpose register within the Macintosh's microprocessor. The Macintosh has eight address registers, though some are used by the system for special purposes. A5 holds the start of the applications globals area; A7 is used as the stack pointer.

Addressing mode: A method for specifying the main memory address of a piece of data. The Macintosh has 13 addressing modes.
Alert: A Macintosh window that is displayed to warn the user that continuation of a particular action could cause damage or that some error has already occurred. Alerts contain text, icons, and buttons to either continue or cancel the action.

Applications globals area: A portion of RAM used for an application's data storage. The size of the application globals area is not fixed. Rather, it is set during the linking process so that only the exact amount of space the program requires will be allocated at run-time. Assembly language programmers should allocate space for all read/write data in the applications globals area.

Application heap: The portion of RAM available to an application program and its constants. Though it is possible to place read/write data storage in the application heap, it is better to avoid doing so whenever possible. (Interactions with the Printing Manager may cause exceptions to this rule.)

ASCII: The American Standard Code for Information Interchange. ASCII is a binary coding scheme that is used to represent characters within a computer. Standard ASCII requires 7 bits to represent the full range of alphanumeric characters. The Macintosh generates extra characters by using 8 bits.

Assembler: A program that translates a programmer's assembly language source code into machine language.

Assembler directive: An instruction in an assembly language source program that gives directions to the assembler. Assembler directives control the assembly process; they do not become a part of the object code.

Assembly language: A programming language that uses mnemonic codes to substitute for the machine language version of a computer's instruction set.

Asynchronous file operations: File operations that permit the application to continue with other activities while the file operation is in progress.

Band: A strip from a printed page. Since it takes a great deal of memory to image and print a spooled print file, each page is broken up into bands which can then be printed separately. Bands may run horizontally or vertically across the page, depending on the orientation of the printed page.

Binary: The base 2 numbering system used to represent quantities, instructions, and characters in a computer.

Bit: A contraction of "binary digit." A bit represents one binary place in a code or quantity. It can take only two values — 0 or 1.
Boot (a computer): To start the computer, either by turning on the power or pressing the Reset switch. It is also possible to reboot the Macintosh by issuing a RB (reboot) command to a debugger.

Boundary rectangle: A set of four coordinates that describe the top left and bottom right corners of a rectangle. The coordinates may be expressed in terms of the screen's global coordinate system or in terms of the local coordinate system of a specific window, depending on the situation. For example, window definitions require global coordinates, but control definitions require the local coordinates of the window in which the controls will appear.

Buffer: A temporary holding area for data. Buffers are generally used to reconcile the speed differences between slow I/O devices and the much faster CPU. For input, for example, a disk drive fills a main memory buffer at its own speed. The CPU can be doing other things while the disk is working. When the buffer is full, the CPU empties it at electronic speeds.

Bus: An electronic pathway that connects the parts of a computer. Buses carry data, addresses, and control signals between the CPU, main memory, and peripheral devices such as disk drives and printers.

Byte: Eight bits viewed as a whole.

Canonical decimal format: An intermediate numeric format used by the Macintosh. It is produced by scanning an ASCII string of characters. Numbers expressed in the canonical decimal format can then be converted into a variety of binary numbers which can be used in mathematical operations.

Clear: 1) Give a bit or a group of bits a value of 0-2) a text editing operation that deletes the contents of the current selection range from the document without affecting the clipboard.

Clipboard: A temporary storage area used by text editing routines to hold text from cut operations. Cut takes the contents of the current selection range and places it on the clipboard, deleting it from the document and erasing the previous contents of the clipboard. Copy also places the current selection range on the clipboard, but does not delete it from the document. Paste takes the contents of the clipboard and inserts it into the document at the current insertion point; the contents of the clipboard are not disturbed.

Compiled language: A programming language (usually a high-level language) that is translated to object code prior to run-time. Compiler output is a machine language file which generally must be linked to run-time libraries before execution.
**Condition codes:** see **Status register.**

**Conditional branch:** An assembly language instruction that checks one or more flags in the status register and executes a branch if the condition specified by the particular instruction is true. If the condition is false, program execution continues with the next sequential instruction. 68000 conditional branch instructions have the general form **Bcc,** where the **cc** is replaced with two letters that represent the condition to be tested.

**Control:** A graphic device that helps to regulate program function. Controls include scroll bars, push buttons, radio button, and check boxes.

**Copy:** A text editing operation that takes the contents of the current selection range and writes it to the clipboard. The document itself is unaffected.

**CPU (central processing unit):** The brain of a computer. The CPU is the site of instruction decoding and execution. When the CPU is placed on a single silicon chip, it is referred to as a microprocessor.

**Creator:** The type of application that created a file. A file's creator is a four-character string stored with the file itself. Unless a file type is explicitly set, an application created by the MDS will have a file type of APPL. The creator for all files created by such an application will therefore be APPL.

**Cursor:** In general, some character on a computer screen (e.g., a blinking line, underbar, or box) that indicates where the next input will appear. On the Macintosh, the cursor is attached to the mouse. Moving the mouse moves the cursor. Macintosh cursors take a variety of shapes, including an arrow, an I-beam, and a wrist watch.

**Cut:** A text editing operation that takes the contents of the current selection range and copies it to the clipboard, at the same time deleting it from the document.

**Data fork:** The part of a Macintosh file that contains data.

**Data register:** A general purpose register within the Macintosh's microprocessor. The Macintosh has eight data registers.

**Debugger:** A program designed to aid a programmer in identifying logic errors within an assembly language program. A debugger permits step-by-step program execution, displays the contents of the CPU's registers, disassembles instructions, etc.

**Decrement:** To decrease by some fixed quantity. If the quantity is not specified, it is assumed to be 1.
Dialog box: A Macintosh window used to collect information from the user or to freeze program action until the user is ready to continue.

Direct access: A method of file processing. Files created for direct access have fixed field lengths, allowing an application to go directly to any record at any time, regardless of the location of the record most recently read or written. Records can be processed in random order.

Direct cursor addressing: Having the capability of moving the cursor anywhere on the screen or printed page at any time, regardless of the cursor's previous position.

Drag region: The title bar of a window except the GoAway box. It is used to move a window around the Macintosh screen.

Edit text: Text that can be edited using any of the Macintosh's editing routines: cut, copy, paste, or clear.

Effective address: The main memory location of an operand for an assembly language instruction. Effective addresses are specified by using one of the Macintosh's 13 addressing modes.

Equate file: A text file that contains a set of EQU statements. Each EQU associates a symbolic address with a constant that is useful in Macintosh programming.

Event: A system activity that the Macintosh can recognize. Events include pressing and releasing the mouse button, pressing and releasing keys, inserting disks, etc.

Event mask: A word whose bits can be selectively set to control which types of events are retrieved from the event queue.

Event queue: An ordered list of events as they occur. The event queue is maintained by the operating system in first in, first out order. In other words, the first event posted to the event queue will be the first event processed.

Excess notation: A method of storing floating point exponents. An excess value is selected so that when added to the smallest possible exponent, it will raise that exponent value to 0. All exponents are then stored with the excess value added to them. All exponents can therefore be kept as positive integers without having to resort to 2's complement representation.

Exponent: The power to which some base number is raised. For example, in the expression \(10^497\), 10 is the base number and 497 is the exponent.
**Fixed point number:** A number that includes a decimal point (or binary point if the number is in base 2) that does not move. For example, 3.44 is a fixed point number.

**Floating point number:** A number expressed as a mantissa multiplied by a base raised to some power. For example, $3.333 \times 10^9$ is a floating point number. Because the exponent can change, the decimal point (or binary point, if the base is 2) is said to "float."

**Fork:** Part of a Macintosh file. Macintosh file's have two forks — a data fork for storing data and a resource fork for storing resources and program code.

**GoAway box:** A box that appears at the left of a title bar. Clicking the arrow cursor in the GoAway box will close the window.

**Grafport:** A contraction of "graphics port." A graphics port is a rectangle in which the Macintosh can draw. Grafports form the basis for Macintosh windows.

**Hexadecimal:** The base 16 numbering system. Since four binary digits can be represented by a single hexadecimal digit, hexadecimal is often used as a shorthand for binary.

**High-level language:** A programming language that looks very much like English. BASIC, Pascal, FORTRAN, PL/1, and COBOL are all high-level languages.

**Highlighting:** Changing the standard coloration of something on the Macintosh screen to draw attention to it in some way. For example, text editing selection ranges are highlighted by displaying them as white characters on a black background.

**High-order:** The upper-half of a group of bits. For example, in a word where the bits are numbered 0 through 15, bits 7 through 15 are the high-order byte. In a longword where the bits are numbered 0 through 31, bits 16 through 31 are the high-order word.

**Hung:** A state in which the computer appears to sit still and do nothing. Many things can cause a computer to hang, but most often it is some sort of infinite loop.

**I/O Buffer:** see Buffer

**Icon:** A small picture that the Macintosh uses to represent an object or program function.
Increment: To increase by some fixed quantity. If the quantity is not specified, it is assumed to be 1.

Insertion point: The place in a document where new characters and/or graphic images are inserted.

Instruction: A single command that a computer can understand and execute.

Instruction set: All the commands that a computer can understand and execute. Each type of microprocessor has its own unique instruction set.

Interpreted language: A programming language (usually a high-level language) that is translated to machine language while the program is being run. No permanent object code is ever generated. Statements that are executed repeatedly are translated each time they are executed.

Interrupt: A signal generated by a peripheral device such as a disk drive and sent to the CPU. The interrupt tells the CPU that the device is in need of attention. The CPU will stop whatever it is doing to take care of the device.

Keyboard equivalents: The pairing of the cloverleaf key with any other printing key on the keyboard as a substitute for using the mouse to make a selection from a pull-down menu.

Launch: To run a Macintosh application.

Least significant digit: In an integer, the digit in the one's place. When a number contains a factional portion, the least significant digit is the right-most non-zero digit.

Linker: A program that pulls together the various parts of an application to create an executable application. The Linker also completes the process of setting the size of the applications globals area.

Longword: On the Macintosh, a group of 32 bits.

Low-order: The lower-half of a group of bits. For example, in a word where the bits are numbered 0 through 15, bits 0 through 7 are the low-order byte. In a longword where the bits are numbered 0 through 31, bits 0 through 15 are the low-order word.

Machine language: A computer can only understand instructions that are written in machine language. Machine language consists of a sequence of binary codes. Since it is so very difficult for humans to write programs that are
comprised of nothing but a series of 0's and 1's, most programs are written in either assembly language or a high-level language. The programs must then be translated into machine language before they can be executed by a computer.

**Macro:** A short block of code defined within a program and given a name. The name of the macro is then used in the source program instead of the macro code. During assembly, the macro code is inserted everywhere the name of the macro appears.

**Mantissa:** The significant digits of a floating point number. The first digit of a mantissa will always be non-zero. For example, in the floating point number $3.9746123 \times 10^7$, $3.9746123$ constitutes the mantissa. The number, therefore, has eight significant digits.

**Mark:** A pointer in a Macintosh file that indicates the position of the next byte to be read from or written to.

**Menu:** A list of options from which a user can select. Macintosh menus descend, or "pull-down", from the menu bar.

**Menu bar:** The top line on the Macintosh screen. It contains the names of all menus currently available to the user. The left-most menu is the Apple menu which supports the standard desk accessories. Directly to its right will be found the File and Edit menus.

**Menu list:** A list maintained by the Macintosh that contains all menus that are displayed in the menu bar. Menus are displayed in their order in the list. An application can control which menus appear in the menu bar by inserting and deleting menus from the menu list.

**Microcomputer:** Commonly, a computer small enough to fit on a desk top. A microcomputer must have a microprocessor, RAM, enough ROM to boot the machine, buses for data and address transfer, a clock, and some provision for I/O.

**Microprocessor:** A CPU (central processing unit) contained on a single chip. The microprocessor is the place where instructions are decoded and executed. It is often called the "brain" of the computer.

**Mnemonic:** A group of two to five letters that stand for a machine language instruction. The collection of letters has some relationship to the name of the instruction. For example, JSR stands for Jump to Subroutine.

**Modal dialog box:** A dialog box that restricts the user to working within the box while the box is present on the screen, such as the dialog boxes that appear when a user selects Print from a File menu.
**Modeless dialog box:** A dialog box that permits the user to work outside the dialog box while the box is present on the screen. An example is the dialog box that appears when a user selects Find from a Search menu.

**Most significant digit:** The left-most non-zero digit in a number.

**Object code:** The machine language version of an assembly or high-level language program.

**Octal:** The base 8 numbering system. Since three binary digits can be represented by one octal digit, octal can be used as a shorthand for binary. It is less commonly used than hexadecimal.

**Op code:** A binary code that represents an assembly language instruction.

**Operand:** A piece of data required by an assembly language instruction.

**Operating system:** A program that controls the operation of the computer. Generally, operating systems for single-user microcomputers provide the means to boot the computer, execute programs, and manage files (delete, re-name, etc.).

**Parameter:** A piece of data used as input to or output from a Pascal subprogram.

**Parse:** To break a sentence down into its constituent parts. In computers, parsing generally refers to analyzing a program statement to determine its elements. It also refers to scanning and breaking down a string of ASCII characters so they can be transformed into some other format (i.e., the canonical decimal format).

**Path reference number:** A quantity that identifies an access path to a file.

**Paste:** A text editing operating that takes the contents of the clipboard and inserts it into a document at the current selection point. The contents of the clipboard are unaffected.

**Patch:** To modify existing program code by changing a small portion of it. Patching usually refers to making modifications to the binary (machine language) version of a program.

**Pixel:** Short for “picture element.” A pixel is one dot on the Macintosh’s screen.

**Program counter:** A 32-bit register in the Macintosh’s microprocessor. The program counter always holds the main memory address of the next program instruction to be executed.

**Prompt:** A piece of static text that tells the user what data should be entered.
RAM (random access memory): A computer’s main memory. An application can both read from and write to RAM. RAM is volatile — when electrical power is removed its contents are lost.

Region: An area within a grafport that can be bounded by a rectangle but is not necessarily rectangular in shape.

Register: A special storage location within a microprocessor. The Macintosh’s general purpose registers are 32 bits wide; each can hold a longword.

Relocatable code: A block of object code that is independent of any fixed main memory location. Relocatable programs can theoretically be run regardless of where they are loaded into memory. The MDS Assembler creates a relocatable object code module which is then tied to a specific place in memory by the Linker.

Resource: An entity used by the Macintosh. In some instances, the Macintosh views the code of an application as a single resource; but more generally, the term refers to something much smaller, such as a window, a menu, an icon, a desk accessory, etc.

Resource file: A file that contains definitions and templates for resources. Resource files are created with a text editor and then translated to machine language by the resource compiler, RMaker.

Resource fork: The part of a Macintosh file that stores resource definitions and program code.

Resource template: An entry in a resource file that contains the parameters that describe a particular resource. The resource type must already have been defined. For example, the resource type WIND is pre-defined to describe a window. A resource file therefore contains only a window template — the data necessary to generate a window.

ROM (read only memory): A type of computer memory from which an application can only read. ROM cannot be modified and is non-volatile — it retains its contents when electrical power is removed.

Run-time library: A set of standard programs, usually handling I/O, that are used by compiled programs. The object code produced by compiler cannot be executed without first being linked to one or more run-time libraries.

Scroll: To move text or graphics so that a different portion of a large document appears in a window on the Macintosh screen.

Selection range: A group of contiguous characters in a block that will be affected
by editing operations such as cut, copy, and paste. The selection range is highlighted by displaying white characters on a black background.

**Sequential access:** A method of file processing. Files created for sequential access have either variable or fixed record lengths. The records are processed in order, generally beginning at the start of the file. Sequential processing proceeds either to the "next" or "prior" record; it is not possible to move randomly through the file.

**Set:** Give a bit or group of bits a value of 1.

**Significant digits:** The part of a number that conveys value rather than magnitude. For example, in the number 0.00009994, the significant digits are 9994. The leading zeros contribute only to the magnitude of the number, not to its exact value. The first significant digit in a number is most often the first non-zero digit from the left. Generally, the more significant digits retained in a number, the greater the accuracy of that number.

**Source code:** The version of a program created by a programmer, regardless of the language in which it is written. Source code must be translated to object code (machine language) before it can be executed.

**Spooling:** In general, using some form of auxiliary storage (usually a disk) as intermediate storage for an I/O operation. In particular, the Macintosh uses spooling for printing. An image of a printed document is stored on disk to be printed at a later time.

**Stack:** A special area in RAM used for temporary storage. The Macintosh's stack is 32 bits wide. Longwords are placed on top of one another on the stack; access is in last in, first out order. Many of the Macintosh's built-in routines take their parameters from the stack.

**Stack pointer:** A register that contains the address of the top of the stack (the address of the last longword placed on the stack). The Macintosh uses register A7 for that purpose.

**Static text:** Text that is for display purposes only. It cannot be edited.

**Status register:** A 16-bit register within the Macintosh's microprocessor. The bits in a status register function independently as flags to signal a variety of conditions within the computer. The flags in the status register are also referred to as "condition codes".

**Symbolic address:** In an assembly language program, a group of characters used in place of an absolute main memory address. Symbolic addresses can be attached to program instructions, constants, and storage locations. The
assembly and linking process translates the symbolic addresses into absolute addresses.

**Synchronous file operations:** File operations that force the application to wait for the file operation to finish before proceeding.

**System byte:** The high-order byte of the Macintosh's status register. Bits 8 through 15 of the status register are used only by the operating system and are not referenced by application programs.

**System Dispatch Table:** An array in RAM that contains the actual location of ToolBox and Operating System routines. When the Macintosh traps calls to those routines, it translates them into references to the System Dispatch Table where the address is found. The Table is kept in ROM but loaded into RAM at system start-up. Because the Table is in RAM when the system is running, it can be patched to install custom routines.

**Title bar:** The top of a window. The window's title is centered in the title bar. An optional GoAway box may appear at the very left.

**ToolBox:** A collection of programs supplied with the Macintosh that support graphics and the features of the Macintosh user interface. Most of the ToolBox is in ROM.

**Trap:** A function of the Macintosh operating system that catches ("traps") binary instruction codes that are not a part of the standard 68000 instruction set. The Macintosh uses the trap mechanism to extend the Macintosh's instruction set by adding instructions which call the ToolBox and operating system routines.

**True magnitude form:** A representation of an integer quantity where the binary value stored in the computer is the same as the actual value of the number.

**Two's complement:** The number which, when added to a binary number, will produce a result of 2.

**Two's complement form:** A representation of an integer quantity where the binary value stored in the computer is the two's complement of the actual value of the number.

**Two's complement system:** A method for representing integer quantities within a computer. Negative numbers are stored in their two's complement form; positive numbers are not converted but are left in their true magnitude form.

**Unconditional branch:** An assembly language instruction that executes a branch under all circumstances. No condition codes are checked. The 68000 has two unconditional branch instructions — **BRA** and **JMP**.
User byte: The low-order byte of the Macintosh's status register. An application often consults the bits in the user byte to determine the result of a particular program instruction.

Value parameter: A parameter that is used only as input to a Pascal program. Even the value of the parameter is changed in the subprogram, it will nonetheless retain its original value as far as the main program is concerned.

Variable parameter: A parameter that can be used for both input to and output from a Pascal subprogram. Any data that are to be returned to a main program must be declared as variable parameters. The only exception to this rule is for the results of functions, which in Pascal are returned across an assignment operator. Assembly language function results are either returned on the stack or in a data register.

Volume: Either a single floppy disk or a partition on a hard disk.

Window: A rectangle on the Macintosh screen. Windows are used to display text and graphics, to collect data essential to program function, and to warn the user about the consequences of specific actions. Virtually all user interaction with an application takes place within windows.

Word: On the Macintosh, a group of 16 bits. Word size does vary from computer to computer.
To learn more about microcomputers and how they work:


To learn more about the 68000 microprocessor and how to program it:


To learn more about the Macintosh ToolBox and operating system routines:

Index

A
absolute data addressing 44-45
absolute long address addressing 44-45
absolute short address addressing 44
activate events
defined 194
in application windows 223
in text edit windows 239-243
ADD 106-107
address register direct addressing 37
address register indirect addressing 37-38
address register indirect with displacement addressing 40-42
address register indirect with index addressing 42-43
address register indirect with postincrement addressing 38-40
address register indirect with predecrement addressing 40
address registers 25-26
addressing modes 33-48
AddMenu 187
Alert 271
alerts
creating/disposing 271-272
definition 13
handling events in 271-272
item lists 266-269
resource file template 265-266
AND 110-112
AppleTalk Manager 147
arithmetic
floating-point numbers 362-366
integers (see: ADD, SUB, MULU, MULS, DIVU, DIVS)
arrays
in RAM 118-120
user-defined data type 138-139
Assembler codes 18-21
Assembler 58-61
assembler directives 82-86
assembling a program 58-61
automatic program assembly and linking 72-74
binary search 123-129, 150-156
BlockMove 151-152, 155
boolean data 137
BRA 97-98
BringToFront 177
BSR 115
BTST 239-240
buttons
definition 10-11
in alerts and dialog boxes 266-269
C
character data 136-137
check boxes
definition 10-11
in alerts and dialog boxes 266-269
Close 339
CloseDialog 270
CloseWindow 182
CLR 99
CMP 94-95
compiled languages 2
Control Manager 145
tools
definition 10
in alerts and dialog boxes 266-269
scroll bars 179-182, 209-212, 214-220
Create 330
cursors
definition 4-5
moving 243-245, 291-292
setting 339-340
D
data register direct addressing 36
data registers 25-26
DC 84-85
DCB 85
data files
adding to a menu 187
defining a menu for 186-187
editing in 206-207
identifying menu selections for 205-206
mouse down events in 208
Desk Manager 145
destination rectangle 238
device drivers 146, 147
Device Manager 146
dialog boxes
creating/disposing 269-270
definition 13-14
handling events in 270-271
item lists 266-259
resource file template 264
Dialog Manager 145
DisableItem 189
Disk Initialization Package 147
displaying text (see also: editing text) 291-299
DisposeHandle 301
DisposeWindow 182
DIVS 109-110
DIVU 109-110
draft printing 275-276
drag region
definition 9-10
mouse down events in 213-214
DragWindow 213
DrawChar 86
DrawGrowable 179
DrawMenu 188
DrawString 292-293
DrawText 295-296
DS 85-86
E
editing text (see: text editing)
Editor 55-58
EnableItem 189
END 86
EndUpdate 214, 220, 222
touch lists 266-269
fonts 258
modify word 196
equates
definitions 196
files, packing 283-284
EraseRect 214, 217
error codes
File Manager 325
Printing Manager 286
system 70-71
events
and program structure 201
definition 193
event loop 198-201
event masks 196-197
event queue 194, 196-197
event record 194-195, 198
retrieving 198
types 193-194
Executive 72-74

F
File Manager
definition 146
error codes 325
files
closing 339
creating 330-332
direct access
definition 322-323
reading 338-339
writing 325-336
opening 332-333, 340-342
parameter blocks 323-328
saving 342-344
sequential access
definition 322-323
reading 337-338
writing 332-325
FindControl 210
FindWindow 202
Floating-Point Arithmetic Package
147, 350-355, 357-369
floating-point numbers
arithmetic operations 362-366
conversions
from ASCII string to canonical 350-355
from binary to canonical 367-368
from canonical to ASCII string 368-377
from canonical to binary 359
format of 357-359
FlushEvents 197
Font Manager 145
fonts
font description 290
font size 258
font style 258
font type 257-258
FrontWindow 209
functions 141-144

G
GetCursor 339
GetFileInfo 330-331
GetFontInfo 290
GetItem 205
GetNewControl 181

GetNewDialog 269
GetNewWindow 173
GetNextEvent 198-199
GetRMenu 186
GlobalToLocal 143-144, 210, 244
goAway box
definition 9-10
mouse down events in 212-213
grafports 160, 222
grow icon
definition 9-10
mouse down events in 214-220
GrowWindow 215

H
handles 137-138, 284, 301
hexadecimal numbering system 22-23
HideControl 217
HideWindow 177
HILiteMenu 204
I
immediate data 47-48
INCLUDE 84
InitCursor 237, 240
InitDialogs 237, 261-262
InitFonts 237
InitGraf 160
initialization sequence 237
InitMenus 186
InitWindows 160
InsertMenu 142, 187
integers 104, 134-135
International Utilities Package 146,
150-151
interpreted languages 2
InvalRect 215
iteration 92-93, 105
IUIDMagString 150
IUMagString 150

J
JMP 97-98
JSR 115

K
keyboard equivalents 220-221
Key down events
as equivalents for menu selections
220-221
defined 194

L
LEA 91-92
line 1010 unimplemented instructions
149
Linker 63-65, 68
linking a program 63-65, 68
LocalToGlobal 244
looping 92-93, 105

M
machine language 2
Macintosh 68000 Development
System 53-55
macros 360-362
Memory Manager 146
MenuKey 221
Menu Manager 145
menus
and program structure 190
creating 186-187
definition 4-9
disabling/enabling items 189
highlighting/unhighlighting 204
menu bar
adding/deleting menus 187-189
displaying 188
mouse down events in 203-205,
207-208
resource file template 183
MenuSelect 143, 203-204
microprocessors 2
ModalDialog 271
mouse down events 193, 202-220
MOVE 87-90
MoveControl 218
MOVEM 148
MoveTo 292
MULS 109
MULU 109

N
NewHandle 283
NewWindow 168
NOT 113
null events 193
NumToString 348

O
object code 2
octal numbering system 23-24
Open 332-333
OpenDeskAcc 206
OpenResFile 172
Operating System Event Manager 146
Operating System Utilities 147
OR 112-113

P

Package Manager 146
packing an equates file 263-264
parameters 141-144
PEA 90-91
pointers 137-138
PrClose 287
PrCloseDoc 289
PrClosePage 299
PrintDefault 287
printing
access to Printing Manager routines 276-279
closing a page 299
computing page size 290-291
draft (definition) 275-276
imaging and printing a spool file 300-301
job dialog 288
moving the pen 291-292
opening a page 289
opening/closing a document 288-289
opening/closing Printing Manager 287
printer record 279-283, 284, 287-288, 301
sequence of Printing Manager routines 285-286
spooled (definition) 275-276
style dialog 288
printing control directives 86
Printing Manager 147, 276-279
error codes 286
PrJobDialog 288
procedures 141-144
program counter 31
program counter relative addressing modes 46-47
program counter with displacement addressing 46
program counter with index addressing 46-47
PrOpen 287
PrOpenDoc 288
PrOpenPage 289
PrStdDialog 288
push buttons (see: buttons)

Q

QuickDraw
definition 145
initializing 160
quick immediate data 48

R

radio buttons (see: buttons)
Read 336-338
real numbers 135
records
user defined data type 139-141
register direct addressing modes 36-37
register indirect addressing modes 37-43
registers 25-29
resource files 171, 175-176, 180, 183-184
Resource Manager 144-145
RTS 115
running an application 68
run time libraries 2
run time system errors 68, 70-72

S

Scc 98-99
Scrap Manager 145
scroll bars
creating 181-182
mouse down events in 209-212
moving 214-220
resource file template 180
searching 123-129, 150-156
Segment Loader 146
SelectWindow 178
SendBehind 177
SetCursor 339
SetFileInfo 331-332
SetPort 222
SetWindowTitle 176-177
SF 99
SFGetFile 340-342
SFPutFile 342-344
ShowControl 220
ShowWindow 177
SignedByte 137
SizeControl 219
SizeWindow 216
sorting 117-118, 120-123, 150-156
spooled printing 275-276, 300-301
ST 99
stack 29-30
Standard File Package 340-344
Standard Utilities Package 146
statement format 55-56
status register 25, 27-29
straight insertion sort 117-118, 120-123
string data 136-137
STRING_FORMAT 136-137
StringToNum 349
SUB 107-108
subroutines 114-115, 355-356
SWAP 112
symbolic addresses 45
SysEdit 206-207
SystemClick 208
system dispatch table 31, 149
System Error Handler 147
system errors 68, 70-72

T

TEInit 237
TEActivate 240
TEClick 244
TECopy 248
TECut 248
TEDeactivate 240
TEDelete 248
TEDispose 251
TEIdle 243
TEInsert 246
TEKey 246
TEPaste 248
TEScroll 260-261
TESetJust 259
TESetSelect 245
TEUpdate 255
text editing
activating/deactivating 239-243
blinking cursor 243
character insertion 245-248
copy 248
cut 248
definition 11-12
delete 248
justification 259
paste 248
selection range 243-245
scrolling 260-261
static text 251-254
text edit record 235-236, 238-239, 251
updating 255-257
TextBox 252-253
TextEdit 145, 235-261
TextFace 258
TextFont 257-258
TextSize 258-259
ToolBox Event Manager 145
ToolBox routines 144-146
ToolBox Utilities 145
**TrackControl** 211
**TrackGoAway** 212-213
Transcendental Functions Package 147
**TRAP** 149-150
traps 149-150
two's complement 103-105

**U**
unimplemented instructions 149
update events
defined 194

in application windows 221-223
in text edit windows 255-257
user-defined data types 137-138

**V**
Vertical Retrace Manager 147
view rectangle 238

**W**
Window Manager 145
windows
boundary rectangles 160-163
changing titles 176-177
closing 182-183, 212-213
creating 168-175
definition 9-11
making active 178
making visible/invisible 177
mouse down events in 209-212
moving in the plane 177-178
moving on the screen 213-214
resource file template 171
scrolling 179-182
sizing 178-179, 214-220
types 163-166
window record 167-168
**Write** 333-336