PROGRAMMING THE MACINTOSH
An Advanced Guide

William B. Twitty



CHAPTER 8 **CREATING AND USING MENUS 151**
Designing Menus **152**
The Anatomy of a Menu **153**
Building Menus **155**
Using Menus **162**
A Skeleton Menu Program **167**
Running the Skeleton Menu Program **175**

CHAPTER 9 **DO IT IN A WINDOW 179**
What's in a Window **180**
Creating Windows **185**
Writing and Drawing in Windows **192**
Mouse Events **196**
Activate Events **199**
Update Events **200**

CHAPTER 10 **GETTING INTO THE HARDWARE 221**
Nuts and Bolts **222**
Chips and Such **222**
Memory **223**
The Display **228**
Peripherals and Interfaces **228**
Interrupts **230**
The 6522 **232**
The Keyboard **237**
The Mouse **240**
The Sound Generator **243**
The Printer **244**

CHAPTER 11 **THE 68000 247**
Architecture **248**
Addressing **251**
Input/Output **256**
Exceptions and Interrupts **257**
Condition Codes **260**
Instruction Set **261**

CHAPTER 12 **ASSEMBLER 271**
The Assemblers **272**
Developing Applications **273**
MacASM **275**
The Macintosh 68000 Development System **276**
Exec **282**
Edit **283**
Asm **287**
Link **292**

PackSyms—the Symbol Packer **295**
RMaker—the Resource Compiler **298**
The Debuggers **300**

CHAPTER **13 DISKS 305**
The Disk Drives **306**
The Disk Format **309**

CHAPTER **14 COMMUNICATING 315**
Communicating with the Mac **316**
Communications Basics **316**
The Communications Ports **320**
The Serial Communications Driver **322**
AppleTalk **324**

APPENDIXES 327
A Pascal Language Comparison **328**
B Microsoft BASIC Commands and Reserved Words **334**
C Macintosh BASIC Commands and Reserved Words **340**
D Macintosh Compilers **346**
E Toolbox Routines **348**

GLOSSARY 361

INDEX 368

CHAPTER

1 INTRODUCTION

Programming the Macintosh

A Bit of History

How the Macintosh Is Different

How the Macintosh Is the Same

What's at the Core of the Macintosh

Languages

PROGRAMMING THE MACINTOSH

Let's face it, the Macintosh is different. Writing programs for the Macintosh isn't like programming for other machines. There is a whole range of unique design considerations to take into account before you write the first line of code. If you want your program to work like other Macintosh programs and take full advantage of the machine's capabilities, you need to be familiar with the Macintosh system software, both the operating system and the programs that support the unique user interface. While CP/M and MS-DOS have roughly 40 system calls, the Macintosh has over 400 system routines. So you can see that there's a lot to learn.

Why all the complexity? To support the Macintosh user interface, which makes things very easy for the user. For the programmer, however, there's a lot to learn in order to design and write programs for the Mac. The value of the Macintosh lies in its user interface, so it's important for a programmer to get inside the Macintosh and understand how it works.

Why was a new user interface built? How did all of this evolve? Let's take a look at how the Mac got to be the way it is.

A BIT OF HISTORY

It started in Palo Alto, California, at the eastern edge of the Santa Cruz mountains, where Xerox Corporation's Palo Alto Research Center (PARC) sits among rolling green hills.

Xerox formed PARC to do research into computer design and use. There was never a mandate that the research center produce a marketable product. In fact, when some of the developments at PARC appeared to have enormous market potential, it seemed impossible for Xerox to convert them into products and bring them to market.

At PARC, a group of researchers was interested in studying the ways in which people use computers and in creating computers that were easy to use. They fought against the idea that computers had to be controlled by cryptic commands.

At that time, computers were controlled by command languages that were indecipherable to the naive user. Users could easily make mistakes if they did not remember what mode they were operating in. The same command or action could have radically different results depending on the application program or mode the user was in.

The PARC researchers argued that users should not be affected by modes, that commands and their results should be obvious, and that the

same action should always produce the same result. One researcher sported a T-shirt with the words "Don't mode me in." The group wanted to create a system that could be used by people in their work without their having to become computer scientists—a system that was controlled by manipulating familiar objects, not by working in an arcane command language.

The result was a computer called the Alto. It was never a product sold by Xerox but was a laboratory project that incorporated new ideas about how computers could be used.

The Alto was about the size of a two-drawer filing cabinet and had a high-resolution, bit-mapped graphics display. Characters were formed on the display as graphic patterns. This gave the machine a "soft" character set, since character shapes were determined by software. The software that ran on the machine created characters in many fonts, styles, and sizes.

The software concepts group at PARC developed Smalltalk, a combined programming language and operating system for the Alto. Smalltalk is an object-oriented language; that is, programs (procedures and data structures) are defined as objects that communicate by sending messages to each other.

Smalltalk's user interface took full advantage of the Alto's bit-mapped display. It used windows to allow the user to view more than one thing at a time. Smalltalk also used icons, or pictures, to represent such objects as disks, documents, and application programs that the operating system manipulated. The displays produced by Smalltalk running on the Alto were very similar to the displays you see on the Macintosh.

Eventually Xerox brought an office product to market that incorporated the user-interface concepts developed at PARC. It was called the Xerox Star. It was too expensive, however, to be bought for home or personal computing.

Steve Jobs of Apple Computer, Inc., saw the Alto perform during a visit to PARC and immediately saw the potential of a computer that anyone could operate. He set up two teams at Apple to produce computers that would use the concepts developed at PARC.

The first team designed and produced the Apple Lisa computer. It was initially more expensive than most personal computers with the same disk storage and memory, but its price later came down. The Lisa was the first machine produced by Apple to embody the concepts developed on the Alto.

The second team worked on a smaller, less expensive machine that could be mass-produced inexpensively but still have the type of user interface pioneered by Xerox's Alto and Star and Apple's Lisa. During its development, this second machine was code-named Macintosh, and later that became the actual name of the product.

The Lisa was an only modest success in the marketplace, but it showed Apple that people needed and were willing to pay for a machine that was easy to use. The Macintosh fulfilled the dreams of Apple executives with its brisk sales and the desires of computer buyers with its easy-to-understand user interface.

HOW THE MACINTOSH IS DIFFERENT

The Macintosh is a significant departure from previous personal computers. It has a well-thought-out user interface that represents system resources (programs, files, devices) as icons that the user can manipulate as if they were objects. Apple Computer, Inc., has specific guidelines for designing Macintosh application programs so that they are all compatible with this user interface and are based on the same concepts. The goal is that an application should offer no surprises to the user. Anyone who has used one Macintosh application program should find the operation of any other application to be almost intuitive.

No computer manufacturer has ever before attempted to provide user-interface routines in the operating system or application development guidelines that provide a consistent user interface from one application to another. The Macintosh is a gem for the user, but the capabilities that make it so easy to use also add complexity to the software.

HOW THE MACINTOSH IS THE SAME

The Macintosh is still a computer. It has an operating system, programming language, device drivers, a file system, interrupt handlers, and everything else that makes a computer go. And, like other computers, the Macintosh has an operating system with routines that perform system services for application programs, access device drivers, and offer other resources.

WHAT'S AT THE CORE OF THE MACINTOSH

In the Macintosh, you will find programs and data structures that support all of the normal operating-system features plus a series of similar resources that provide the standard user-interface capabilities. Some of the

the disk for us, keeps track of where our data is stored on the disk, and maintains a directory of disk contents so we can access files by name.

THE UNIQUE MACINTOSH OPERATING SYSTEM

The Macintosh operating system is an exciting departure from previous operating systems. It does all of the things you would expect an operating system to do, but its user interface is unique in the world of microcomputers. The Macintosh operating system turns the computer's world of bits and bytes into an environment that people understand, full of familiar objects instead of files, RAM, and ROM.

You tell the operating system what to do by manipulating objects on a *desk top*. Let's look at the Macintosh screen and see what sorts of objects we find (figure 2.1).

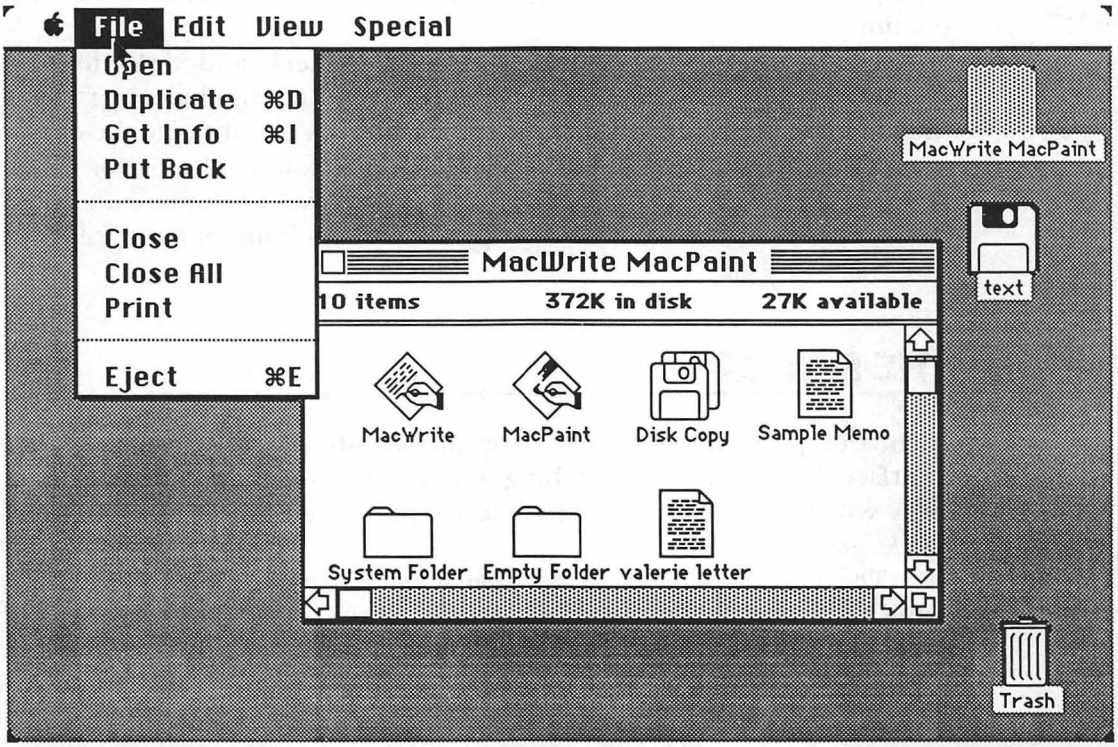


Figure 2.1 Macintosh desk top

On the screen, we see what appears to be a series of objects on a desk top. A particular type of picture, an icon, is used to represent an object or type of object.

Note that the cursor icon on our screen (upper left-hand corner) looks like a pointer. The cursor changes shape to indicate its capabilities.

You control the Macintosh by using the mouse and cursor icon to select, open, or drag objects on the screen. To select an object, you move the cursor to the object and press the mouse button once. This is called *clicking* the object.

Note the horizontal array of words at the top of the screen. These words represent menus that contain commands. On this screen, the user has selected the File menu by moving the cursor to it and holding down the mouse button.

Suppose you are the user and you want to look at a particular document that you have stored on a disk. You select the document and open it. The operating system converts this series of actions into commands for programs that run the disk controller, find the document in the disk directory, and run the proper program to allow the user to view the document.

The icons that represent documents, folders, and disks may be rearranged on the screen by being *dragged*. You drag an object by moving the cursor to it, holding down the mouse button, and moving the cursor again. The object or an outline of the object then follows the cursor. When you release the mouse button, the object is moved to the new location. Dragging is also used to move or copy documents from folder to folder or from disk to disk.

THE FINDER

From our point of view, what the Macintosh operating system's user interface does for us is to find things that we want to work on and put them away when we are done. The people at Apple Computer decided to call it *the Finder*.

The Finder keeps track of what resources and disks we have been using and where things are on those disks. It reads disk directories to find programs, folders, and documents.

The Finder uses icons to represent the objects that we can manipulate.

DISKS, FOLDERS, AND DOCUMENTS

The data that we can manipulate with the Macintosh is kept in documents. There are different types of documents for different classes of data. Text documents are created by a text editor (like MacWrite or Word) and contain text: letters, books, notes, and so on. Graphics documents are created by a graphics program (like MacPaint) and contain free-form graphics: pictures, pie charts, graphs, bar charts, and the like. Other types of application programs create documents unique to the application.

Note that each type of document is associated with a particular type of program. An individual document is always associated with the program that created it. If you choose to look at a text document created by MacWrite, the Finder automatically executes MacWrite when you ask to look at the document.

Folders are used to keep a group of documents together. If you are writing a book, you might want to keep each chapter in a separate document and all of the chapters together in a folder. Programming professionals will recognize folders as a type of subdirectory.

Documents, folders, and programs are stored on disks. To find the document you want, the Finder must know which disk to go to, which folder the document is in, and what program was used to create the document. The Finder gets all of this information from a directory on the disk itself.

When you first turn on your Macintosh, the Finder doesn't know anything about your disks. If you want to view a particular document, you must know which disk it is on and put that disk in the disk drive. The Finder then reads the directory of the disk and stores it in memory. If you later put a different disk in the drive and ask for a document stored on the first disk, the Finder will ask you to put the first disk back in the drive.

MENUS

To select a menu, you move the mouse pointer to the menu title and hold down the mouse button. When the menu is selected, its title is displayed in reverse video (white on black), and the entire menu is displayed under the title, overlaying any other information that might be on the screen. As long as the mouse button is down, the menu is displayed. We call this action *pulling down the menu*.

To select a menu item, you move the mouse pointer down the menu until it rests on the desired menu item and then release the mouse button.

As the pointer moves over each item, the item is displayed in reverse video. When you release the mouse button, the menu disappears, its title reverts to normal video, and the area it overlaid on the screen is restored. If the item you selected performs an action that takes a while, the menu title will remain in reverse video until that action is completed.

SELECTING AND OPENING

To use the contents of a disk, folder, or document, you must select it. To select it, you move the mouse pointer to the icon representing the object you want and then click the icon. Clicking identifies the object to the Finder so it can prepare to manipulate it for you.

In order to see what's inside an object (or to change it), you must open it. There are two ways to open an object. If you have already selected it, pull down the File menu and choose the Open option. If you have not selected the object, move the pointer to it and double-click. When the object is opened, the Finder will put a window on the screen displaying its contents.

Suppose you want to look at a memo that you created with MacWrite. It's in the Memos folder on a disk labeled Office Stuff. You put the Office Stuff disk in the disk drive and select the icon for that disk (figure 2.2).

The Finder opens the disk and displays a window showing the contents of the disk (figure 2.3). Note that the name of the disk is in the window's title space. You see the Memos folder, move the pointer to it, and double-click.

The Finder opens the folder and displays a window with the contents of the folder (figure 2.4). The word in the window's title space—Memos—is the name of the folder. You now see your Deadlines memo and click it with the mouse.

The Finder opens your document and displays your memo in an Edit window (figure 2.5, page 16). The title area of this window has the name of the document.

When you open a document, the Finder not only opens the document, it runs the program that created the document. If you want to create a new document, you just run one of the programs that deal with documents. To run a program, you select and open the icon that represents the program.

If you want to create a graphics document, you first open the Write/Paint disk. You then move the pointer to the MacPaint icon and double-click it. The Finder creates a document, displays a window, and starts MacPaint (figure 2.6, page 17). Note that the document it has created

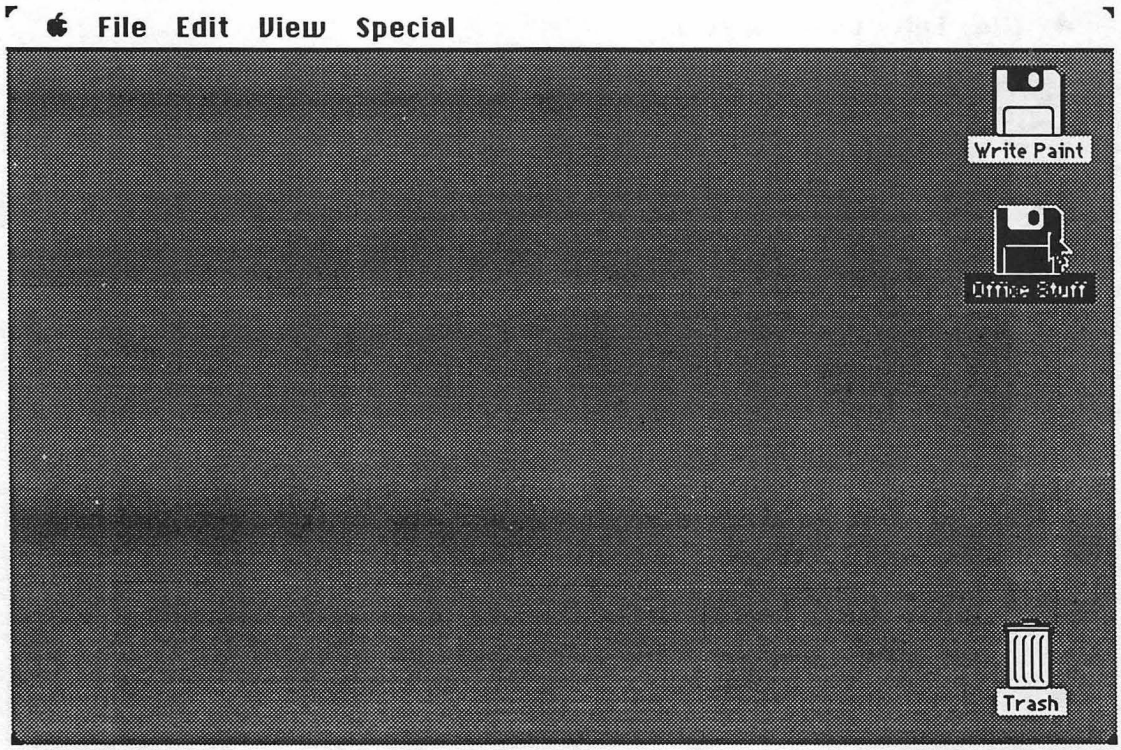


Figure 2.2 Office Stuff disk

is as yet untitled. When you are finished working on the document and you save it, the Finder will ask you to name it.

DIALOGS AND ALERTS

When the Finder has an urgent message for you, it will display an alert box with the message. The box may overlay your active window but will disappear when you have acted on the alert message. For example, if you have tried to open a document and the program that uses that document is not on a disk in one of the disk drives, you will see an alert box like the one in figure 2.7 (page 18).

This alert box tells you that you can't do what you are trying to do because the program for that function isn't in the computer. The rounded rectangle with OK in it is called a button. You simulate pushing the button by clicking it with the mouse. When you have read the alert, click the OK button, and the alert box will disappear.

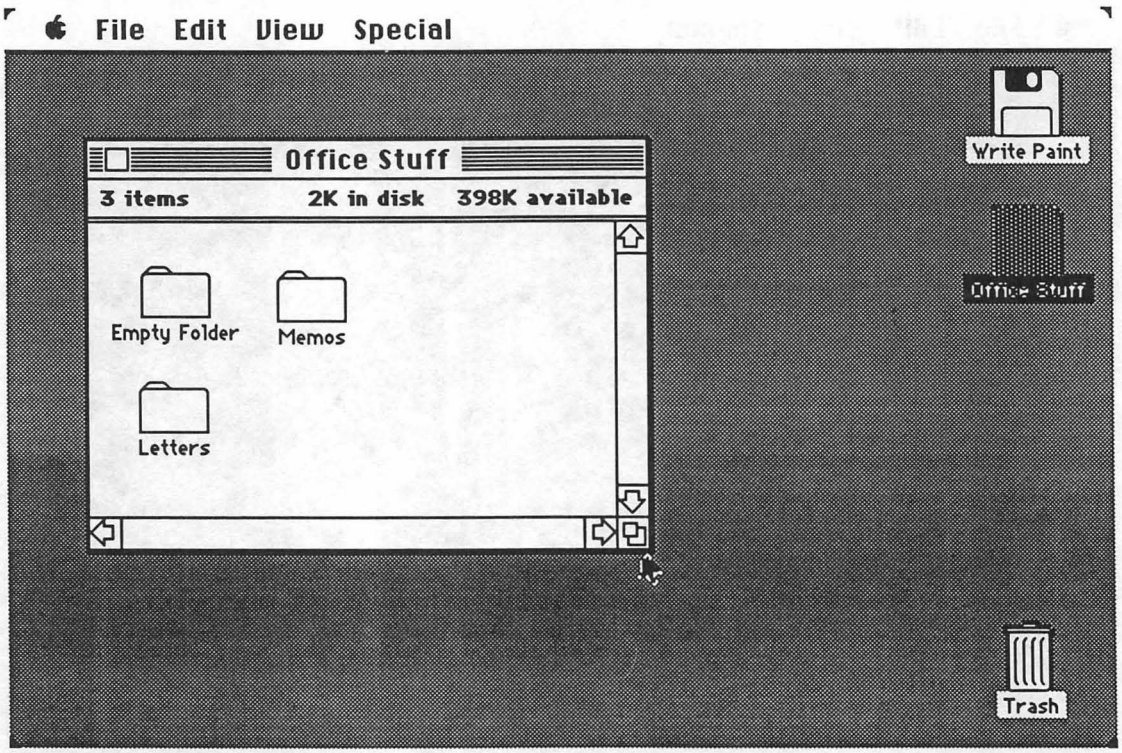


Figure 2.3 Open Office Stuff disk

When the Finder or an application program needs information from you, it will open a dialog box and display a question. If you click the Save option in the File menu, you will see a dialog box like the one in figure 2.8 (page 19). The Save dialog box gives you the option of canceling the save request or supplying a name for the document and saving it. As the term *dialog* implies, there may be more than one question in the box that requires an answer. You may actually carry on a dialog with the program that generated the dialog box.

EDITING AND THE CLIPBOARD

The Macintosh user interface includes routines for editing text, wherever it appears on the screen. The text may be part of a document or even the name of an object, such as a document, folder, or disk. Whenever a programmer wants a user to enter data, the programmer can use the edit routines to conform to the Macintosh standard user interface.

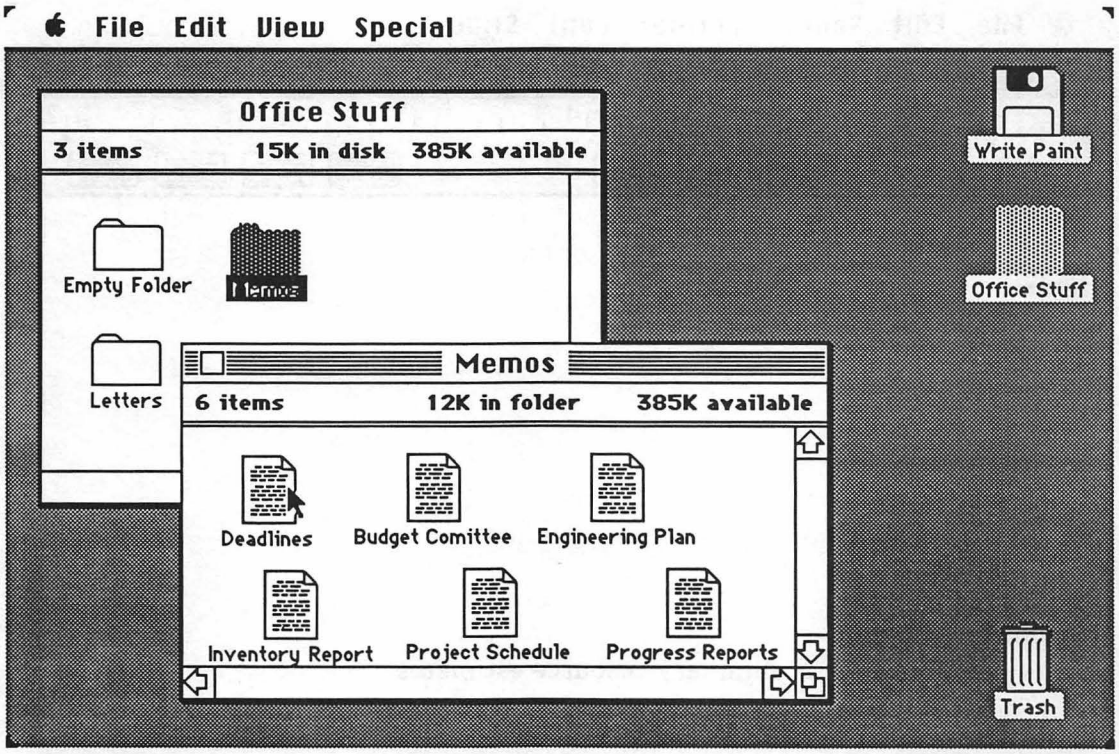


Figure 2.4 Open Memos folder

The edit routines allow you to delete text from the area you are editing by using the Cut option from the Edit menu. Let's take a look at the Edit menu and see what it can do (figure 2.9, page 19).

Note the Show Clipboard option at the bottom of the menu. The Clipboard is fundamental to the operation of the edit routines. It is an area of memory set aside for storing any text that you delete or copy from a document.

You can delete text by selecting the text (dragging over it with the mouse) and choosing Cut from the Edit menu. The text is then deleted from the area you are editing and copied onto the Clipboard.

You can use the Paste option to move the text on the Clipboard to another location in the document. First move the cursor to the place where you want to insert the text, and click the mouse. Then select the Paste option from the Edit menu, and the text is moved.

The Copy option from the Edit menu is similar in operation to the Cut option, but it puts the selected text on the Clipboard without deleting it from the document.

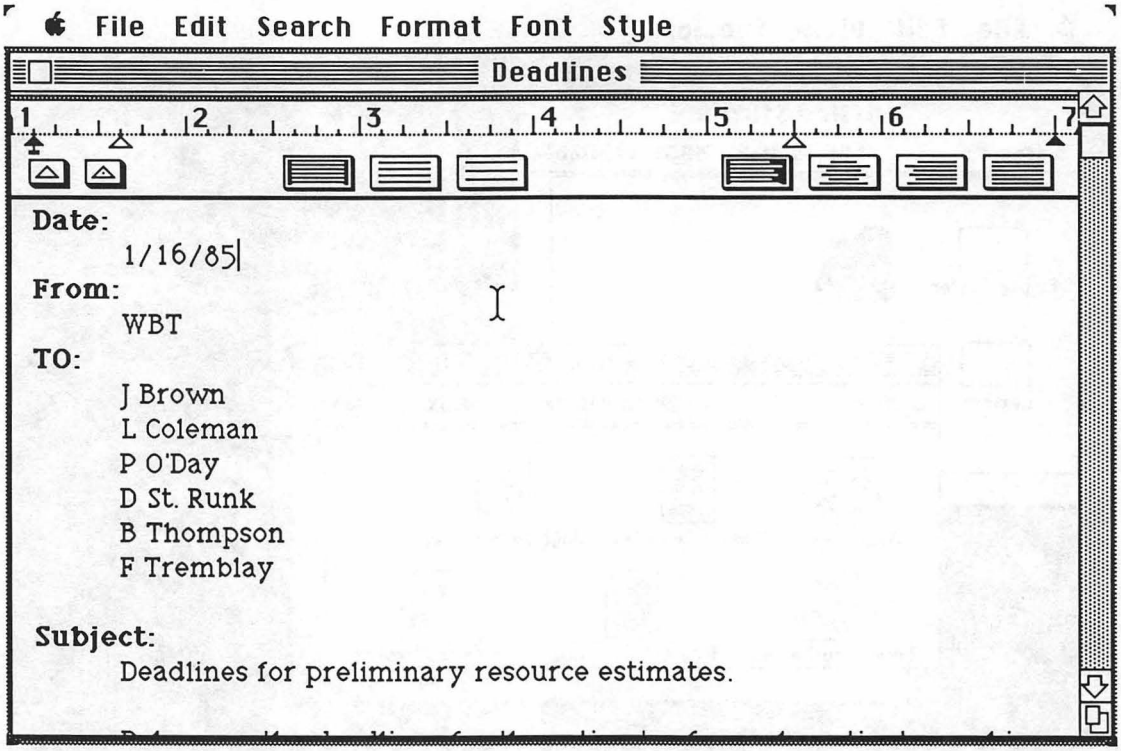


Figure 2.5 Open Deadlines document

You can use the Clipboard to move text from one document to another. If you leave the document you are working on (close the document) and go to another, the text you last put on the Clipboard is still there. You can then paste it into the new document.

THE SCRAPBOOK

The Scrapbook is similar to the Clipboard except that it retains data even when the disk is ejected and the Macintosh is turned off. The Scrapbook can store images as well as text.

You put text or pictures into the Scrapbook by copying them onto the Clipboard, opening the Scrapbook, and pasting them into the Scrapbook. You retrieve things from the Scrapbook by doing just the reverse: open the Scrapbook, select the text or picture, copy it onto the Clipboard, open a document, and paste the item into the document.

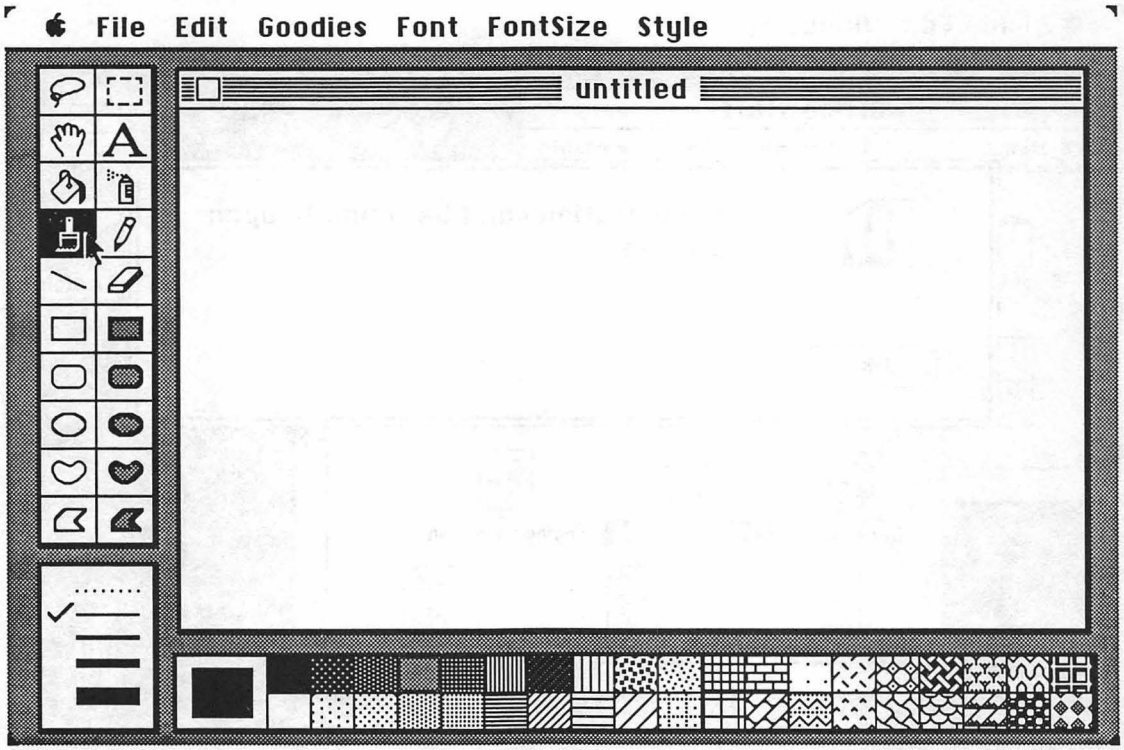


Figure 2.6 MacPaint window

When you store text in the Scrapbook, the text does not retain the formatting, font, style, or size information. Text copied onto the Clipboard does retain that information.

The Scrapbook is a convenient place to keep pictures and text that you use frequently.

THE FILE AND VIEW MENUS

At the top of the Macintosh screen you always see the words File, Edit, and View. Clicking any of these will get you a menu. Let's look at these menus and see how they are used. First we will look at a list of the menu options. Then we will try an exercise in using the File and View menus.

The File menu lets you open, close, print, and do other things to files. The File menu selections include:

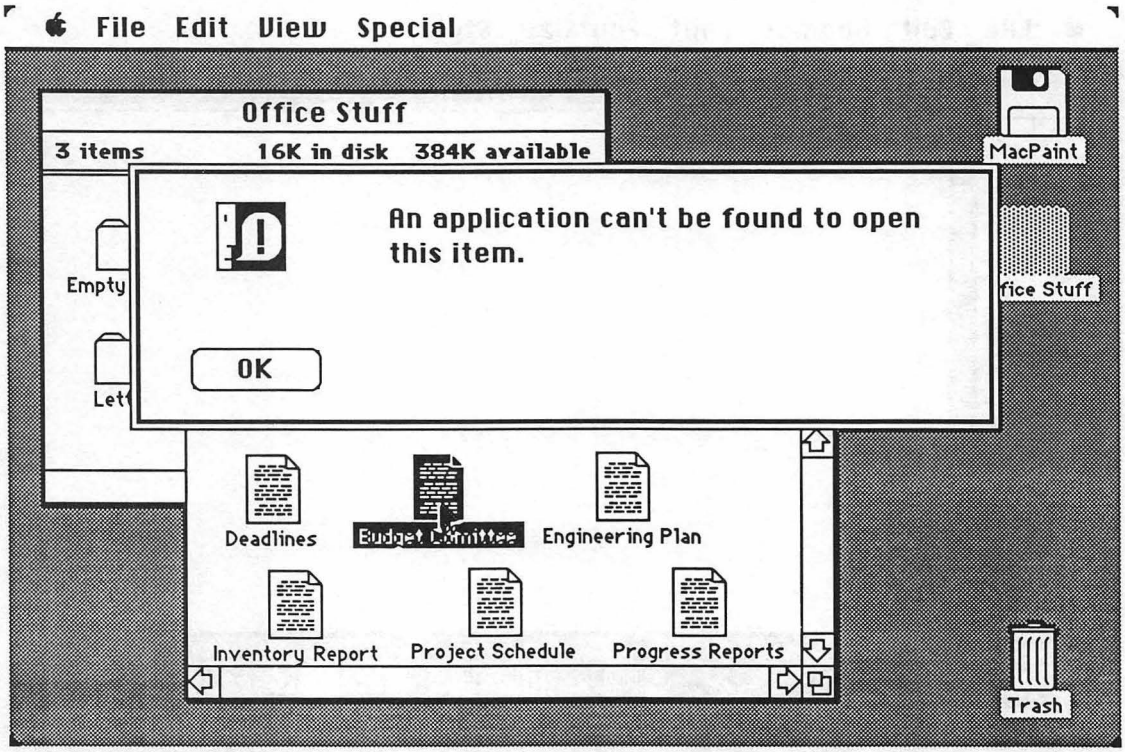


Figure 2.7 Alert box for nonexistent program

Open	Open selected icon
Duplicate	Duplicate selected icon
Get Info	Show information about selected icon
Put Back	Put selected document or folder back in folder or disk that it came from
Close	Close active window
Close All	Close all windows
Print	Print selected document
Eject	Eject selected disk

The View menu displays the directory for a disk, a folder, or the trash. The View menu options allow you to view the contents of the directory in a number of ways:

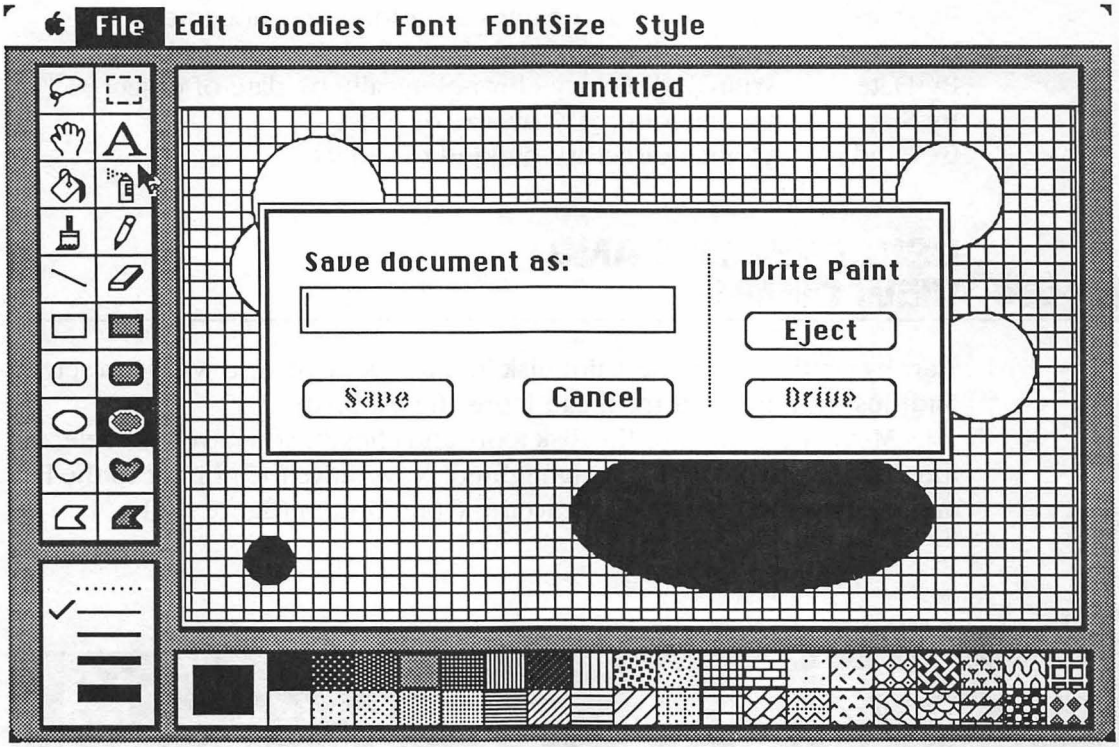


Figure 2.8 Save dialog box

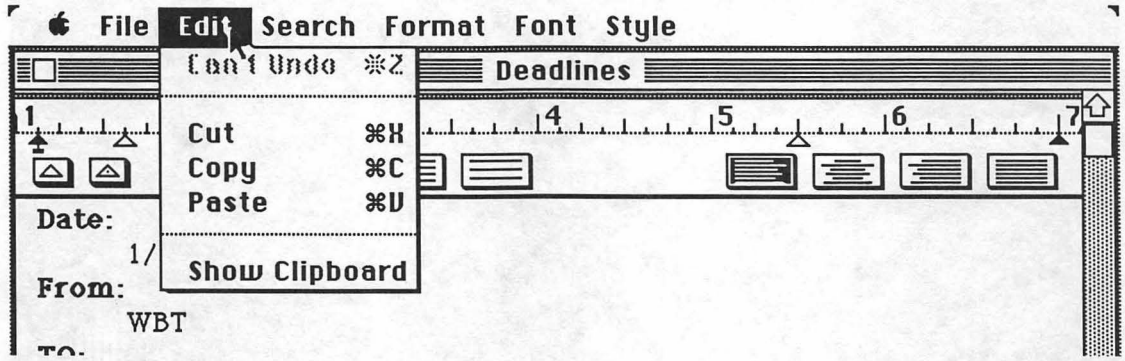


Figure 2.9 Edit Menu

By Icon	Show icons and names for folders and documents
By Name	Arrange directory alphabetically by name of object
By Date	Arrange directory chronologically by date of object
By Size	Arrange directory by size of object
By Kind	Arrange directory by kind of object

USING THE FILE AND VIEW MENUS

Start by putting the Write/Paint disk in the disk drive. You will see a screen with just the disk and trash can icons (figure 2.10).

Move the pointer to the disk icon, and click it. You have now selected the disk icon. Note that it is highlighted. Now move the pointer to the File menu. Drag the pointer down to the Open option (figure 2.11).

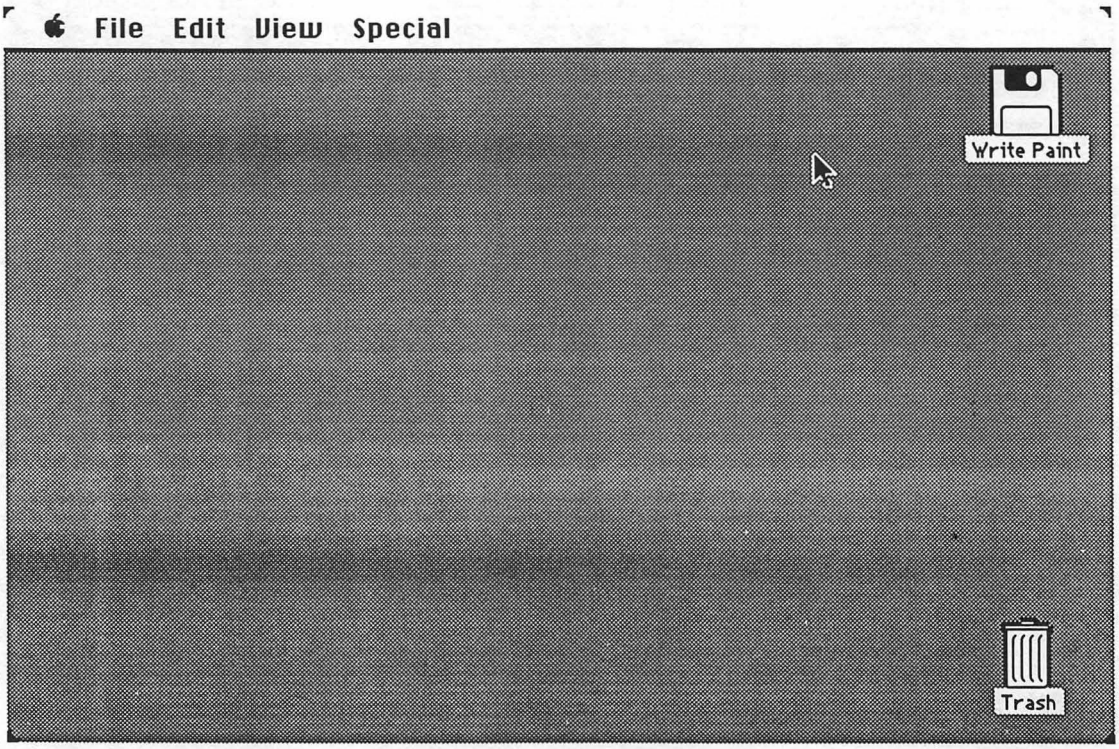


Figure 2.10 Write/Paint disk icon

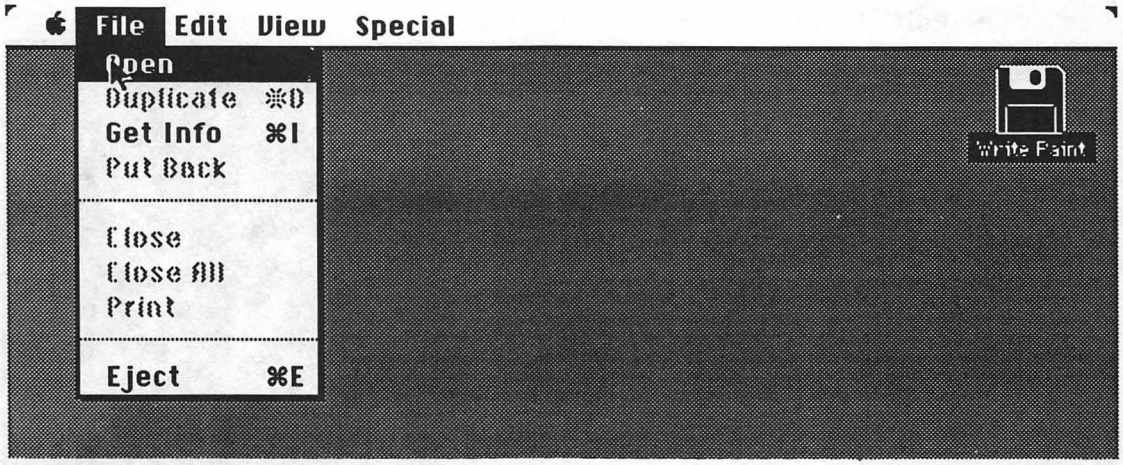


Figure 2.11 Open option selected in File menu

Release the mouse button, and the Finder will open the disk and display the disk window with the names of the programs, folders, and documents on the disk. Your disk window may be different from the one shown (figure 2.12) because you may have different documents on your disk.

Now move the pointer to the icon for the MacWrite program (figure 2.13). When you click the icon, the Finder opens a window for MacWrite. Note that the cursor has become an I-beam instead of a pointer. There is also a flashing vertical bar to indicate the insertion point in the text. Now type your name or anything else you feel moved to type; we just need some text for our document (figure 2.14).

Move the pointer to the File menu, hold down the mouse button, and drag down to the Quit option (figure 2.15).

When you release the mouse button, a dialog box appears, asking if you want to save the changes you made to the document (figure 2.16).

Use the mouse to click the Yes button in the dialog box. A Save dialog box now appears to request a name for the document (figure 2.17, page 25). Type a file name in the dialog box; use *tinydoc* as a name, or invent your own (figure 2.18, page 25).

Move the pointer to the Save button in the dialog box, and click it. The dialog box disappears, and the MacWrite window shrinks into the MacWrite icon. Now move the pointer to the View menu, and drag down to the By Date option (figure 2.19, page 26).

When you release the mouse button, the Finder replaces the disk-window contents with a list of files arranged chronologically by the date

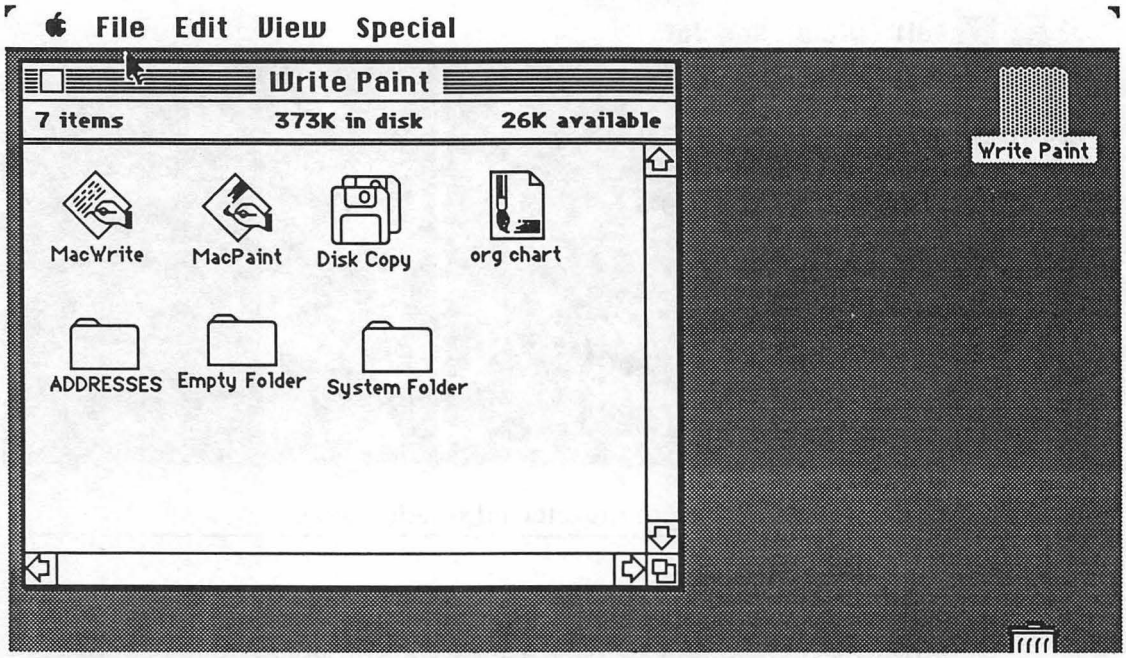


Figure 2.12 Write/Paint disk directory window

each was last modified (figure 2.20, page 26). The file you just created should appear at the top of the list. You can try the other View-menu options on your own.

Now pull down the File menu, drag down to the Close option, and release the mouse button. The disk window shrinks back into the disk icon. If you are through using your Macintosh for now, use the Eject option in the File menu to eject your disk.

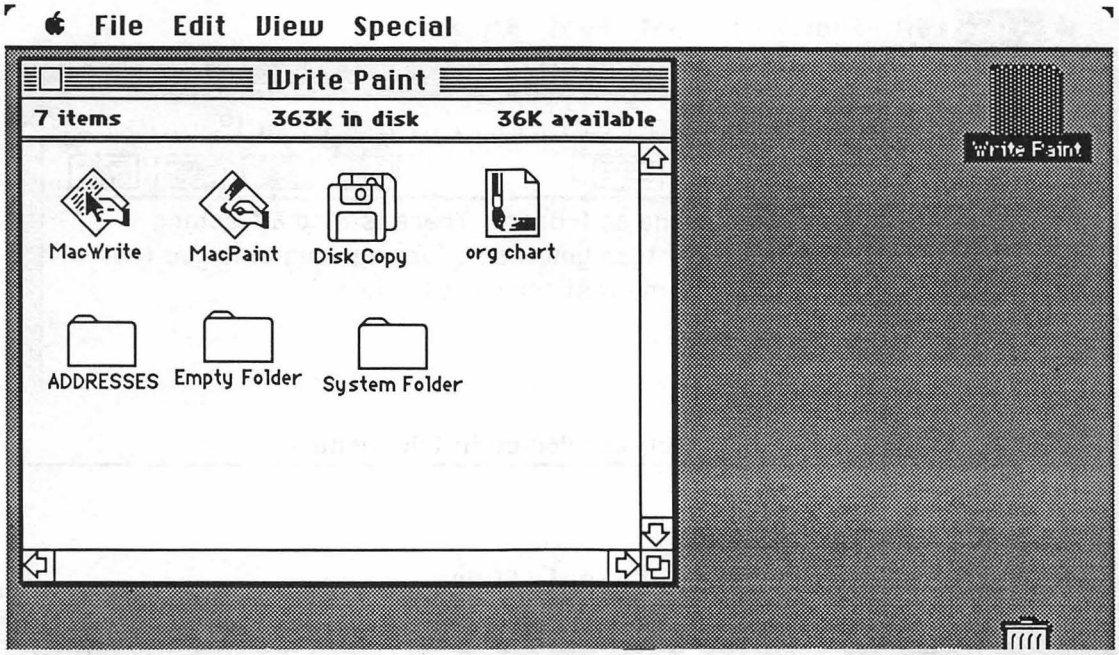


Figure 2.13 Pointer on MacWrite icon

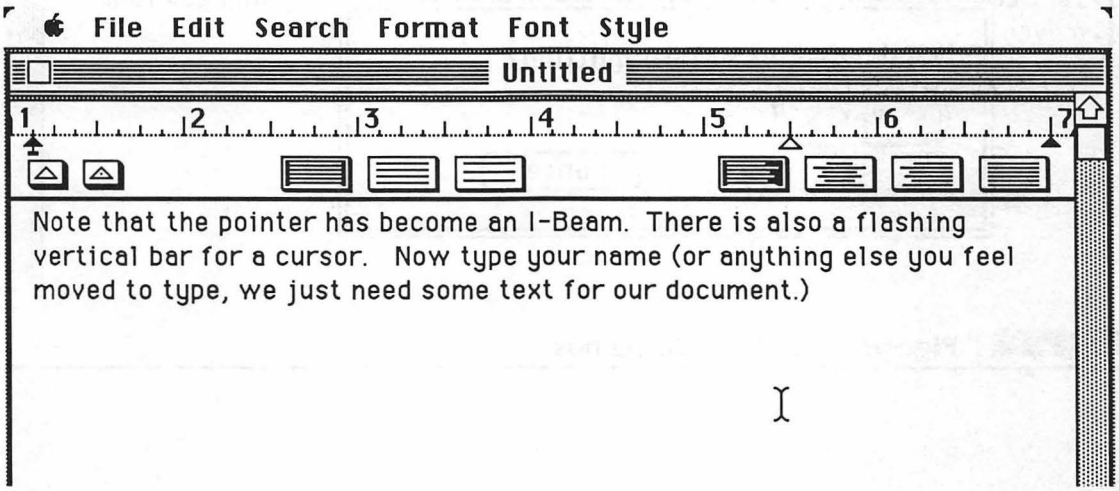


Figure 2.14 Text in MacWrite window

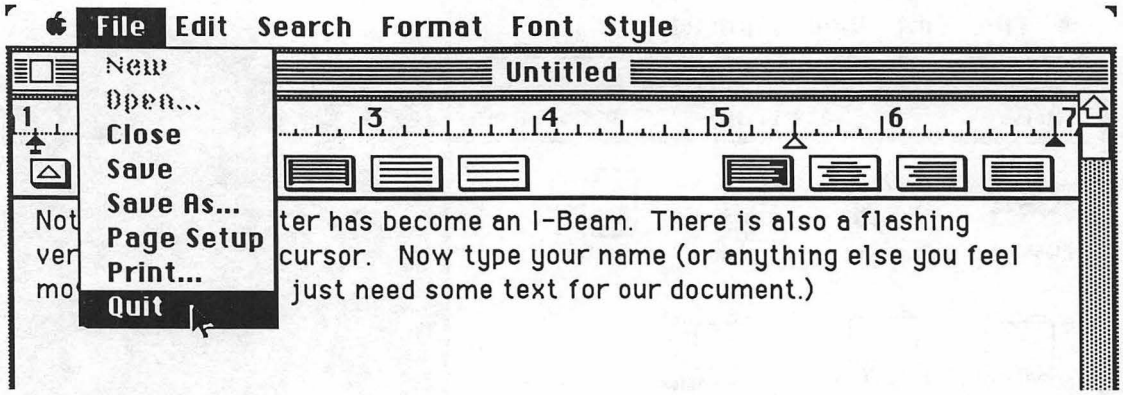


Figure 2.15 Quit option selected in File menu

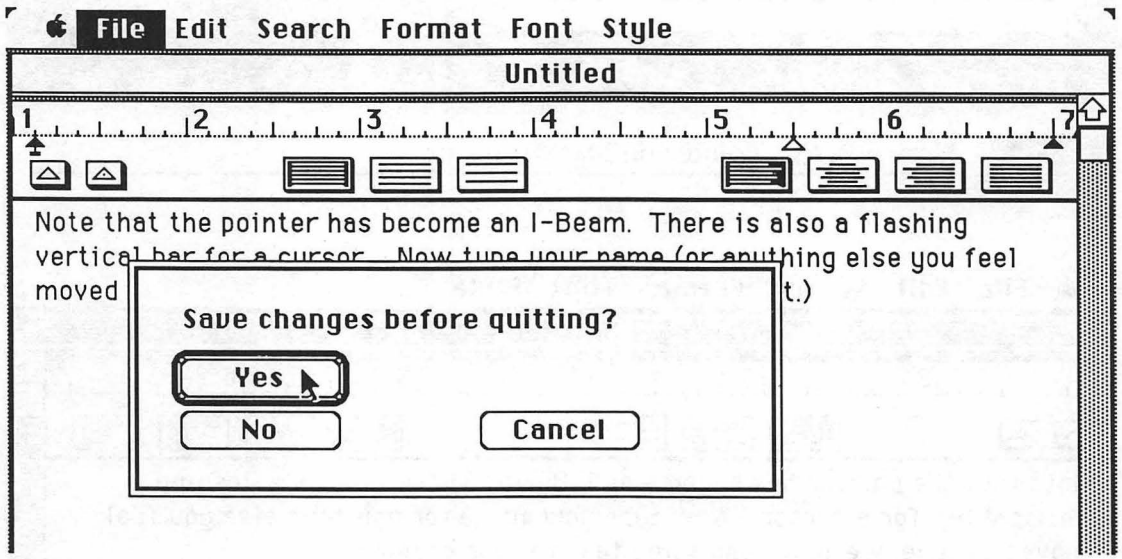


Figure 2.16 Quit dialog box

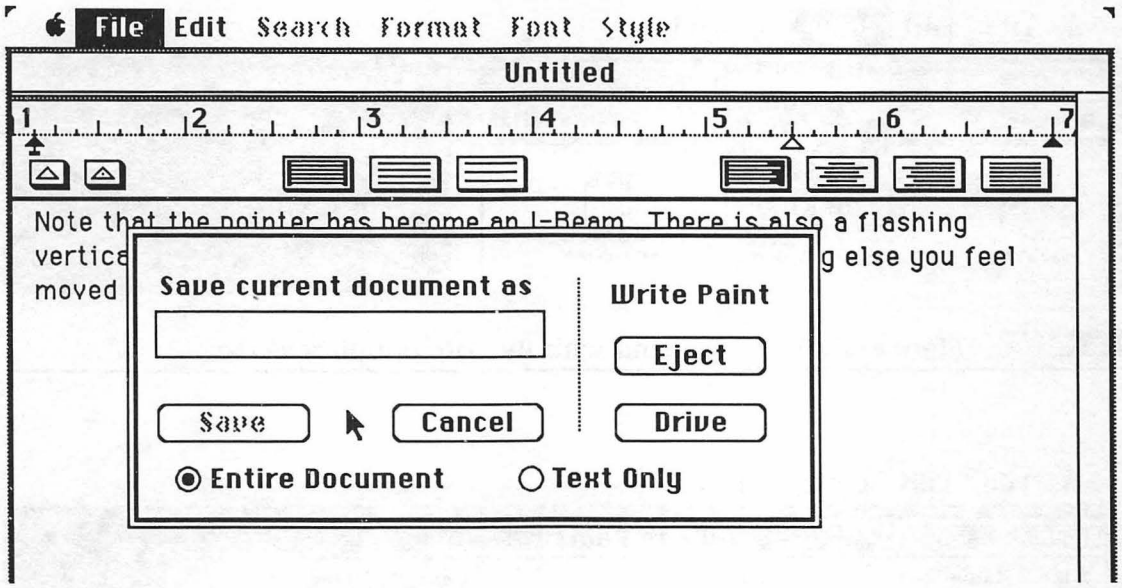


Figure 2.17 Save dialog box

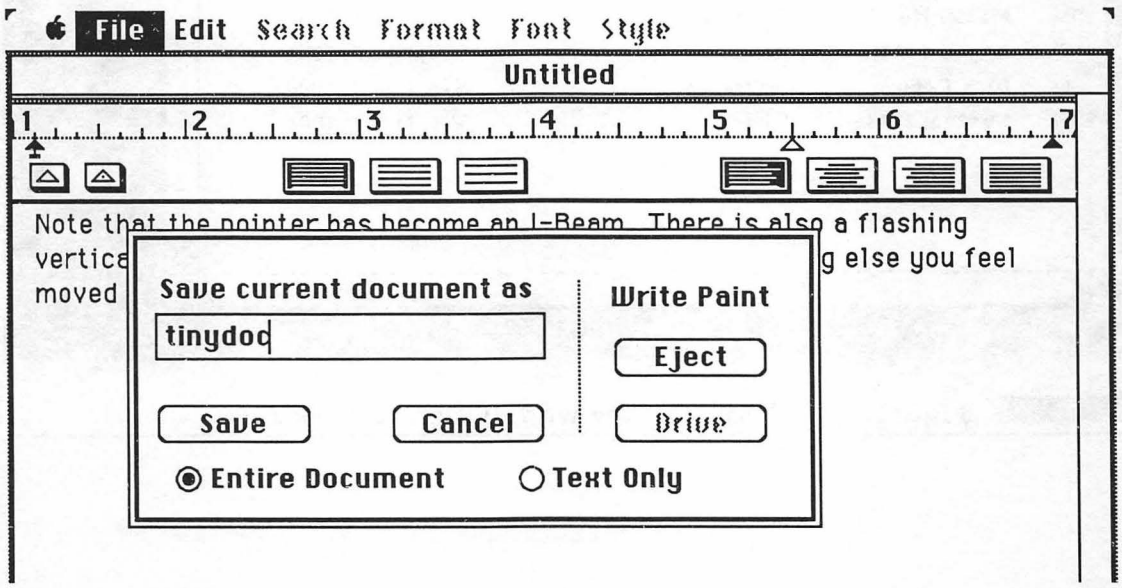


Figure 2.18 Save dialog box with file name

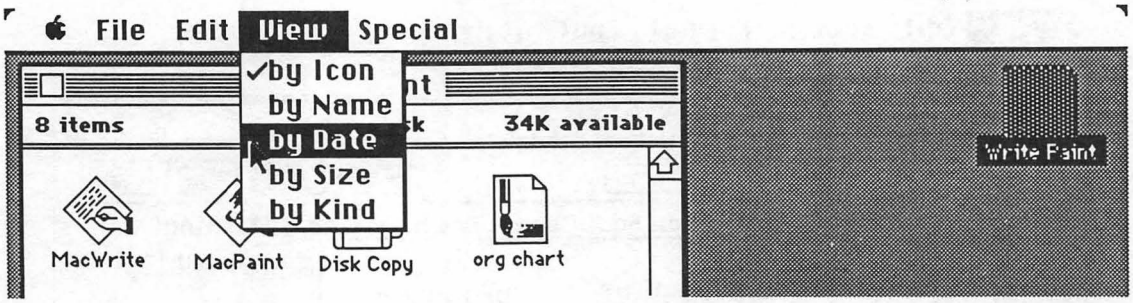


Figure 2.19 View menu with By Date option selected

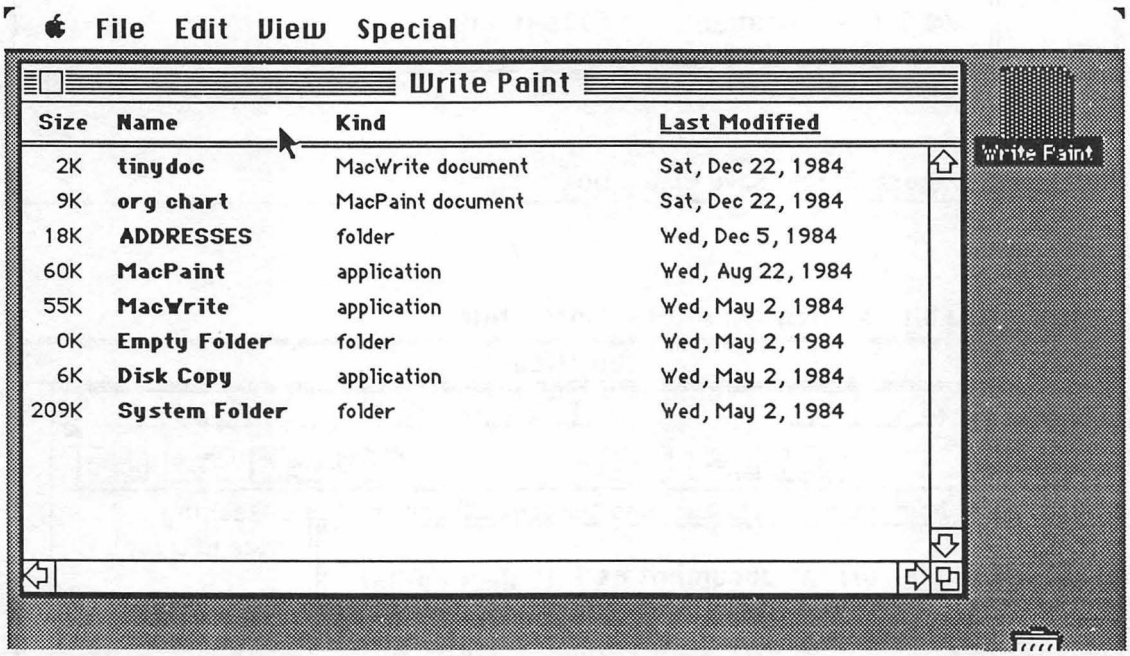


Figure 2.20 Disk window with By Date directory

CHAPTER

3 DESIGNING SOFTWARE FOR THE MACINTOSH

The Macintosh Environment
The User Interface
Designing Menus
Alert and Dialog Boxes
Designing with Windows
Desk Accessories
The Display and Graphics
Character Fonts and Styles
Disks, Folders, Files, and Documents
The Conceptual Model Reviewed

THE MACINTOSH ENVIRONMENT

It's no longer enough to be the world's greatest programmer. With the Macintosh, designing software is as much an art as a science. Like an architect, a software designer must be technically competent and creative. The designer must create something that is functional, yet easy to use, easy to learn, and aesthetically pleasing.

A central issue in designing software for the Macintosh is consistency. A user who selects a section of text to be deleted should be able to use the same action to select a range of cells in a spreadsheet. There should be no surprises. Even, and especially, within a single application, the same action should always produce the same result.

The user should be able to run the program by manipulating familiar objects. In the Macintosh, we represent these objects with a special 32-by-32-bit picture called an icon.

Programs shouldn't confuse users. When users are confused, they get frustrated; they don't trust themselves to operate the application and probably don't trust the results. So they're not likely to recommend your program to a friend or buy your next one.

One of the best ways to confuse users is to change modes without telling them. This causes programs to produce apparently unpredictable results. Before the Macintosh, many programs were designed to operate in several modes. A text editor is a good example. Many editors have two modes, command mode and input mode. In command mode, the editor interprets everything you type as commands to the editor. In input mode, it interprets everything you type as text to be put in the file you are editing. If you think you are in input mode but are really in command mode, you are typing text for your file, but the editor is trying to interpret that text as commands. The results can be surprising and sometimes disastrous.

Modes have a place in software design, but they need to be used sparingly. The purpose of a mode is to tell the application program how to interpret the user's input. To be useful, a mode must be obvious to the user. You can make a mode obvious by using the shape of the mouse pointer, the type of display or window, or another graphic device to indicate the mode the user is in.

Another technique is to make the mode dependent on the user's holding down a key or the mouse button. In that way, the user becomes part of the mode and cannot enter or leave it accidentally.

A user can become hopelessly lost in a complicated system with too many modes, more than one way to produce the same results, or too many features. A good design principle is to keep the user interface simple:

eliminate all unnecessary displays, modes, and commands, and then eliminate some more.

Keep your displays simple, too. A cluttered screen is aesthetically ugly and hard to understand.

Any program results that are displayed on the screen should be identical to the same results printed on paper. The rule should be: What you see is what you get.

Let's summarize our design goals:

1 Avoid modes. If you use them, make them obvious.
2 There should be no surprises.
3 The same action should produce the same result.
4 What you see is what you get.
5 Keep it simple.

THE USER INTERFACE

Apple has provided a number of devices that can be used by our programs to create a user interface. They are all pictures that appear on the screen and can be manipulated by the user.

The conceptual basis for the user interface is a desk top, and the screen of the Macintosh is an electronic representation of it. The user keeps familiar objects on the desk top and may move them around, pile them on top of each other, or dispose of them.

There are other icons and devices that we use to fill out our conceptual model: alert and dialog boxes, windows, disks, folders, documents, desk accessories, and menus.

DESIGNING MENUS

Menus are the primary means of controlling programs. The Macintosh menus are represented in a menu bar at the top of the screen. Menus may vary from one program to another, but the three standard menus—Desk Accessory, File, and Edit—should always be included and should always appear in the same place. The Desk Accessory menu is represented by an apple symbol on the far left side of the menu bar (figure 3.1).

In designing the menus for your applications, consistency is of prime importance. The menus should always appear on the menu bar in the same order. Each should always have the same entries. As conditions change, you will not want all of the menu entries to be effective all of the time, but

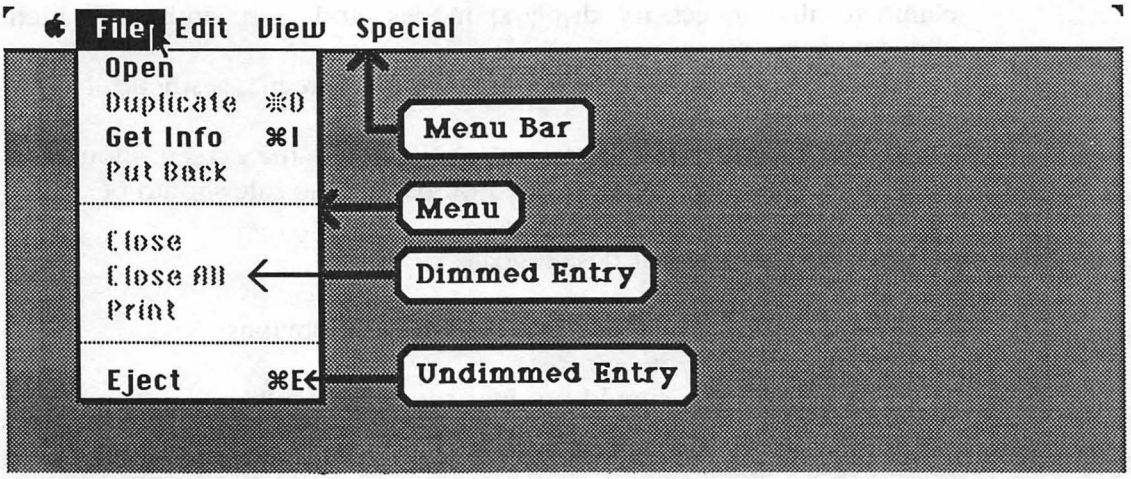


Figure 3.1 Menu

you needn't remove an entry to disable it. Instead, you can make it display with dimmed characters. Then only the menu items displayed with undimmed characters are active.

A menu may be as wide as you want, as long as it fits on the screen. Since only one menu may be pulled down at one time, there is no danger of one menu's overlaying another.

You can use menus such as the File menu to select actions for the program to take. Or you can use menus to select options. The Font and Style menus in MacWrite are menus that allow the user to select options (figure 3.2). The user pulls down the menu, selects a menu entry, and releases the menu. The next time the menu is pulled down, the currently selected entry is identified by a check mark.

Some menus, like the Edit menu, are usually coupled to a set of toolbox routines that perform their functions. By always using the same Edit menu that the system uses and calling the toolbox routines to do the editing, you avoid having to write edit routines and ensure that your program is consistent with every other Macintosh application that does editing.

The user controls the program with menus, but the program controls the user with alert boxes.

ALERT AND DIALOG BOXES

An alert box delivers the message that something unexpected is happening or that the user needs to take some action. The alert box overlays whatever

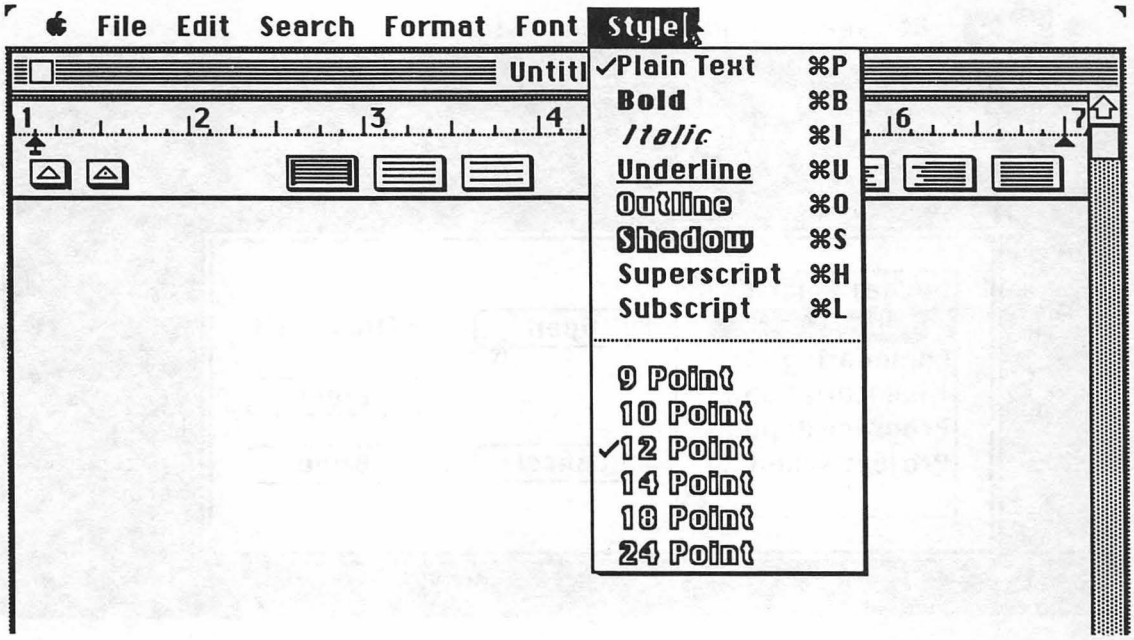


Figure 3.2 Style menu

is on the screen; the overlaid screen is restored when the alert box disappears. Most alert boxes do not disappear until the user recognizes them by clicking a button in the box.

Dialog boxes are similar to alert boxes, but they appear when longer exchanges of information between a program and a user are required. They should be used in situations that cannot be handled by the user's normal method of input to the program. A dialog box may be just a small area where the program writes messages to and reads messages from the user, or it may contain a variety of messages, buttons, dials, or icons. The dialog box shown in figure 3.3 appears when you select the Open option from the File menu in the MacWrite program. This particular dialog box is often called the Minifinder because it does one of the Finder's functions: it displays a disk directory with documents of a particular type and lets you choose one.

Note that the user may choose from several courses of action by clicking one of four buttons. The Cancel button allows the user to back out of the situation by canceling whatever action invoked the dialog box. Always allow a way out with a Cancel button unless there is a catastrophic situation that the user cannot recover from, such as the disk's being full.

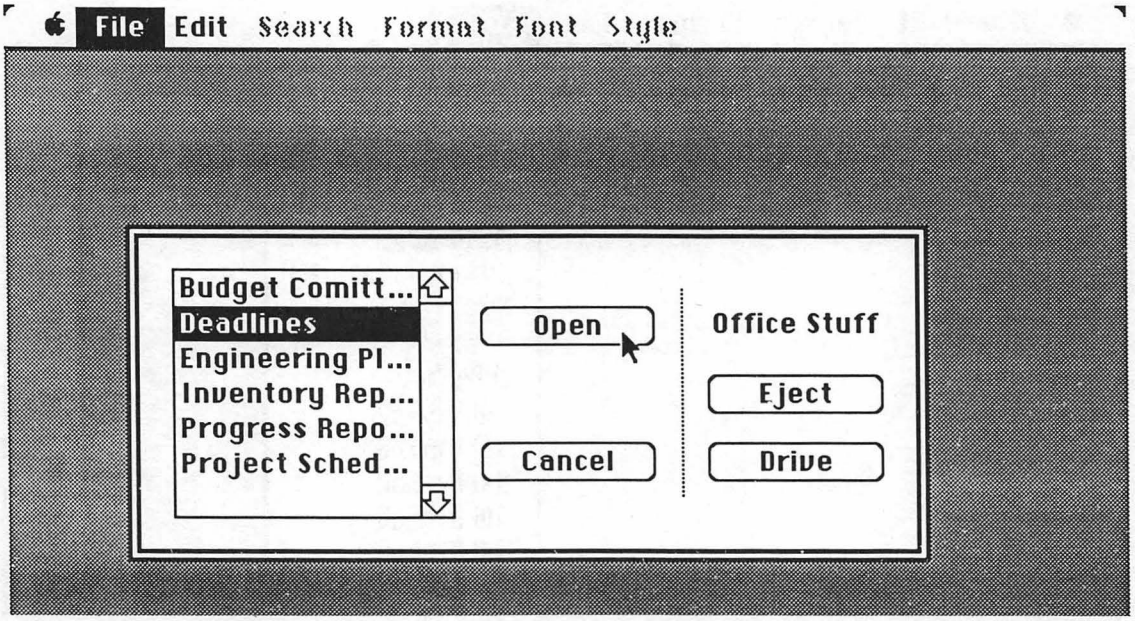


Figure 3.3 Open dialog box

A dialog or alert box should be used to resolve a temporary situation that requires a decision or information from the user. A dialog with the user that is of an ongoing nature would be better handled by a window.

DESIGNING WITH WINDOWS

A program, such as a word processor, that needs a display area for output or for the user's work space should put the display in a window. The usefulness of windows lies in their consistency. All windows behave the same way. Thus, once users learn how to control one window, they can control any window.

A window is a display area for viewing a portion of a document and is provided by the user-interface toolbox. As a user, you may change the size of the window to see more or less of the document, or you may change the portion of the document you are viewing by scrolling the window. You may move the window to another portion of the screen.

The window contains a title bar that lists the title of the document or the word *untitled* if the document has no title (figure 3.4). To the left of the title bar is a box called the *close box*. Clicking the close box closes the window and removes it from the screen.

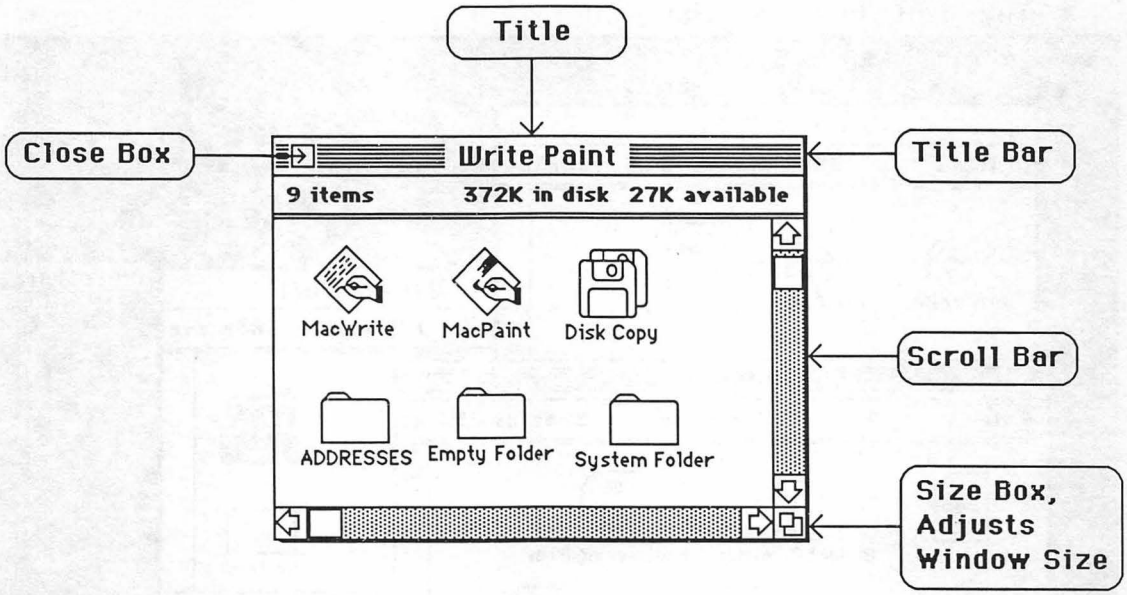


Figure 3.4 Window

When there is more than one window on the screen, only one window is active. The others are displayed, but you cannot do anything with them; you cannot scroll them or make any changes to the document each displays. To make a window active, you simply click the mouse anywhere inside it.

Windows can overlay one another on the desk top (figure 3.5). There is always one window on top; it is the only active window on the screen. The overlaid windows look like a pile of documents on your desk. When you leave one window and make another active, the new active window moves to the top of the pile and overlays the others. Even the topmost window can be overlaid by a desk accessory.

DESK ACCESSORIES

A desk accessory is a program that can be accessed from the Desk Accessory menu (the menu represented by the apple symbol in the menu bar). It is the only type of program that you may run without closing an active application. When you select a desk accessory from the menu, the system temporarily suspends the current application and overlays the display on the screen with the desk accessory's display. When you close

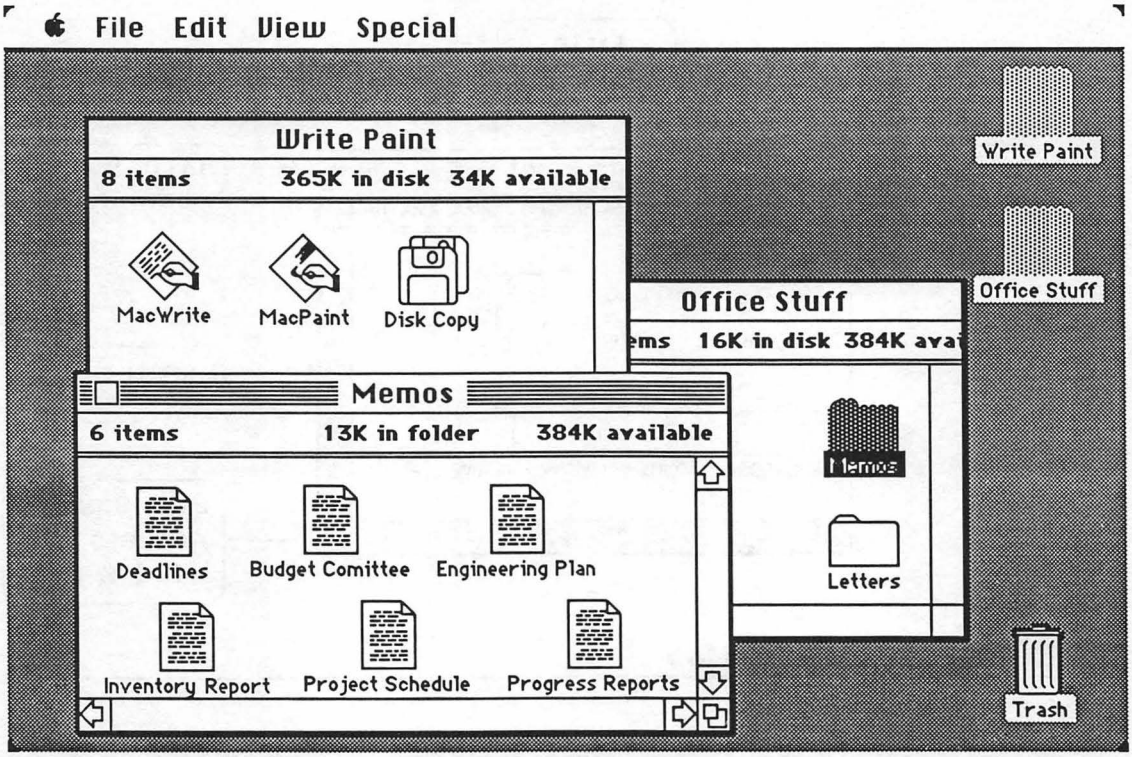


Figure 3.5 Overlaid windows

the desk accessory (click its close box), its display disappears, the previous display reappears on top again, and the application you were running is restarted at the point where the system suspended it. The application never knows that it was suspended.

The most likely candidates to become desk accessories are small, simple programs. A desk accessory usually does not create a normal window with a size box and scroll bars, nor does it use menus. Let's look at a sample desk accessory. We'll start with an active MacWrite window; our old Deadlines memo is being edited (figure 3.6).

We will use the Note Pad desk accessory and add an item to our "To Do" list. First we move the pointer to the apple symbol in the menu bar. Then we move the pointer down to the Note Pad item (figure 3.7).

We select the Note Pad by releasing the mouse button with the pointer on Note Pad. The Note Pad overlays a portion of the MacWrite window (figure 3.8). The system reads the current Note Pad file contents and displays them on the Note Pad.

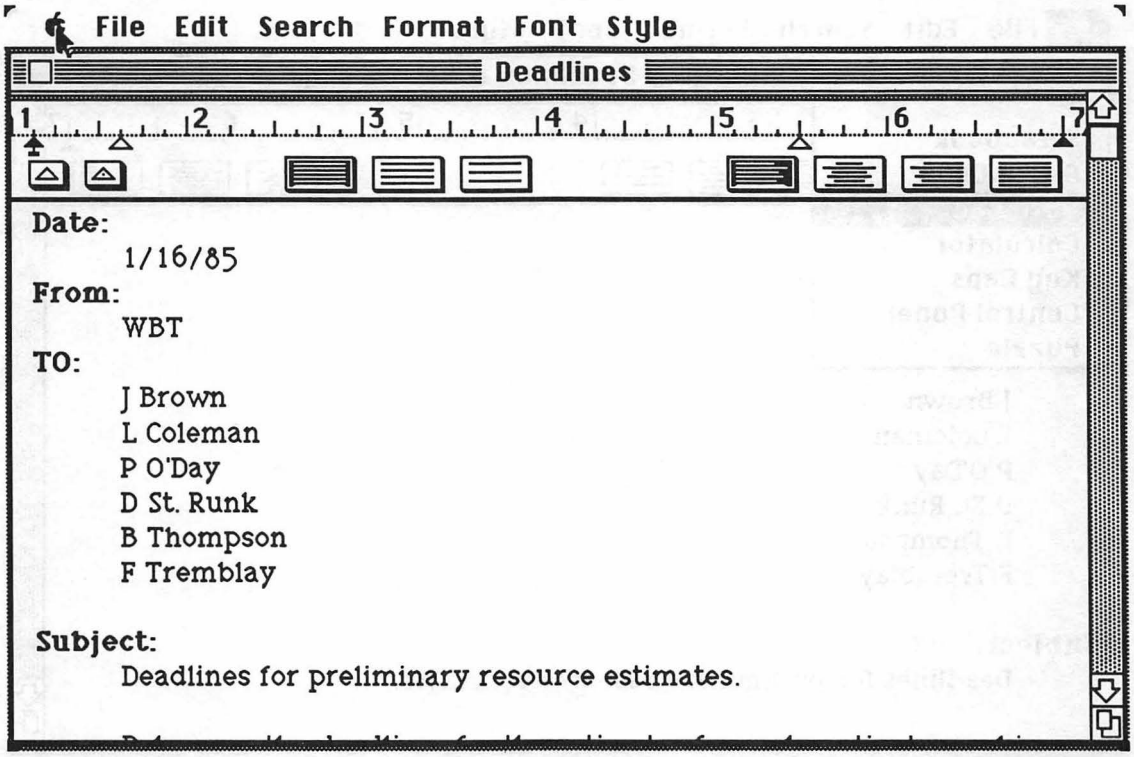


Figure 3.6 Display before Desk Accessory menu is invoked

We add a fourth item to the Note Pad and move the pointer to its close box (figure 3.9). We click the close box, the Note Pad disappears, and we are back in the MacWrite window. The Note Pad program stored the new Note Pad contents in a file so that they will be available the next time we invoke the Note Pad desk accessory.

This example demonstrated some of the design principles discussed earlier. We not only changed modes but also went to a completely different program, and the change was still obvious to the user. The new program's display was quite different from the old one, and by overlaying the old display, the desk accessory program made it evident to the user that something new was happening. The user could also see that while the Note Pad was active, no actions could affect the document that was being edited with MacWrite.

Try to imagine making that kind of transition on a computer that cannot display text and graphics together. How would it be possible to make the change without losing the user? It's easy to see that we depend

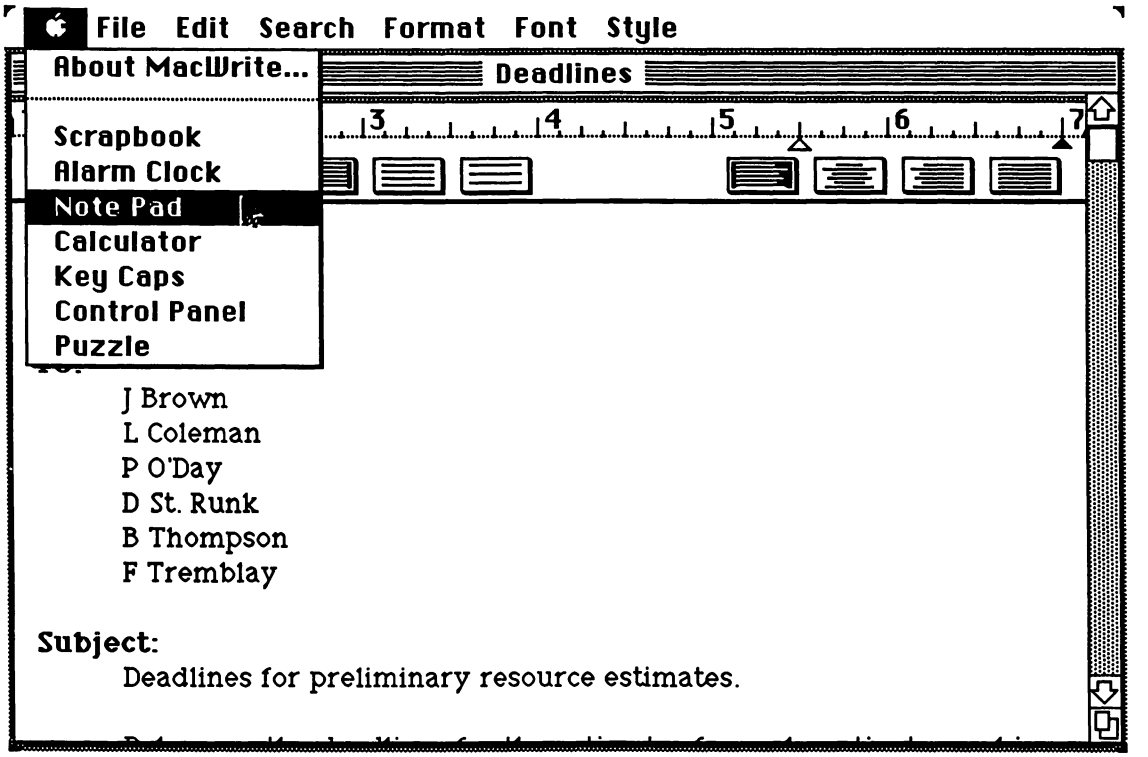


Figure 3.7 Desk Accessory menu

heavily on the graphics capabilities of the Macintosh to implement our desktop conceptual model. Now let's examine the graphics display a little more closely.

THE DISPLAY AND GRAPHICS

The Macintosh screen is a bit-mapped graphics display with a 512-by-342-pixel resolution. A pixel is the smallest point on the screen that may be turned on or off. When a pixel is off, it is seen as a dark point; when it is on, it lights up. A bit-mapped display is one in which each pixel has a corresponding bit in the machine's random access memory (RAM). When the bit in memory is set to 1, the pixel is turned on; when the bit is set to 0, the pixel is turned off (figure 3.10).

The Macintosh draws pictures on the screen by turning on a series of pixels. To draw a line, you turn on all of the pixels that would fall on that

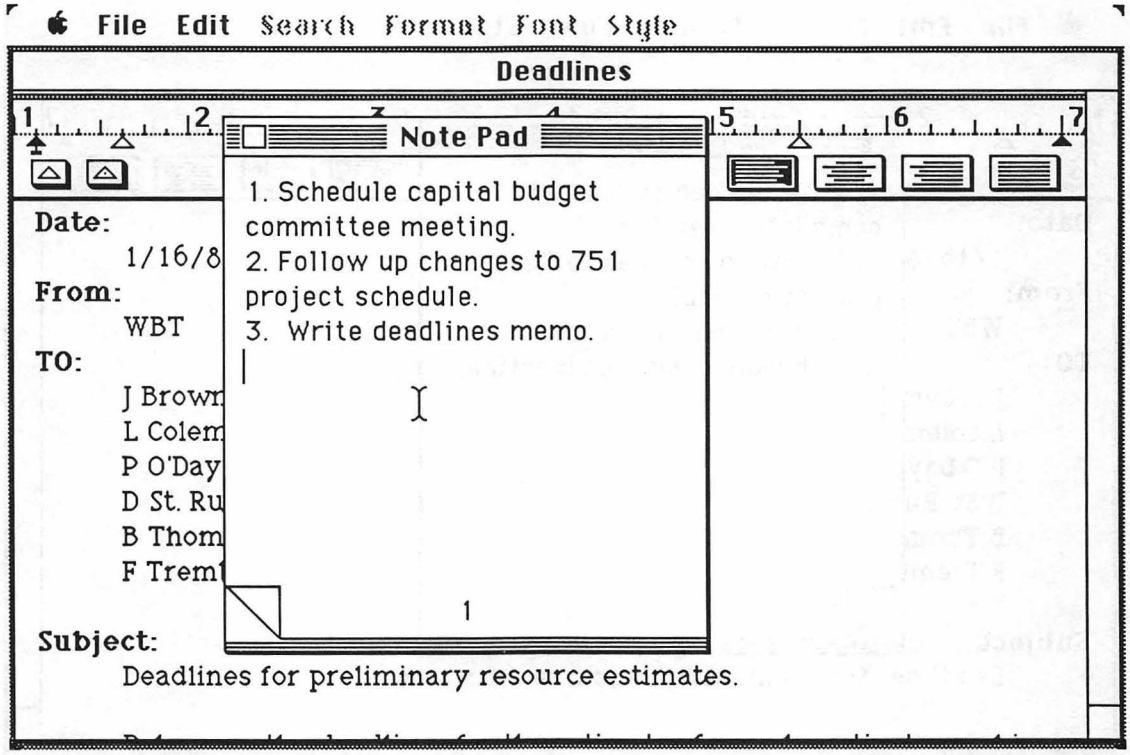


Figure 3.8 Note Pad

line if you were to draw it on the screen with a ruler. If a line is not exactly horizontal or vertical, it will not fall exactly on a row of pixels. The pixels turned on will then be those that are closest to where the line should be, often giving a straight line a jagged appearance that is technically called *aliasing* (figure 3.11). You draw all other shapes by using the same method—turning on the pixels that will most closely represent the shape you want to draw.

The Macintosh display does not allow you to set the brightness or intensity of a pixel. But you can simulate that effect by employing a technique similar to that used to reproduce pictures in newspapers. You can make an area of the screen appear brighter or dimmer by selective shading. For instance, you can make an area appear black by turning off all the pixels within its boundary. Or you can turn on all of its pixels and make the area appear white. If you turn on every other pixel instead of all of them, you can make the area appear gray. You achieve shades of gray by turning on more or fewer pixels (figure 3.12).

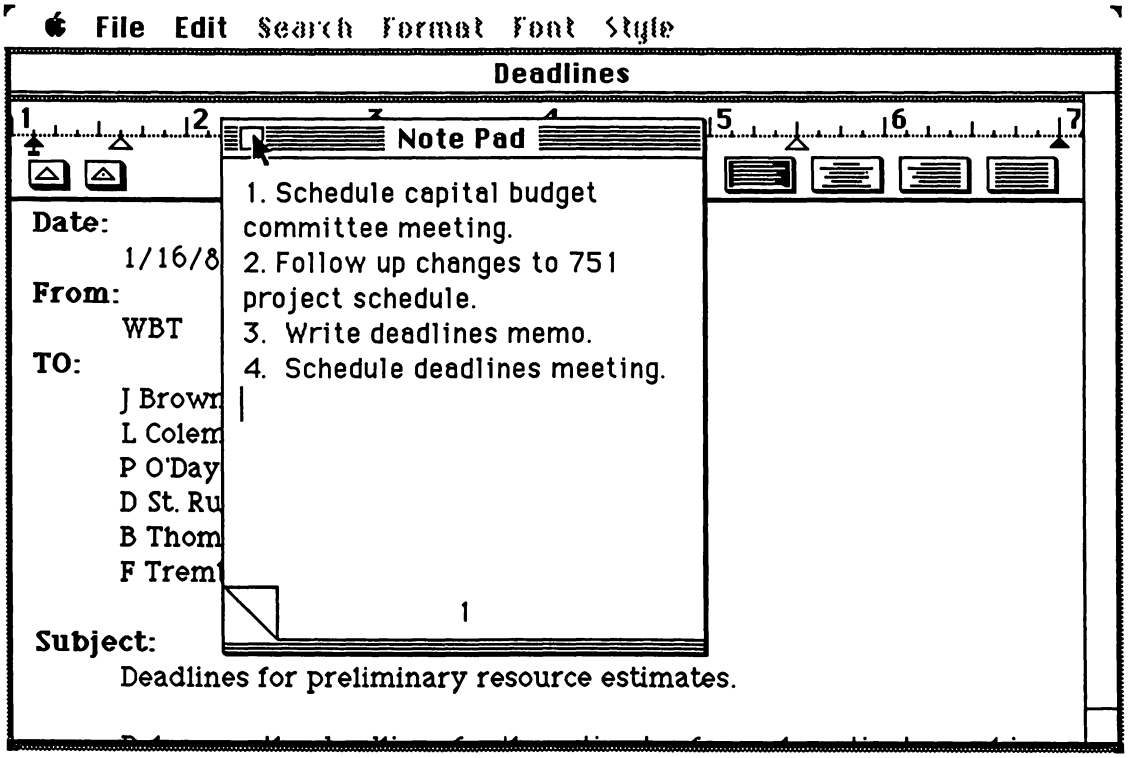


Figure 3.9 Closing the Note Pad

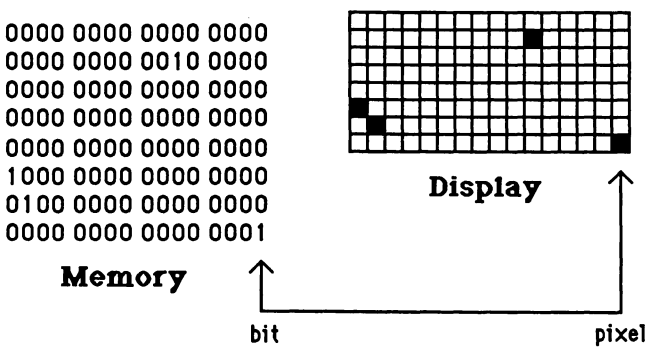


Figure 3.10 Memory-mapped pixels

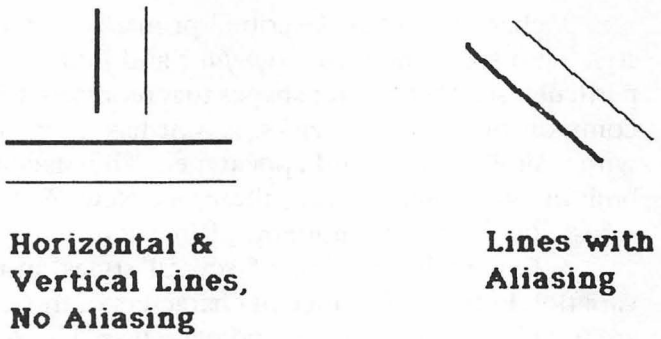


Figure 3.11 Lines with and without aliasing

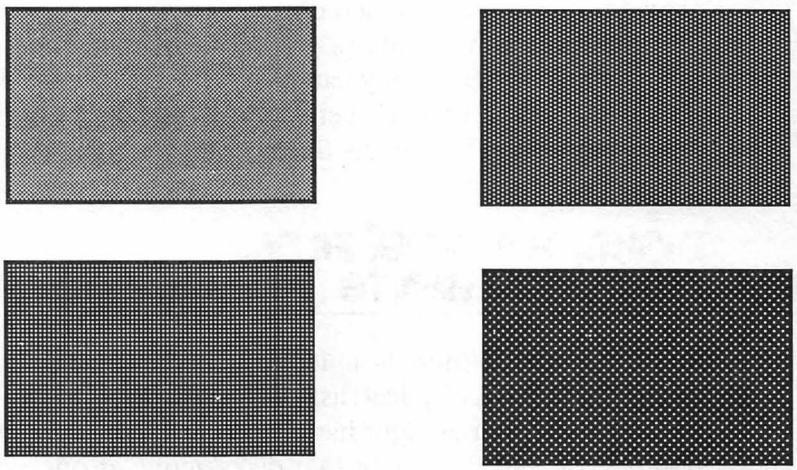


Figure 3.12 Examples of shading

CHARACTER FONTS AND STYLES

The Macintosh has no text mode. All of the characters you see on its screen are drawn as individual graphics pictures. This means that you can alter the appearance of the characters to suit your needs. You can make the characters different sizes, shapes, or styles.

Since almost every application you create for the Mac will have some characters displayed, you should become familiar with the terms used by typesetters to describe character shapes and sizes.

A character set is described primarily in terms of its typeface, font, style, and size. The terms *typeface* and *font* are both used to describe a particular set of character shapes that represent the complete alphabet and common punctuation marks. A font has a set of characters all designed with a similar shape and appearance. The Macintosh comes with several built-in type fonts; among these are New York (a traditional font with serifs) and Geneva (a modern, plainer font without serifs).

A font includes type of several styles and sizes. *Style* refers to a variation in the appearance of characters of the same font. Different styles are boldface, italic, outline, and plain text. The size of type is measured in points; one point is 1/72 inch. See figure 3.13.

The pull-down menus in MacWrite allow you to select different type fonts, styles, and sizes (figure 3.14). Some programming languages for the Macintosh allow you to set the font, style, and size of type that your program displays on the screen.

Not only can the Mac's graphics capabilities be used to draw characters on the screen, they can also be used to draw icons that look like ordinary desktop objects. Let's look at icons representing files and disks to see how desktop objects fit into our conceptual model.

DISKS, FOLDERS, FILES, AND DOCUMENTS

In the Macintosh, there is a hierarchy of data-storage icons: disks, folders, and documents. A physical disk, not a disk drive, is represented by an icon, usually in the upper right-hand portion of the screen. With a two-drive system, you can have only two disks active at one time, but more disks open. Each disk would have a disk icon on the screen, but the icons for

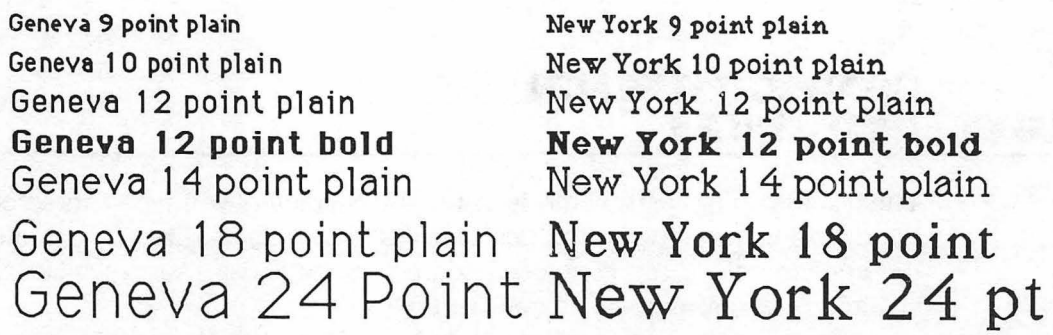


Figure 3.13 New York and Geneva typefaces

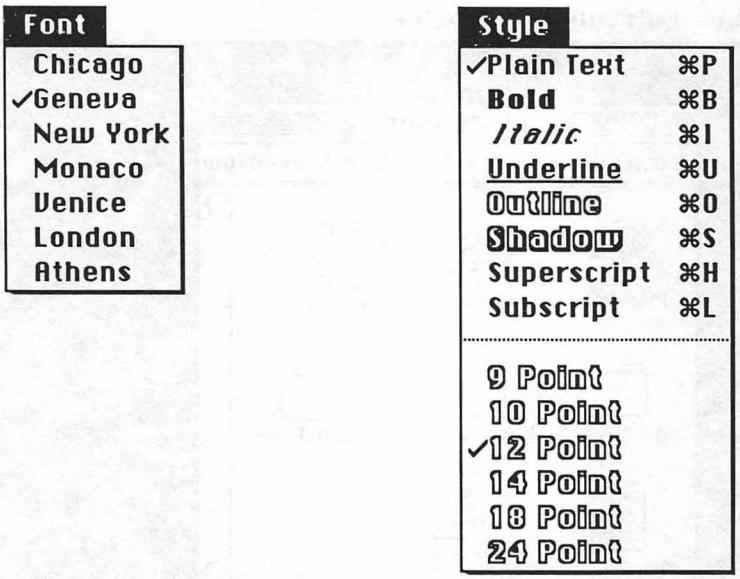


Figure 3.14 MacWrite Font and Style menus

the disks that are not in a drive would be dimmed. (A dimmed icon displayed on the screen appears to be a lighter shade of gray than an undimmed icon.)

In figure 3.15, the bottom disk icon representing the text disk is dimmed. The Office Stuff disk is selected; you can tell by its negative image (black instead of white). The Write/Paint disk is open, and you see its directory window. In the window are icons representing the MacWrite and MacPaint applications and a disk-copying application. There are two MacWrite documents, a memo and a letter. Note the two folder icons. If you were to open a folder, you would see a folder directory window just like the disk directory window. Disks can contain folders or documents. A folder contains documents or other folders.

The Macintosh user is never concerned with files and doesn't even know they exist. For the user, there are only disks, folders, documents, and tools (applications). The programmer, however, does know about files and must know how to deal with them.

The Macintosh has different types of files: data files (documents), applications (programs), and resource files. Each data file has information about which application program created the file. When you open a data file, the Finder automatically loads and runs the application that created that file.

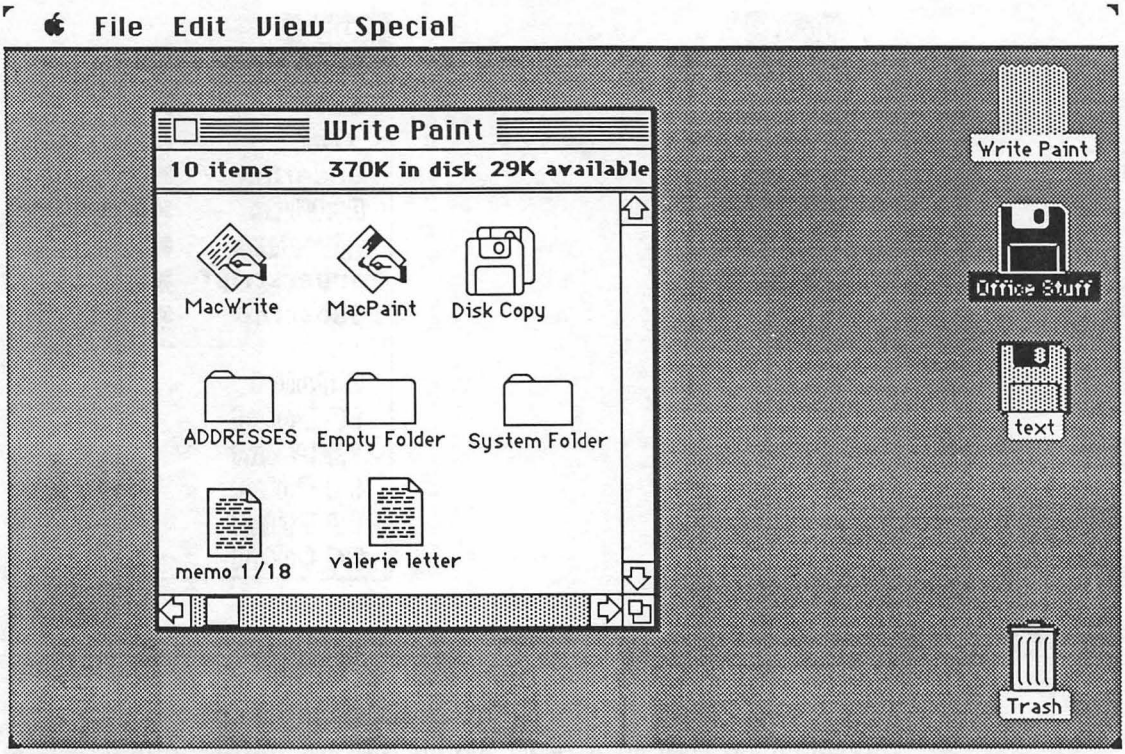


Figure 3.15 Disks, folders, and documents

Resource files are system files that contain both data and executable code for use by the operating system and application programs. Use of resource files is transparent to the user. The references to resources are automatically resolved by the system; the user never knows they exist.

There is a system resource file that contains all of the resources used by the system and most of the resources used by applications. Each application may also have its own resource file that supplements the system resource file. The system knows that each resource file (other than the system resource file) is associated with an application. If you copy an application to another disk or move it to another folder, the Finder moves its resource file also.

Data in the Macintosh is stored in document files. Each type of document has its own characteristics. A MacPaint document consists of a bit-mapped picture that can be displayed or printed. A Multiplan document consists of an array of cells for doing numerical calculations. A MacWrite document consists of text, typeface indicators, and format definitions.

If you have used a UNIX system, you are already aware of how valuable subdirectories can be. The Macintosh has something similar—folders. A folder contains a subset of the files in the main disk directory but is not a true subdirectory. You can't have two files with the same name in different folders on the same disk.

A user keeps documents, applications, or other folders in a folder. You open a folder the same way you open a disk, either by double-clicking it or by selecting it and choosing the Open option from the File menu.

In keeping with our desktop concept, we move or copy a document, folder, or application by dragging an icon from one place to another. We could drag a file from one disk to another (by copying it) or from one folder to another (by moving it). We can copy the entire contents of a folder by dragging its icon to another disk's icon or directory window.

What would be the natural thing to do with a desktop object that you want to dispose of? Throw it in the trash! The trash can icon is in the lower right corner of the screen. Moving an object to the trash can icon marks it for deletion from the system. The object actually gets deleted when the disk containing it is ejected, but you may delete everything in the trash by selecting the Empty Trash option from the Special menu.

THE CONCEPTUAL MODEL REVIEWED

Let's review our conceptual model of the Macintosh.

- ☐ Actions take place on the desk top.
- ☐ The user knows only about disks, documents, tools, and folders.
- ☐ A document contains information.
- ☐ A document is copied, moved, or deleted by having its icon moved.
- ☐ A document is associated with the application that created it.
- ☐ An application (tool) manipulates or transforms information.
- ☐ A window displays information.
- ☐ A resource file contains system data and programs.
- ☐ An application may have an associated resource file.
- ☐ A dialog box seeks information from the user.
- ☐ An alert box notifies the user of an unusual situation.
- ☐ A desk accessory is a program that can be used while another application is active.

CHAPTER

4 CARVED IN SILICON, THE ROMS

Inside the ROMs

The User-Interface Toolbox

The Operating System

Languages and the Toolbox

If you opened your Macintosh (don't; opening voids your warranty), you would find 64K bytes of ROM (read-only memory). Apple has programmed the ROM with programs and data structures commonly used by the user interface and operating system. With these programs and data structures in ROM, you are assured that they will always be there. They needn't be loaded from a disk, and they do not occupy scarce RAM (random access memory).

A few of the operating-system and toolbox routines reside in RAM and are loaded from a disk, but most are in ROM. If a ROM routine needs to be updated, a new version may be loaded into RAM from a disk; the Macintosh will then use the new RAM routine instead of the old ROM routine.

INSIDE THE ROMS

The Macintosh has an operating system and user interface that consist of an amazing number of modules and subroutines. At first glance, their complexity seems to rival that of an operating system for a large minicomputer. Fear not. You don't need to become conversant with each little subroutine in order to write programs for the Macintosh. Most of the time you will use only a small group of modules for setting up menus, managing windows, and performing similar tasks. You won't use most of the other modules on a regular basis, but you should be able to recognize their names and their functions so that you can find more detailed information when you need it.

For now, let's just look at the different modules and briefly examine the purpose of each. We will look more closely at a few of these modules as we do some programming later in this book and the other books in this series. For further information, you may want to refer to Apple Computer's documentation for software developers. It goes into the operating-system and toolbox routines in excruciating detail and is useful if you are developing sophisticated software for the Macintosh.

As we have seen, there are two sections of the Macintosh system software, the user interface and the operating system. First let's take a look at what's in the user interface; then we'll see what the operating system is like.

THE USER-INTERFACE TOOLBOX

The user interface consists of the Finder and a collection of subroutines called the user-interface toolbox. The Finder is the program that handles

most of the ordinary user-interface tasks for the operating system. It creates the desktop display, draws the disk directory windows, and displays the menu bar that you see when the operating system is in control (the Apple, File, Edit, View, and Special menus). The Finder uses toolbox routines to build the user interface just as application programs do.

The toolbox routines are organized into several modules. In the following list, these modules are organized in hierarchical order; many routines in modules at the top of the list call routines in modules lower in the list.

Module	*Function*
Desk Manager	Manages the desk accessories (Note Pad, Calculator, Control Panel, Alarm Clock, and so on)
Scrap Manager	Manages the Scrapbook and the Clipboard
Dialog Manager	Manages the dialog and alert boxes
Control Manager	Manages controls such as buttons, scroll bars, and check boxes
Menu Manager	Manages menus
Text Edit	Controls editing via the Edit menu (Cut, Copy, and Paste)
Core Edit	Resides in RAM and controls the more advanced editing functions, such as justifying copy
Window Manager	Manages windows: creates, activates, and moves them; changes window size; and so on
Toolbox Utilities	Potpourri of routines for mathematical functions, string manipulation, and the like
QuickDraw	Set of routines for drawing and manipulating graphics images on the screen
Toolbox Event Manager	Reports events that an application must handle, such as keyboard input, software events, and clicking of the mouse button
Font Manager	Routines for providing information about fonts to QuickDraw
Resource Manager	Manages the resources in an application's resource file and the system resource file

Module	*Function*
Package Manager	Manages packages of routines and data structures that are stored in resource files
System Error Handler	Detects system errors and displays the alert box with a bomb icon

THE OPERATING SYSTEM

Like the toolbox, the operating system consists of program modules that reside in ROM as well as in RAM. Here, too, the ROM routines may be updated and replaced by RAM routines. Here's the list of the operating system (OS) modules:

Module	*Function*
Memory Manager	Manages the memory space: assigns memory to applications and releases it when an application is finished
Segment Loader	Loads an application program into memory from a disk
File Manager	Manages access to disk files
Device Manager	Manages access to input/output (I/O) devices
Disk Driver	Routines to read from and write to the disk
Sound Driver	Routines to control the sound synthesizer hardware
Serial Driver	Routines to control, read from, and write to the serial I/O ports (printer and modem ports)
OS Event Manager	Handles hardware events
Keyboard and Mouse Handler	Routines to access the keyboard and the mouse, and to report their events to the OS event manager
Vertical Retrace Manager	Controls the execution of routines during the display's vertical-retrace time interval; used to update the display without flickering

Module	*Function*
OS Core	Kernel of the operating system: contains the lowest-level OS routines for initialization, handling interrupts, and scheduling routines to execute
OS Utilities	Collection of miscellaneous operating-system routines that don't fit into any other category

In addition to the toolbox and operating-system modules, several other modules are part of the Macintosh system software: the printing routines, floating-point math routines, transcendental functions, standard file package, international utilities package, and disk-formatting package. All of these reside in RAM.

LANGUAGES AND THE TOOLBOX

The programming language you are using determines which routines you may use. Some languages don't allow access to the toolbox routines at all. Other languages provide the functionality of the toolbox routines but do so with commands built into the language. You need become familiar only with the built-in commands because the compiler or interpreter actually calls the toolbox routines. Still, these built-in commands are only a subset of the toolbox routines, so some languages with built-in commands also allow direct access to the toolbox routines.

If you are programming in C or assembly language, you need to be familiar with the toolbox routines themselves. You won't have built-in commands to perform their functions. The compiler should come with "include" or "header" files that define the toolbox subroutines and their locations.

If you are programming in BASIC, Pascal, or another higher-level language, you need to know which toolbox functions are supported by the language. You'll find information on languages and toolbox support in chapter 5, Macintosh Pascal; chapter 6, Microsoft BASIC; and chapter 7, Macintosh BASIC.

CHAPTER

5 MACINTOSH PASCAL

Introducing Pascal

The Pascal Language

Macintosh Pascal

Macintosh Pascal Compatibility

Toolbox and Library Routines

This chapter deals with Pascal at several levels. The first section introduces Pascal for the Macintosh. The second section briefly describes the Pascal language for programmers who have not used it. If you are already a Pascal programmer, you should skip to the third section, Macintosh Pascal. The last two sections assume you have a thorough knowledge of Pascal; they discuss Macintosh Pascal's access to toolbox routines and its compatibility with the ANSI Pascal standard, Lisa Pascal, and Apple II Pascal.

INTRODUCING PASCAL

Pascal was originally developed as a teaching language, one that would enforce structured programming rules. Students learned to organize their programs with good structure while they were learning the basics of programming.

Because Pascal is so stringent about enforcing structured programming rules, it is a good candidate for business software and other applications. The structure and strong data typing of Pascal programs make them work better and require less maintenance.

There are a number of Pascal compilers available for the Macintosh. Some are conversions of compilers that run on other machines; you can expect to convert programs written for these compilers with a minimum of frustration. The Pascal system from Apple Computer is something special, though.

Apple has broken new ground with Macintosh Pascal. It is interpretive, in much the same way as BASIC. It reads, interprets, and executes one line of the program at a time. The interpreter can be stopped at any point to allow you to look at variables, and it can be restarted where it was stopped. You may step through a program, executing one line of code at a time, or you may tell the interpreter to stop at chosen places in your program.

Macintosh Pascal is an ideal language for designing application prototypes. It is easier to debug and modify programs with Macintosh Pascal than it is with compilers that require batch-oriented edit, compile, and link steps. Macintosh Pascal is slower in execution than compiled Pascal, but it is invaluable for producing working programs quickly and providing a test bed for new programs.

Once a program is working in Macintosh Pascal and you are satisfied with its user-interface design, you can convert the program into a compiled Pascal, C, assembler, or any other language you choose. If the conversion follows the Pascal version closely, you end up with a well-debugged, well-structured program in a language that executes with reasonable speed.

Macintosh Pascal was developed for Apple Computer by THINK Technologies, which is working on a compiled version of Macintosh Pascal that will make the path from prototype to finished product even smoother.

THE PASCAL LANGUAGE

There's no way to cram an entire Pascal textbook into a single chapter, but this section should give you a feel for the character of the language and its major features.

One of Pascal's features is its strong data typing. In Pascal, all variables and constants must be declared in advance; there is no default data type. The language does a limited set of implicit type conversion. You may not assign a value from a variable of one data type to a variable of a different type unless the two data types are "assignment compatible" (as defined in the *Macintosh Pascal Reference Manual*). The most commonly used assignment-compatible data types are listed below. Of course, types that are identical are assignment compatible.

Assign variable of type:	*To variable of type:*
Integer	Real
String of length 1	Character
Character	String
String	Packed string of the same length
String of length *n*	String of length *n* or $>n$
Packed string of length *n*	String of length *n* or $>n$

Pascal has a wide variety of data types and allows you to define new data types. By comparison, BASIC falls far short in number of data types, with just three: integer, floating-point (real), and string. Most Pascal compilers support the following predefined data types: integer, long integer, real, scalar, Boolean, character, pointer, set, string, record, and file. (Macintosh Pascal calls the scalar type the *enumerated* type.)

Pascal also allows you to define data structures and reference elements of a structure by structure name and element name (Pascal calls a data structure a *record*).

Pascal programs must conform to a fairly rigid format that restricts the programmer to structured software designs. The format has specific ways to define the program itself, procedures, functions, and variables so that a procedure, function, or variable's scope is unambiguous. (A *procedure*

is a subroutine that does not return a data item to its caller; a *function* is a subroutine that does.)

The term *scope* refers to how widely known a part of a program is. For instance, a variable defined in the main program is known to all parts of the program, all procedures and functions. It can be read or changed by any subroutine. A variable defined in a procedure is not known outside of that procedure. Another procedure could not read or change the variable. In fact, if another procedure had its own variable of the same name, there would be no conflict. Each would have its own unique variable. The exception to this rule is variables in nested procedures.

A *nested procedure* is a procedure that is completely contained within the definition of another procedure (its parent procedure). The variables in the parent procedure are known to the nested procedure but not to other procedures not nested within the same parent.

Now let's take a look at what makes up a Pascal program—how the program and its procedures, functions, and variables are defined—to see how the rules of scope apply. A Pascal program is organized as blocks of program lines (source code). There is a block that defines the overall program. It contains other blocks that define procedures and functions.

Figure 5.1 is a structure diagram for the program named Example. It calls two procedures, named A and B. Procedure B calls another procedure, named C. Figure 5.2 shows how the program Example is organized in Pascal. The program contains procedures A and B, and procedure B contains procedure C.

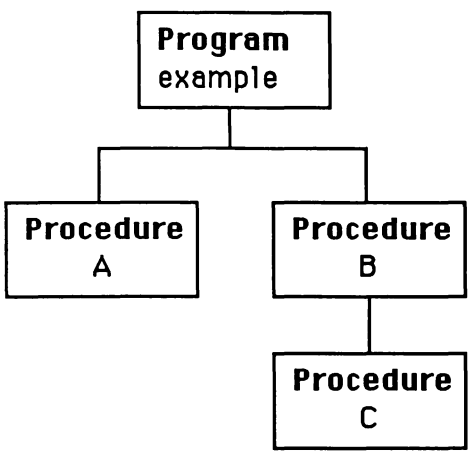


Figure 5.1 Example program structure

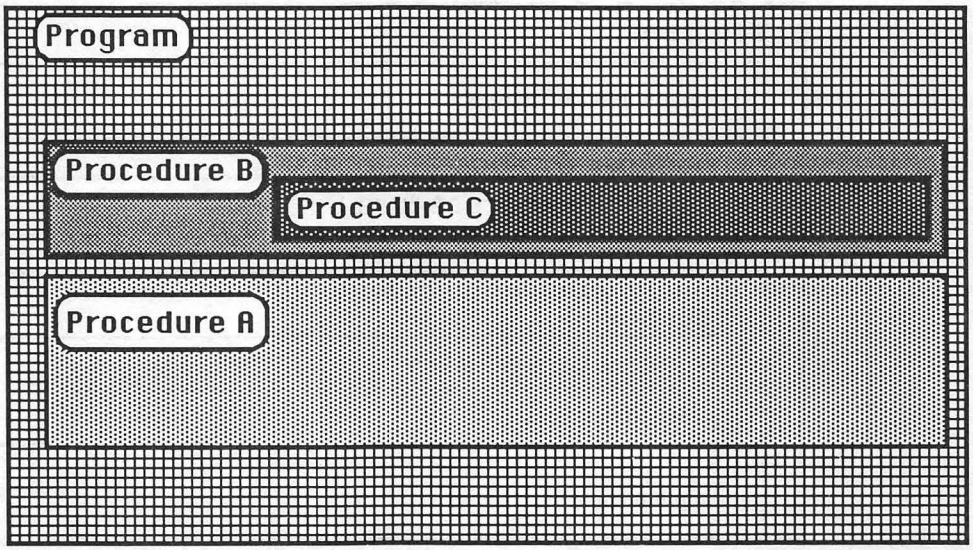


Figure 5.2 Pascal program organization

In our program Example, procedure C is nested within procedure B. Procedure C knows about all of the variables defined in procedure B. Procedures B and A do not know about the variables in procedure C. Note that procedures A, B, and C are nested within the main program. Procedures A, B, and C know about all of the variables defined in the main program. The main program cannot access a variable that is defined in a function or procedure.

These rules of scope also apply to the definitions of procedures and functions. For example, a procedure defined within another procedure can be accessed only by the procedure in which it is defined; in the program Example, procedure C can be called from procedure B but not from procedure A or the main program.

The restrictions on accessing variables and procedures make programs easier to debug and more reliable in operation. The restrictions eliminate the possibility of a subroutine's accessing the wrong variable or changing a variable that would affect another subroutine. This rigid structure helps to confine errors to the subroutine in which they occur.

A Pascal program is defined at two places. Its data is defined at the beginning, and its executable code is defined at the end. Everything that comes between the program's data and code definitions is contained within the program.

Figure 5.3 shows where the program and procedure A are defined. The definition of each procedure or function looks just like the defini-

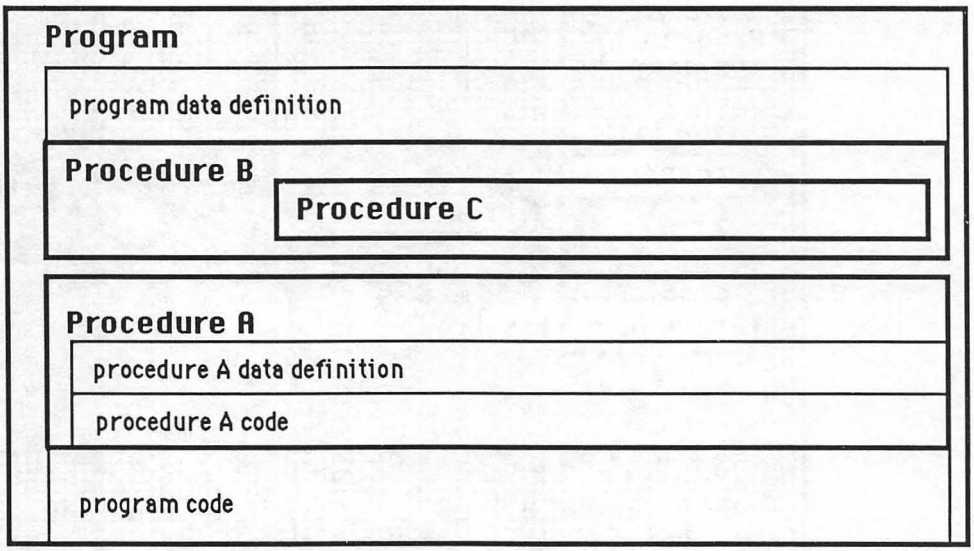


Figure 5.3 Program and procedure definition

tion of a program. Its data is defined first, then its code. The data definition contains definitions of variables, constants, and user-defined data types. The code block contains all of the Pascal executable code for the procedure.

A procedure may contain other procedures between its data definition and code. These procedures would then be limited to the scope of the procedure in which they are defined. Outside procedures would not know about them and could not call them.

In figure 5.4, our diagram is expanded to show what the data definition looks like and how the program code is delimited. The boldface words are Pascal reserved words that have special meaning to the compiler and cannot be used for user-defined data. The reserved word *const* indicates the start of the declarations of constants. *Var* indicates the start of variable declarations. A section of program or procedure code starts with the reserved word *begin* and ends with *end*.

Note that procedures A and B are placed between the program's data declarations and its code. Procedure C is contained within procedure B; it is placed between procedure B's data declarations and its code.

In an actual Pascal program, the structure we have shown in our block diagram is indicated by indentation. Listing 5.1 illustrates what an actual Pascal program for our example would look like.

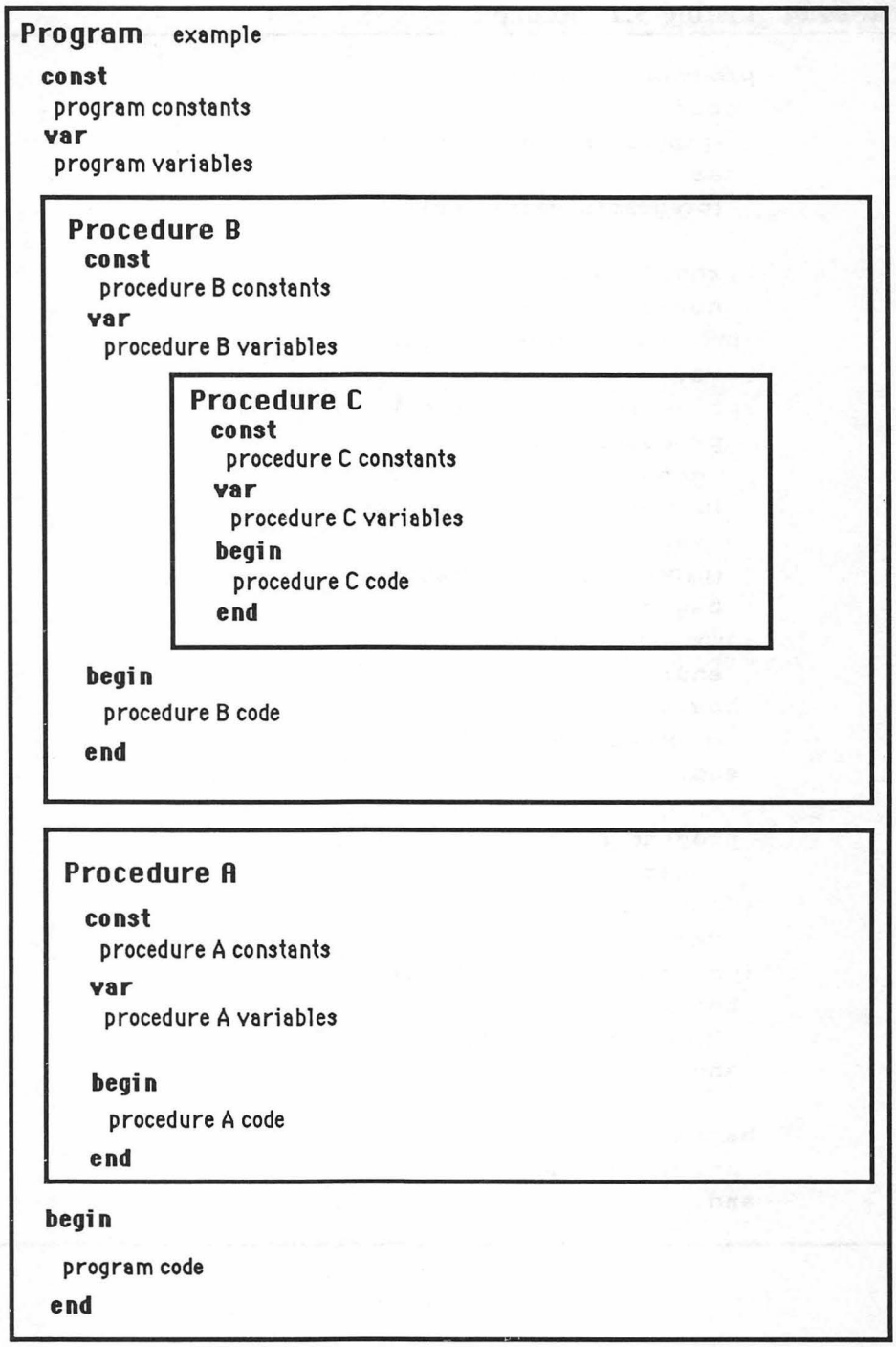


Figure 5.4 Data and code definitions of program Example

Listing 5.1 Example

```
program example;
 const
  {program's constants}
 var
  {program's variables}

 procedure B;
  const
{procedure B's constants}
  var
{procedure B's variables}
  procedure C;
   const
  {procedure C's constants}
   var
  {procedure C's variables}
  begin
{procedure C's code}
  end;
 begin
  {procedure B's code}
 end;

 procedure A;
  const
{procedure A's constants}
  var
{procedure A's variables}
 begin
  {procedure A's code}
 end;

begin
{program's code}
end.
```

Pascal has many predefined data types and allows the user to define new data types. Pascal also has a number of predefined procedures and functions (usually called *implicit*, or *built-in*, functions). Each Pascal compiler lists its predefined data types and implicit functions in its documentation. Many compilers come with extensive libraries of procedures for various applications. Apple II Pascal has a library of turtle graphics procedures, and Macintosh Pascal has libraries for graphics, mathematical functions, and Macintosh toolbox access.

Pascal's characteristics are:

1. Strong data typing
2. Many predefined data types
3. Limited implicit data conversion
4. Enforced program structure
5. Enforced rules of scope
6. Libraries of procedures and functions

MACINTOSH PASCAL

Let's explore Macintosh Pascal by looking at a Pascal program named Yeti and seeing how we can use the features of Macintosh Pascal to debug it. Yeti is a program that queries the user for the parameters (four integers) to draw a graphics pattern on the screen.

An elusive creature, the yeti is reported to live deep in the remote regions of the Himalaya Mountains. After many hardships and expensive expeditions, we have discovered a fingerprint of the shy beast and imbedded its pattern in a mathematical equation that is the heart of this program. The program draws the fingerprint of the yeti. We don't want to spoil the fun, so we won't show the pattern here. To satisfy your curiosity, type in the program and run it.

Figure 5.5 shows the structure of the program. The main section of the program calls four procedures and a function defined within the program. The main program and the Plot procedure call several of the QuickDraw procedures from the Macintosh toolbox.

The GetParameters procedure queries the user for the parameters to be used by the YetiCalc function. The SetUpDrawing procedure calls QuickDraw routines to set up the Drawing window. PenNormal is a QuickDraw procedure that sets up the pen-definition data structures for drawing on the screen.

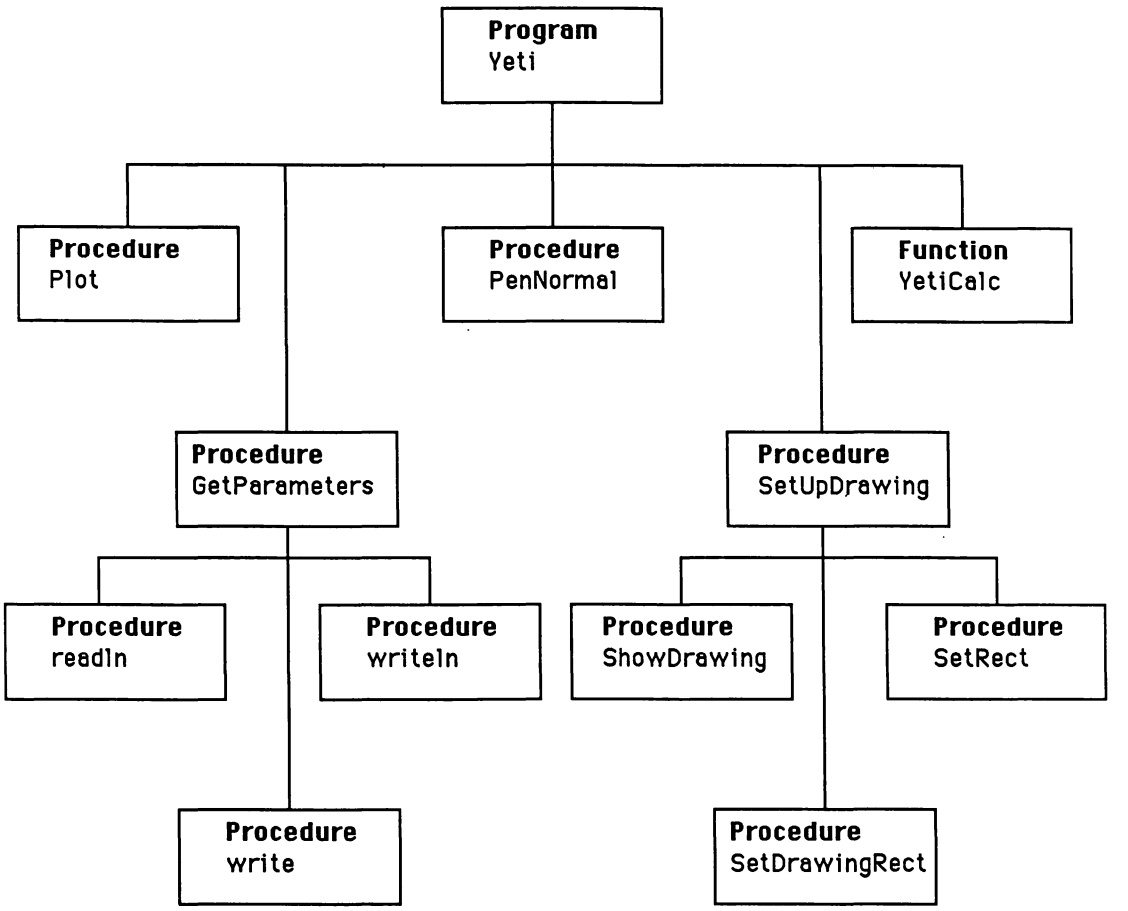


Figure 5.5 Yeti program structure

The YetiCalc function takes the x and y coordinates of a point on the screen and decides whether or not to turn on a pixel at that point. The Plot procedure sets the pixel. In this case, we are drawing a black image on a white background, so Plot actually turns off the pixel.

If you look at the Yeti program (listing 5.2), you will see how the structure diagram is reflected in the source code of the program. Note, again, that in Pascal the main part of the program comes at the end of the listing; everything it uses is defined further up in the listing.

Now let's put the Macintosh Pascal disk in the Macintosh and see what we have. Figure 5.6 shows the Macintosh Pascal disk directory window. A new Macintosh Pascal disk contains several documents that are

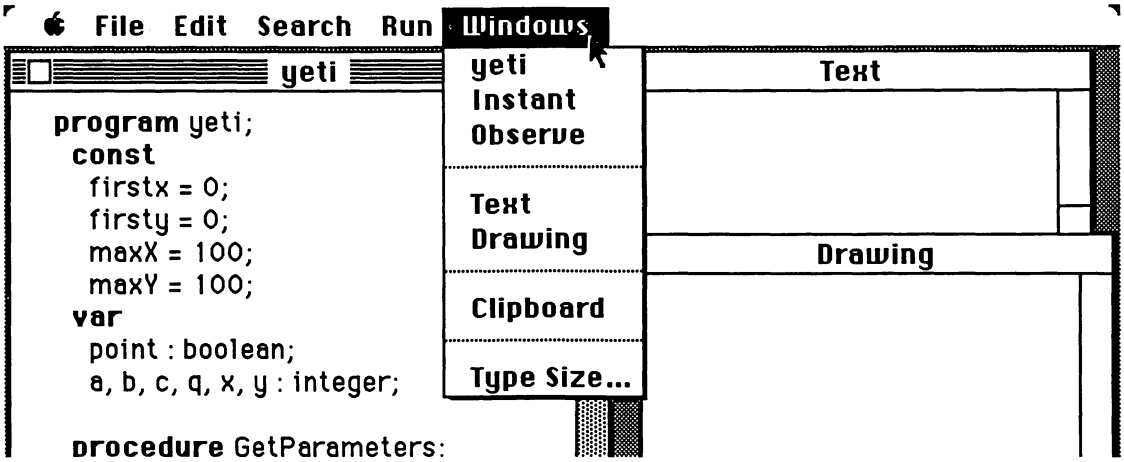


Figure 5.8 Windows menu

line of the program to execute, the line identified by the pointer (the pointer icon is a hand with a pointing finger).

Let's choose Step from the menu and see what happens. The Yeti window scrolls to show the pointer sitting at the begin statement for the main part of the program. If we choose Step again, the pointer stops on the GetParameters statement.

We can continue stepping through the program one line at a time, or we can get through it faster by several methods. Choosing Step-Step causes the program to step continuously until stopped. The value of Step-Step is that it updates the Observe window after each line of the program is executed.

Suppose we want to execute quickly and stop at a predetermined point in the program. We can do that by inserting stops. A stop is a little stop sign that appears in the left margin of the program (Yeti) window. When the program gets to the line with the stop sign, it stops and updates the Observe window.

Let's put a stop sign at the end statement in the GetParameters procedure. To do that, we just move the cursor to the left margin of the program listing, and the cursor becomes a stop sign. When we click the mouse button by the end statement, a stop sign is left there (figure 5.11).

Now we'll choose Go from the Run menu. The program executes, and when it reaches the GetParameters procedure, it prompts us for input in the Text window. We supply the requested integers, and at the end of the GetParameters procedure, the interpreter encounters the stop sign and

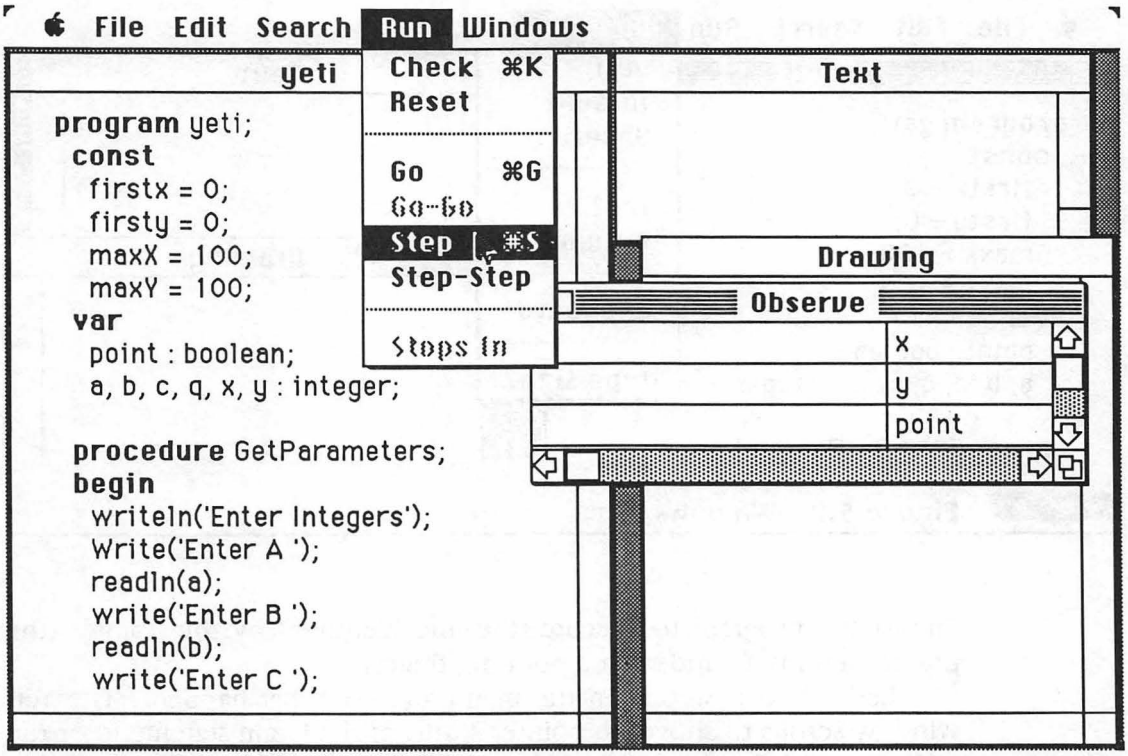


Figure 5.9 Observe window and Run menu

stops execution. Figure 5.12 shows the screen after the program has stopped at the stop sign.

At this point we could put a stop in the YetiCalc function and choose Go-Go from the Run menu. The program would hit the stop every time YetiCalc executes and then start running again. As a result, the Observe window would be updated every time YetiCalc executes.

Let's suppose that, instead, we choose to remove the stop signs and just let the program run. While it's running, we see another menu on the menu bar, the Pause menu. If we pull down the Pause menu, the program pauses. If we let the menu go without selecting Halt (the only option on the menu), the program resumes. If we select Halt, the program does not resume.

Figure 5.13 (page 70) shows the program paused by the Pause menu. The pointer in the program (Yeti) window shows where the program stopped, and the Observe window has been updated. If you really try this, you will see part of the Yeti fingerprint in the Drawing window. (We removed it to maintain the suspense.)

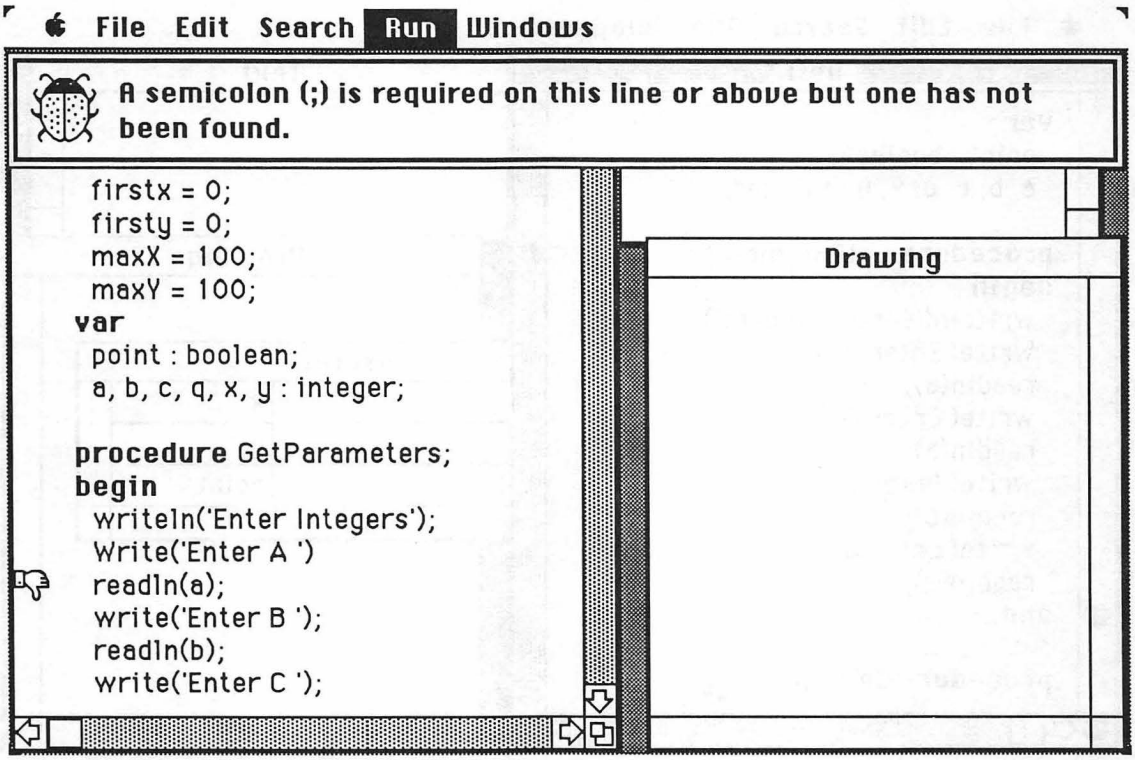


Figure 5.10 Macintosh Pascal error message

You can try different parameters with the Yeti program. Values of Q in the range of 20 through 100 produce interesting results. Also try the program with B = 0 and with unequal A and C parameters. The program is set up so that you can make the Drawing window and the fingerprint larger by increasing the values maxX and maxY. However, you will find that the program takes a very long time to fill a large window.

MACINTOSH PASCAL COMPATIBILITY

Macintosh Pascal was designed to be as compatible as possible with Lisa Pascal and the ANSI Pascal standard. This means that it differs from Apple II Pascal and UCSD Pascal, both of which were designed before an ANSI standard existed. At the time that UCSD and Apple II Pascal (a dialect of

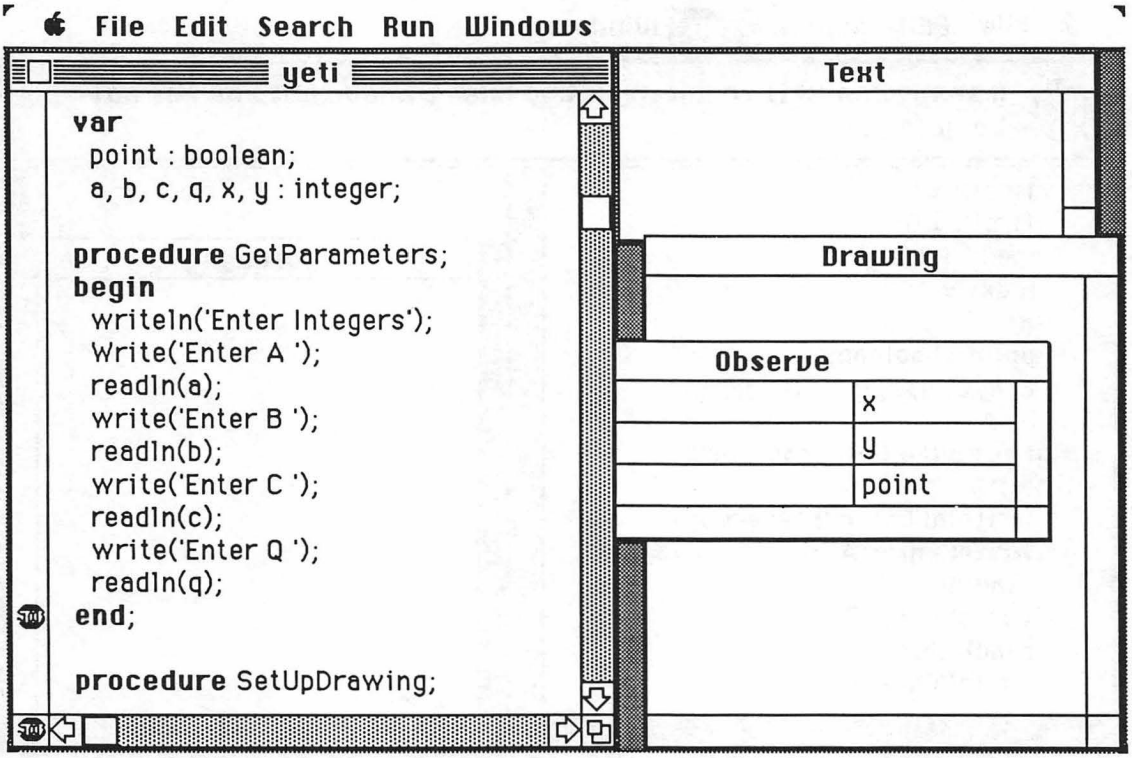


Figure 5.11 Stop sign

UCSD) were created, the closest thing to a standard was the *Pascal User Manual and Report* by Kathleen Jensen and Niklaus Wirth. It was actually more a language definition than a user's manual.

Appendix D is a list of compilers available for the Macintosh and contains a list of the Pascal compilers. Appendix A is an extensive list of Pascal reserved words, data types, and predefined identifiers for Macintosh Pascal, Apple II Pascal, and UCSD Pascal. Appendix A is the definitive list, but we can discuss the general differences among compilers.

Macintosh Pascal and UCSD Pascal differ significantly in three areas: identifiers, data types, and I/O procedures. The only I/O procedures that are even remotely alike are read, readln, write, and writeln, and these are very close to being the same. All of the I/O procedures dealing with specific devices or graphics are completely incompatible.

Macintosh Pascal has several data types that are not available with UCSD or Apple II Pascal—notably, the computational, double, and extended types. UCSD and Apple II Pascal have a data type called *scalar* (Macintosh Pascal calls the same data type *enumerated*).

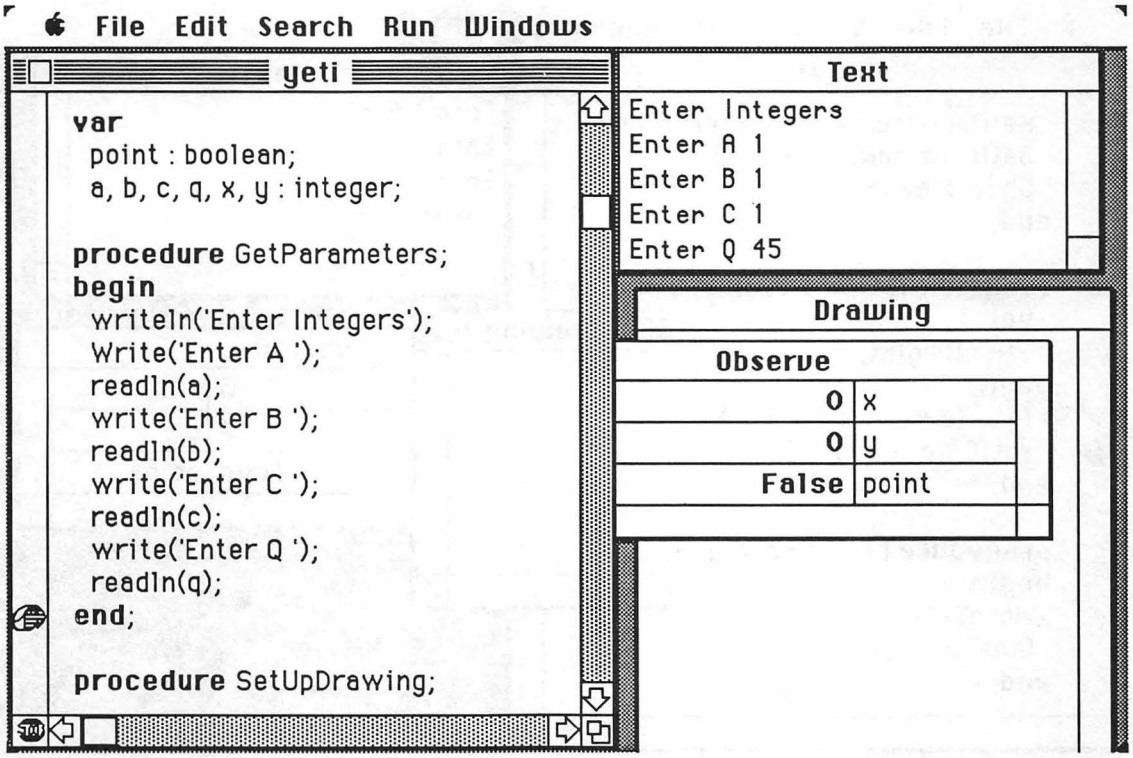


Figure 5.12 Stopped Yeti program

In UCSD Pascal, an identifier can have any number of characters, but only eight are significant. In Macintosh Pascal, an identifier can have up to 255 characters, all significant. The key differences between Macintosh Pascal and Apple II/UCSD Pascal are listed below:

Macintosh	*Apple II*
255-character identifiers	8-character identifiers
Macintosh graphics procedures	Turtle graphics procedures
Double, extended, and computational data types	No double, extended, or computational data types
Enumerated data type	Scalar data type

Macintosh Pascal differs from Lisa Pascal in several areas; for more detailed information, refer to the *Macintosh Pascal Technical Appendix* that comes with the Macintosh Pascal documentation. The technical

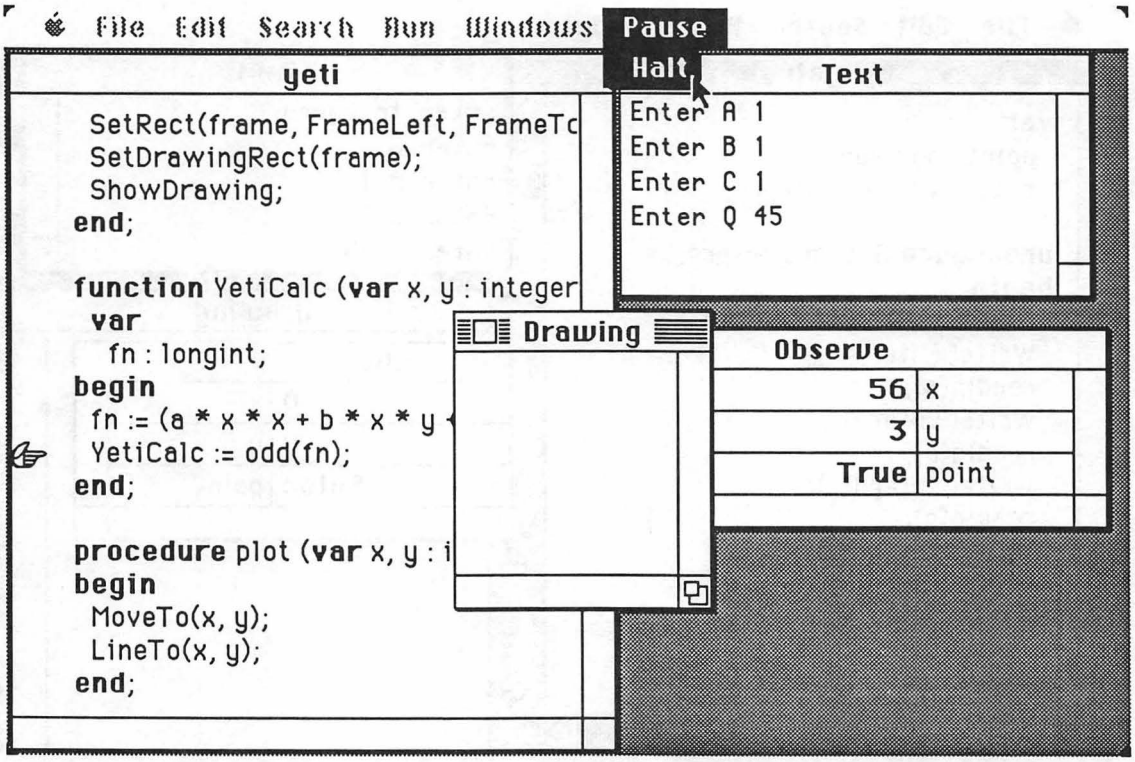


Figure 5.13 Paused program

appendix also has a section delineating the differences between Macintosh Pascal and the ANSI Pascal standard.

In Lisa Pascal, only eight characters of identifiers are significant. The Macintosh Pascal data types double, longint, and computational are not in Lisa Pascal. The results of some expressions differ in type. In Macintosh Pascal, all integer expression results are the longint type, and all real expression results are the extended type.

Some procedures and functions that appear in Lisa Pascal do not appear in Macintosh Pascal, primarily those dealing with concurrent processes and memory management. Unlike Lisa Pascal, Macintosh Pascal does not allow external procedures or functions. Macintosh Pascal does allow access to toolbox routines, though, by use of the InLine procedure (see the next section, Toolbox and Library Routines). The following list highlights the differences between Macintosh Pascal and Lisa Pascal:

Macintosh	*Lisa*
Identifiers have 255 significant characters	Identifiers have 8 significant characters
Double, longint, extended, and computational data types	No double, longint, extended, or computational data types
No memory-management procedures	Memory-management procedures
No concurrent-process procedures	Concurrent-process procedures
No external procedures	External procedures

TOOLBOX AND LIBRARY ROUTINES

Macintosh Pascal includes numerous built-in (predefined) procedures and functions, and two libraries of functions and procedures. Some deal with numeric calculations; others provide access to toolbox routines. The toolbox routines that are not accessible from predefined and library procedures can be used via the InLine procedure.

Macintosh Pascal supports numeric calculations through a library of numeric routines—called Standard Apple Numeric Environment (SANE). The library is based on the IEEE 754 standard for binary floating-point arithmetic but actually supports more than the standard calls for. SANE also includes routines for extended-precision, scientific, and financial calculations. Apple plans to implement this library across a wide range of products. For more information on the SANE library, consult Appendix D in the *Macintosh Pascal Technical Appendix*.

You can use a number of Macintosh Pascal built-in procedures and functions to access a limited set of toolbox routines. With built-in procedures, Macintosh Pascal provides access to memory-manager and window-manager routines to control the Drawing window. Built-in procedures also access event-manager routines (for controlling mouse-button operations and the like) and sound-driver routines for controlling the sound synthesizer. Other procedures access a collection of routines, such as string manipulation, fixed-point arithmetic, time and date, graphics utilities, and CRT synchronization.

The QuickDraw routines are the most fully supported of all the toolbox facilities. Access is via the Macintosh Pascal QuickDraw library. There are QuickDraw routines for defining drawing areas, windows, and clipping; for drawing predefined shapes, such as ovals, rectangles, rounded rectangles, wedges, regions, and polygons; for filling shapes with predefined or user-defined patterns; and for drawing characters in any of the fonts and styles supported by the Macintosh. A complete discussion of the QuickDraw library and the design of the QuickDraw graphics system appears in Appendix C of the *Macintosh Pascal Technical Appendix.*

The Macintosh Pascal disk contains documents with information on manual updates, error messages, hints, and the InLine facility. InLine is a predefined Pascal procedure that calls a toolbox routine for you. You supply InLine with a list containing the address of the procedure you want to call (the trap number) and the parameters required by the procedure.

InLine is for use by programmers experienced in Pascal and the Macintosh toolbox. Making an error with InLine can be catastrophic. But it is currently the only way to access the menu manager and other toolbox routines necessary for developing a fully functional application with Macintosh Pascal that conforms completely to the Macintosh user-interface guidelines. Officially, InLine is unsupported. If you get into trouble with it, you're on your own.

CHAPTER

6 MICROSOFT BASIC

Vive la Différence

In the Driver's Seat—Controlling BASIC

Writing and Debugging a Program

Graphics and Fonts on the Printer

The Picture Statement

Microsoft BASIC and the Toolbox

Creating and Controlling Menus

Controlling Windows

Buttons

A Menu and Window Example—the Memory Dump Program

After all this hoopla about sophisticated, interactive user interfaces, structured languages, rules of scope, menus, and windows, we are back to old, familiar, comfortable, if clunky, BASIC. Right? Well, maybe. Take a glance at figure 6.1.

Looks like windows. Where are the line numbers? What's all the text to the right of the lines of code? Could it be comments on the same line as code? Some lines of code are indented, and we see what appears to be a label—in BASIC? The BASIC commands are in boldface. This looks serious. It's different! Where's old, familiar Microsoft BASIC?

VIVE LA DIFFÉRENCE

It is different, but the old BASIC is still there, along with a lot more. Microsoft BASIC for the Macintosh has all of the old BASIC commands and constructs that you are accustomed to, but it also has a number of enhancements that allow programs to have more structured designs and to

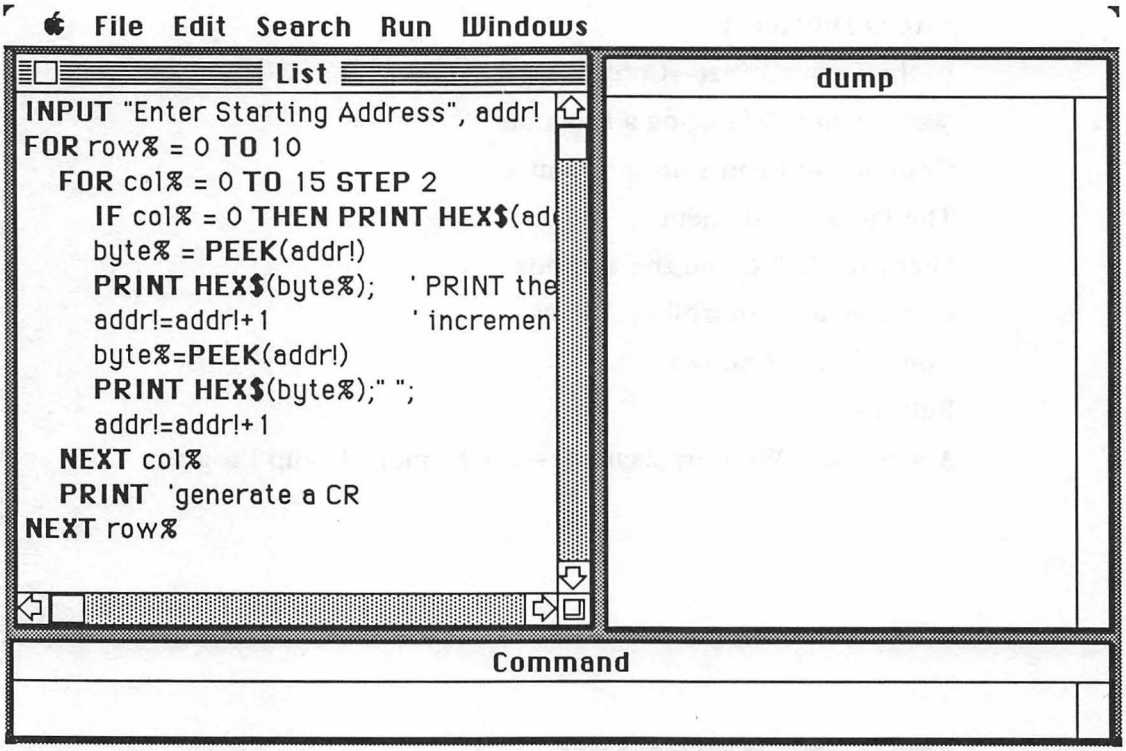


Figure 6.1 Microsoft BASIC

support the Macintosh user interface. Also, the BASIC interpreter's operation is controlled with menus and windows, so it conforms to the Macintosh user-interface guidelines.

Like previous versions of Microsoft BASIC, the Macintosh version has the usual set of string functions, math functions, and a complete collection of disk-file I/O commands for random and sequential files.

Microsoft BASIC for the Macintosh also has data types not usually found in BASIC, and the language allows meaningful names (of up to forty characters) for variables instead of one- or two-character identifiers.

This new version of BASIC gives you more latitude in formatting programs. You may use line labels as well as line numbers, although neither is required (except on lines that are the target of a branching or calling instruction, such as GOTO or GOSUB). You may indent program lines to make the program more readable and highlight its structure. You may put comments on any line of code or use the old standard REM statements for remarks.

On the Macintosh, BASIC starts to look more like a structured language. But there's more than just the appearance of structure: Microsoft BASIC for the Macintosh has a new program construct called the *subprogram*. A subprogram is much more like a Pascal procedure than a BASIC subroutine. It has local variables that are unaffected by the operation of the main program or any other subprogram. There's no way to accidentally execute a subprogram; you can't fall into it from another section of code. It can be reached only by a subprogram call.

Another feature—one that is usually found only in more sophisticated compiled languages—is the ability to extend the language with runtime libraries.

Other versions of Microsoft BASIC have built-in commands for graphics and sound production, but this version also has functions and commands for making programs conform to the Macintosh user interface: commands to control windows, menus, dialog boxes, and the mouse.

The Microsoft BASIC interpreter is rich in graphics commands and functions (figure 6.2). A few are graphics commands common to other versions of BASIC, but most use the Macintosh graphics routines in the toolbox. You can use other toolbox routines by calling them as assembler subroutines.

Other features that Microsoft has added to BASIC for the Macintosh are device-independent input/output, the ability to use very large strings (of up to 32,767 characters), and a set of commands for moving sets of pixel values (parts of pictures) from the screen to memory and back again.

Appendix B contains a complete list of Microsoft BASIC commands and reserved words arranged by function.

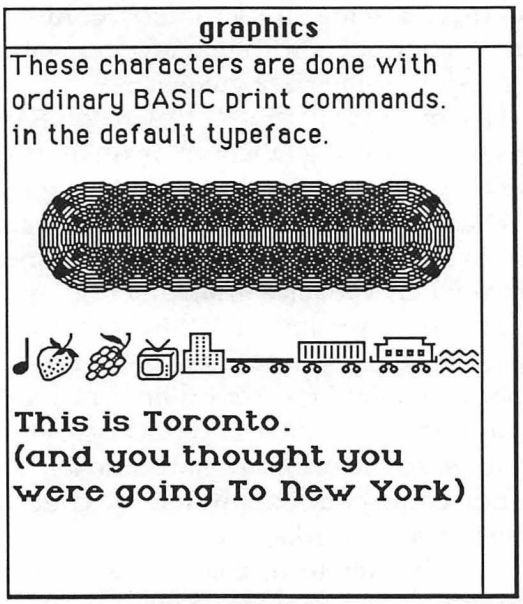


Figure 6.2 Microsoft BASIC graphics

Microsoft BASIC for the Macintosh is actually two different BASIC interpreters: one does calculations using floating-point arithmetic; the other uses an extended decimal representation of numbers for financial calculations. The files produced by the two interpreters are not compatible, but conversion functions can be used to change data from one format to the other.

IN THE DRIVER'S SEAT—CONTROLLING BASIC

Sitting at the controls of Microsoft BASIC, you get a feeling of déjà vu (if you've read the preceding chapter). Microsoft BASIC and Macintosh Pascal are both interpretive languages operating in the same environment, so they have similarities.

In figure 6.3, we open the Microsoft BASIC disk and see two BASIC interpreters in the directory window: the engineering and scientific one (labeled *b* for binary) and the financial one (labeled *d* for decimal).

You can tell which interpreter was used to create a program by checking the upper right-hand corner of the program's icon: π, the

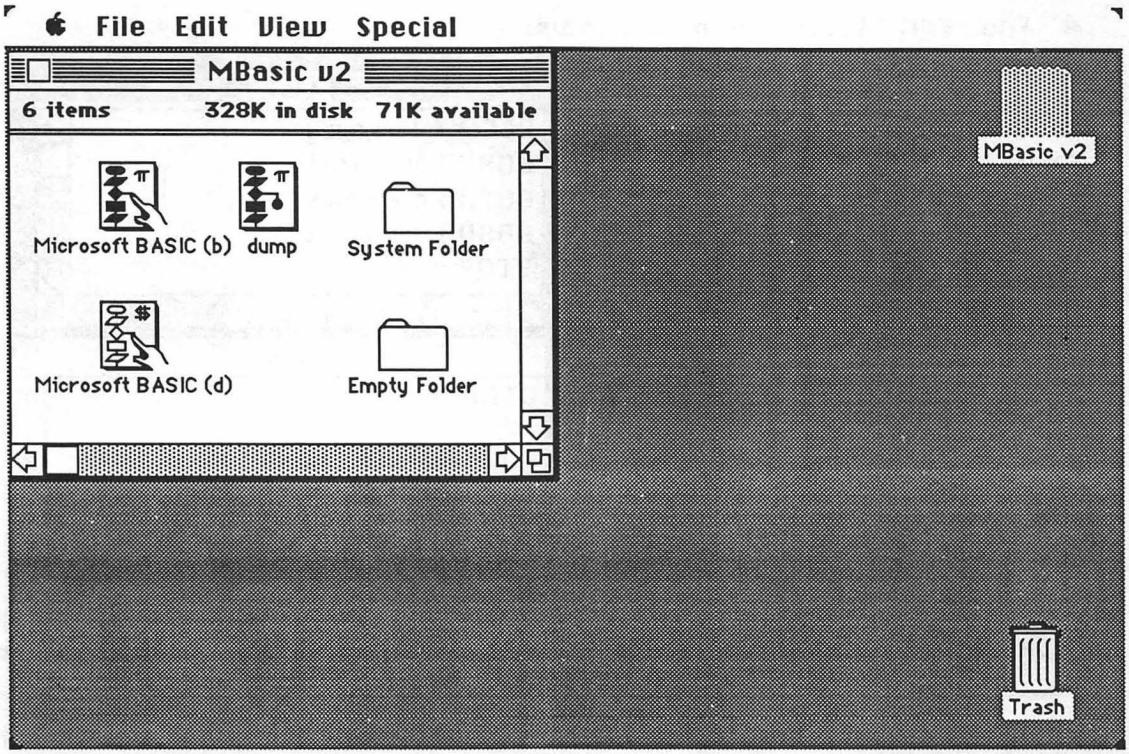


Figure 6.3 Microsoft BASIC disk directory

Greek letter pi, identifies the engineering and scientific version, and a dollar sign identifies the financial version.

If you intend to do most of your work in one version, it would be wise to move the other interpreter to another disk, to avoid using the wrong version. In this book we will use the engineering and scientific interpreter, so we won't be seeing the financial version again.

Take a look at figure 6.4. You can see the windows and the titles of menus used by Microsoft BASIC.

BASIC displays your program in two List windows so that you can examine and edit two areas of a long program at the same time.

The output window is where BASIC displays the output from your program. That output might be text, produced by the traditional BASIC output commands (PRINT, PRINT USING, and WRITE) or graphics produced by BASIC commands that access toolbox graphics routines.

At the bottom of the screen rests BASIC's Command window. Most of the things now done with menus on the Macintosh were done in older versions of Microsoft BASIC by commands entered at the keyboard. You

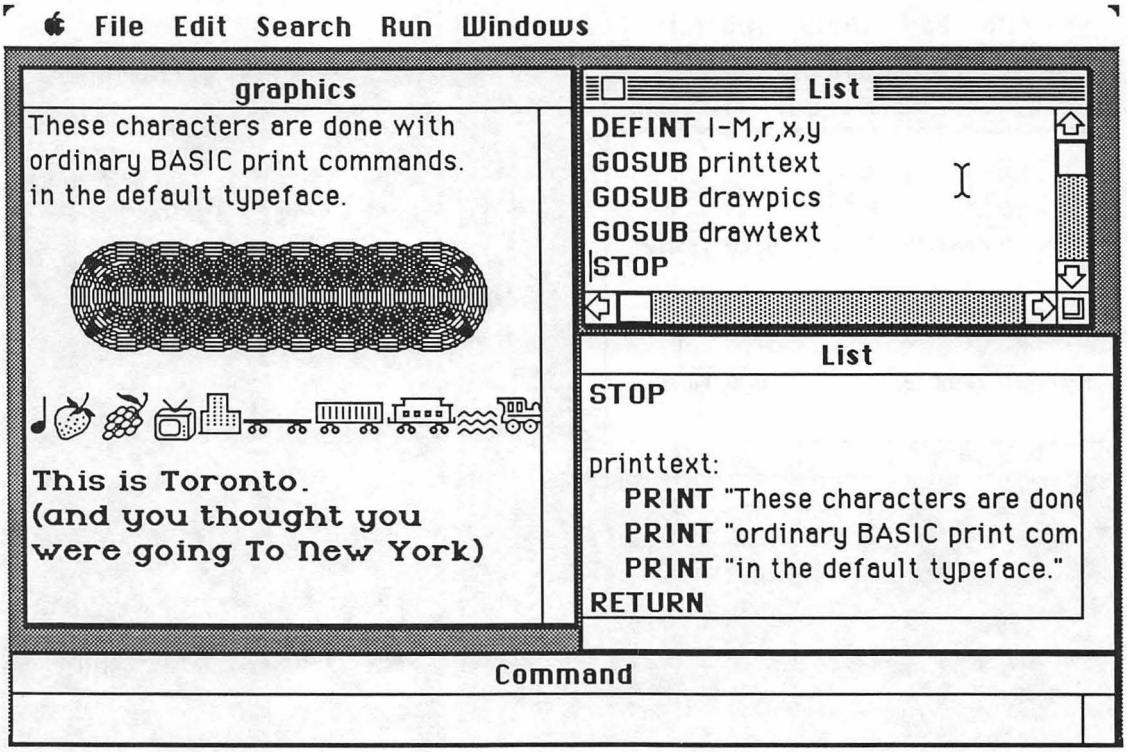


Figure 6.4 Microsoft BASIC menus and windows

can still use these commands by typing them in the Command window. You can also type any single line of BASIC in the Command window; the line is then executed immediately. The Command window also makes a handy calculator or a place to test BASIC statements before you put them in a program.

All the Microsoft BASIC windows are standard Macintosh windows with scroll bars, a size box, and a close box.

The File and Edit menus are the same menus you are accustomed to seeing in other applications, except that there is no Undo item in the Edit menu (figure 6.5). It's possible to get along without Undo, even though it is convenient to have. You can undo a Cut by pasting the text back where it was. Likewise, you can undo a Paste by selecting and cutting out the pasted text.

Also, the Close option in the File menu doesn't operate the way it does in most Macintosh applications. In Microsoft BASIC, it just closes the active window instead of closing the program file.

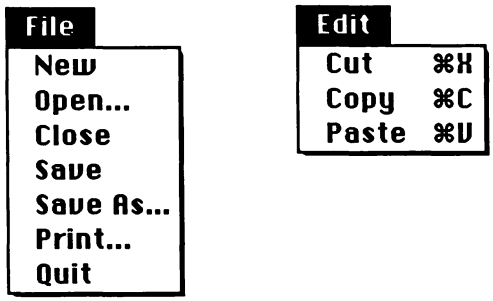


Figure 6.5 File and Edit menus

The Search menu finds and replaces text or finds the cursor (you can lose it in a very long program if you scroll away from it). See figure 6.6.

The Windows menu selectively activates the List, Second List, Command, and output windows. To close a window, first make it the active window and then either click its close box or choose Close from the File menu.

It's the Run menu that really controls the execution of your BASIC program (figure 6.7). Choosing Start does exactly what you would expect: it starts your program. You can stop the program at any time by selecting Stop from the menu. Once you've stopped the program, you can continue from that point by choosing Continue from the Run menu. Like most BASIC interpreters, this one lets you continue even if you have modified the program or changed the value of a variable.

Instead of continuing your program, you can single-step through it by selecting Step. Each time you select Step, the interpreter executes a BASIC statement. Several Step commands are required to execute a line of code that has more than one BASIC statement.

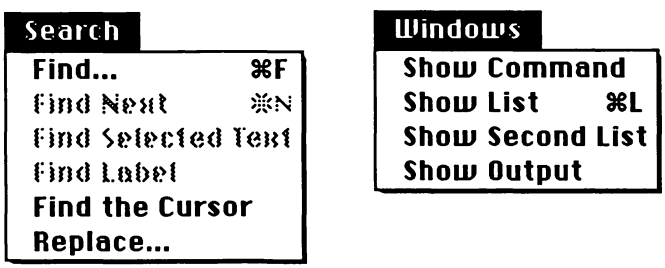


Figure 6.6 Search and Windows menus

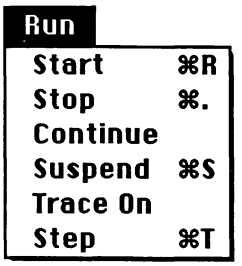


Figure 6.7 Run menu

Selecting the Suspend item suspends the program's execution. The program remains suspended until you press a key (any key) on the keyboard. When a program is stopped, you can examine variables and make changes, but when it is suspended, you cannot make any changes or examine variables.

Actually, it is easier and faster to suspend the program by using the command key: depress the command key and type S. You can use Suspend to pause and think about what the program is doing, stop and look at some output before it scrolls out of the output window, or take some action, like setting up the printer. For short pauses, you can just click the Run menu with the mouse. The program pauses as long as the menu is down.

Trace On causes the interpreter to trace program execution in the List window. It draws a box around each program line while that line is executing. When you select Trace On, the trace item in the Run menu changes to Trace Off. Selecting Trace On again will, of course, turn off tracing. Trace On and Trace Off operate in exactly the same way as the BASIC commands TRON and TROFF.

Controlling Microsoft's BASIC for the Macintosh is certainly easier than dealing with other BASIC interpreters. Of course, if you want to continue using the old BASIC commands to control the interpreter, feel free. They're still there. You just type them in the Command window.

WRITING AND DEBUGGING A PROGRAM

To see how Microsoft BASIC and the Macintosh user interface can be used to write a BASIC program, let's go through an example. We'll use the program that produced the window full of graphics in figure 6.2.

We start by putting the BASIC disk in the machine and opening the icon representing the scientific and engineering version of the BASIC interpreter (the icon with a π). When the interpreter executes, it opens the output window, the Command window, and one List window (see figure 6.8).

The large window labeled *Untitled* is the output window. The output window is always labeled with the program's name; since we haven't named it yet, it's untitled at this point.

Our program is very simple. It's going to print some text in the default typeface, draw some pictures, and print some text in other typefaces. We start by declaring a range of variable names (all those beginning with *j*, *k*, *l*, *m*, *n*, *x*, or *y*) to be integers. Next, we call subroutines to do each of the three functions we want. Then we end the program.

Figure 6.9 shows our program in the List window. Note that we have rearranged the windows in a more convenient layout. The output window is behind the List window, which we don't need for now.

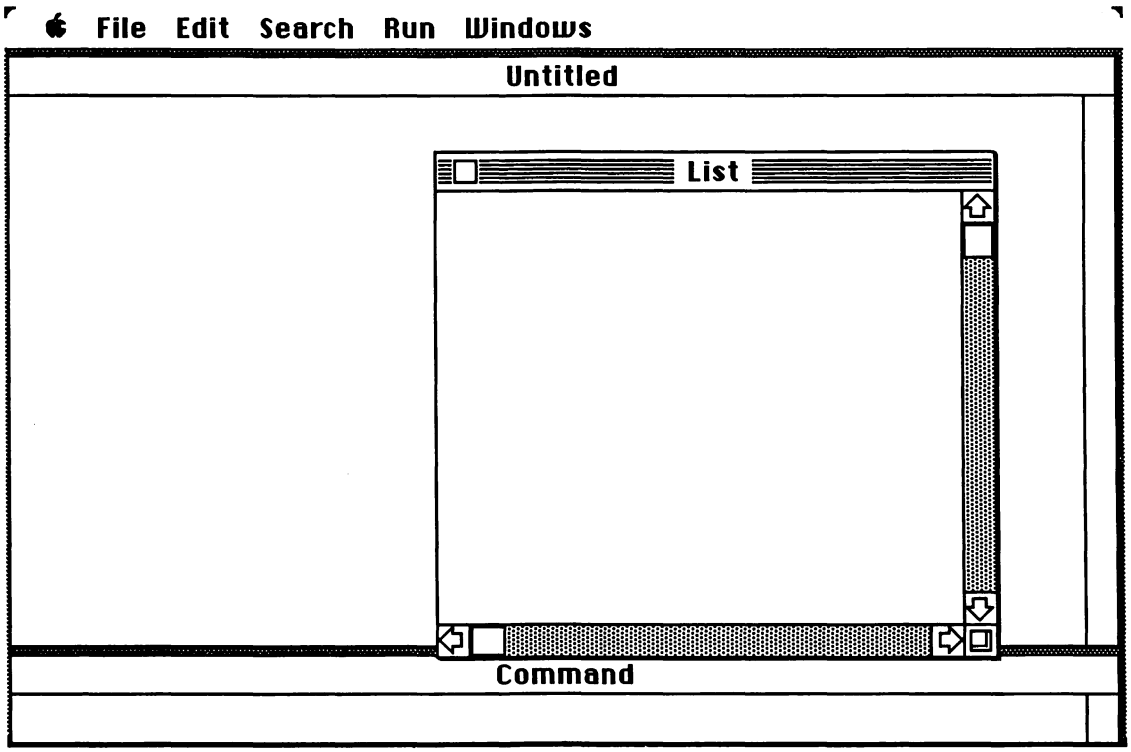


Figure 6.8 BASIC windows

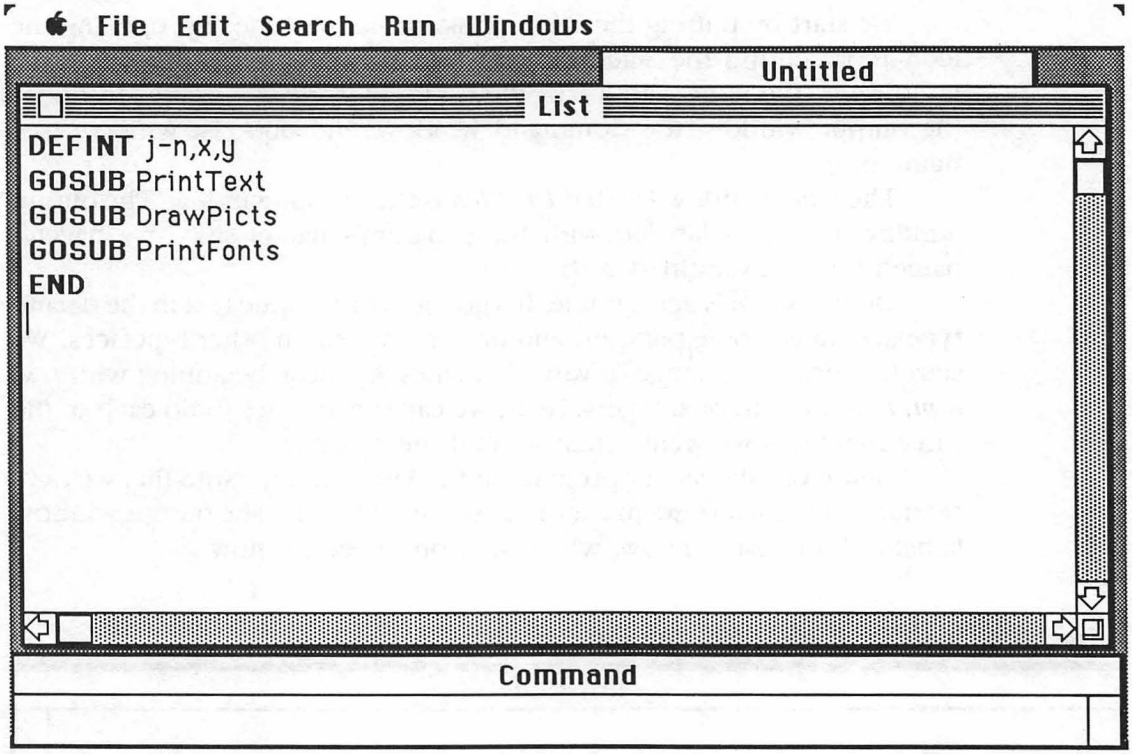


Figure 6.9 Start of the graphics program

We've ended the program with an END statement. We could have used the STOP statement, but that is more appropriate for debugging. END closes all of your files and does not leave the program in a state from which it can be restarted. STOP keeps the files open and the variables allocated. It leaves the program in a state from which it can be continued (via the Continue item in the Run menu or the CONTINUE command typed in the Command window).

The PrintText subroutine consists of three print statements. In figure 6.10, we've just started typing the PrintText routine. The routine's name serves as a label. We've followed that with a PRINT statement on the next line. We wanted to indent the lines composing the executable code in the subroutine. Since we indented this line, when we typed a carriage return to end the line, the cursor did not return to the left margin. Instead, it stopped at the tab point of our indentation so that we wouldn't have to manually indent each succeeding line of code.

Something else happened when we hit return: the PRINT command was changed to boldface. When we move the cursor out of a line of

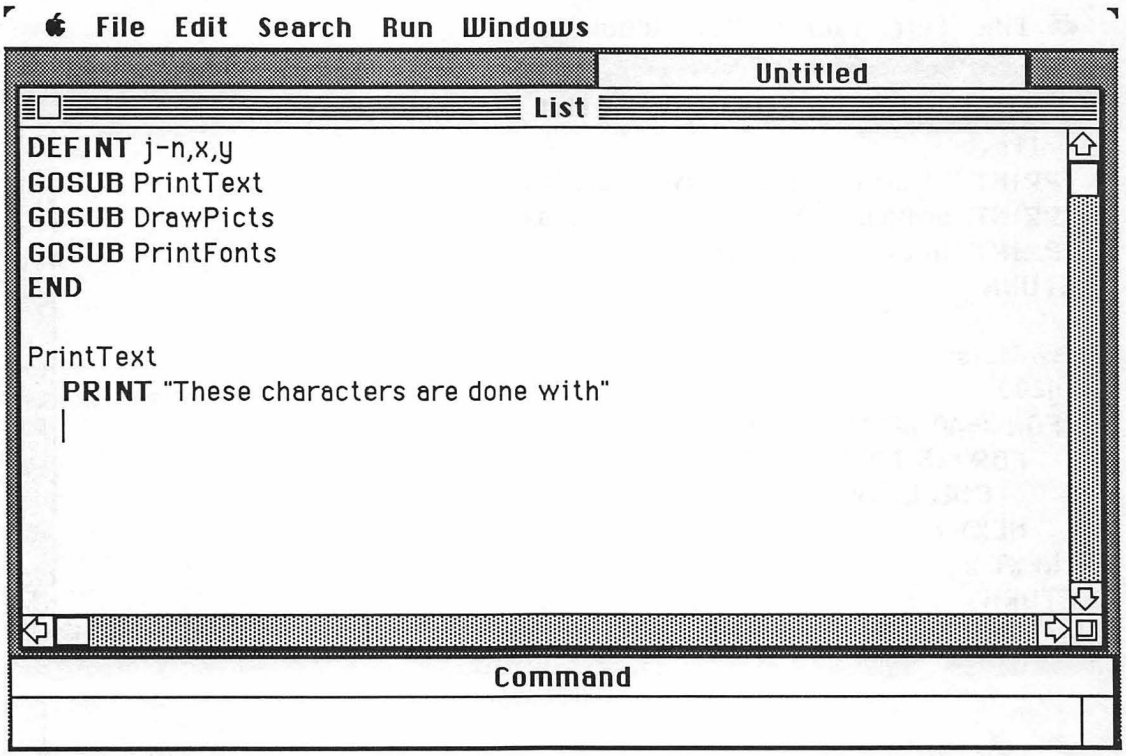


Figure 6.10 Entering the PrintText routine

code whether by typing a carriage return or by moving the cursor with the mouse, the interpreter puts all of the BASIC reserved words in boldface.

When we finish typing the PRINT statements, we want to put the RETURN statement at the left-hand margin, so we need to backspace the cursor to the margin after typing carriage return for the last PRINT statement (see figure 6.11).

The DrawPicts routine is slightly more involved. It uses two FOR-NEXT loops to step two variables through a range of parameters for drawing a circle; then it draws the circle by using the CIRCLE statement.

In figure 6.12, we see the PrintFonts subroutine. The actual printing is done by standard BASIC PRINT statements, but we call some built-in subroutines to set the typeface first. A number supplied as a parameter to the TextFont subroutine selects the font that will be used by PRINT.

In order to use a font, you must have it installed in your system file. To save disk space, the Microsoft BASIC disk comes with a minimum of installed fonts. If you want to print with any fonts other than the default

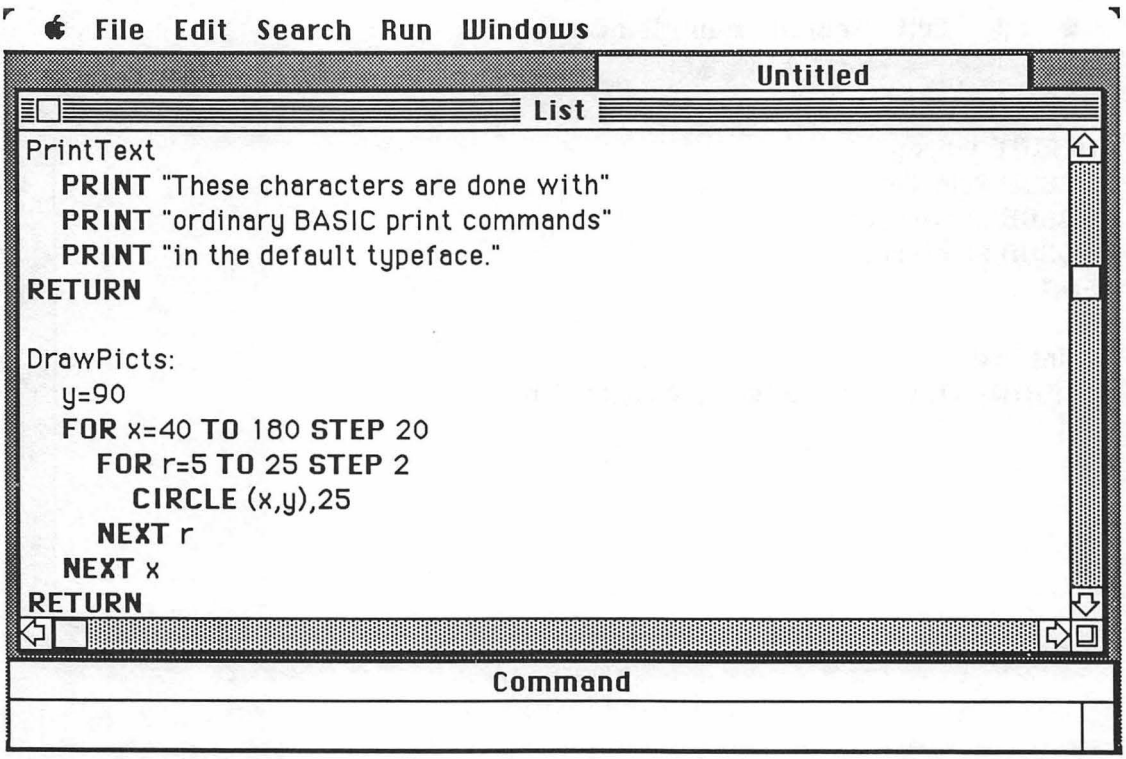


Figure 6.11 PrintText and DrawPicts

(Geneva), you need to get the font mover program from your system disk and use it to add fonts to the system file on your BASIC disk. First copy the font mover program and the Fonts file to the BASIC disk; then run the font mover and add the fonts you want. Be conservative in adding fonts. They take up a lot of disk space. The TextFont font selection numbers are listed below:

Number	*Font*
0	System font (Chicago)
1	Application font (Geneva)
2	New York
3	Geneva
4	Monaco
5	Venice
6	London

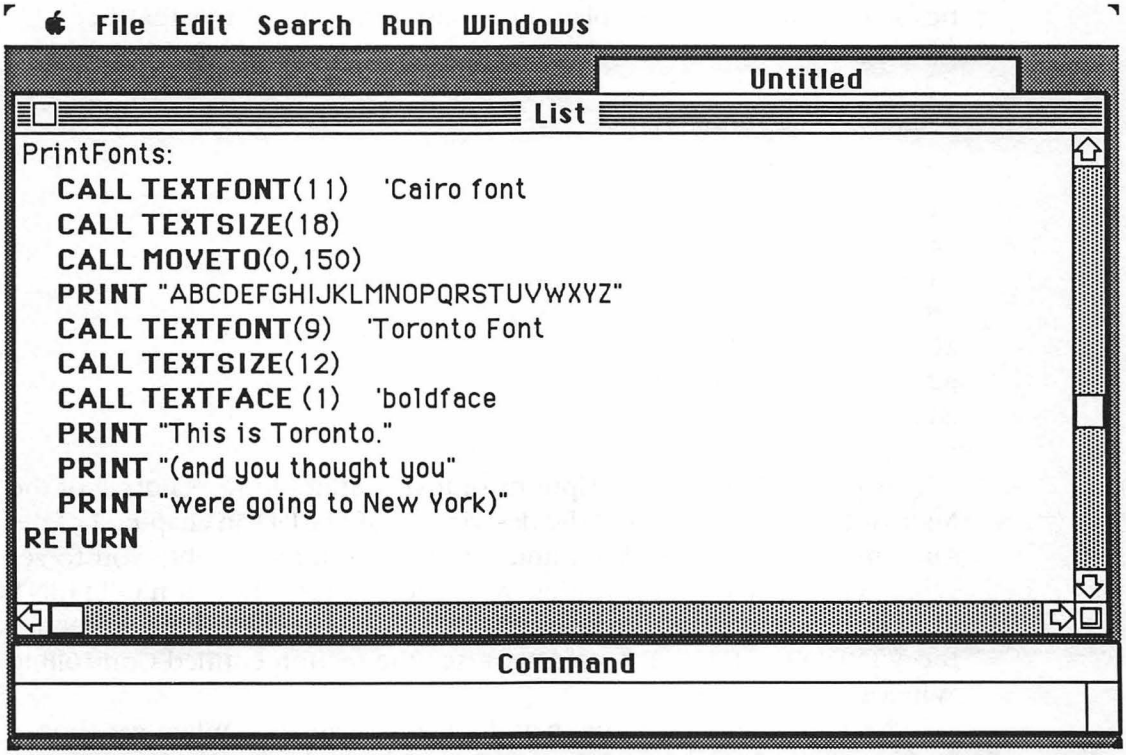


Figure 6.12 PrintFonts subroutine

Number	*Font*
7	Athens
8	San Francisco
9	Toronto
10	Seattle
11	Cairo

The TextSize subprogram sets the size of type that PRINT uses. We have used 18-point Cairo and 12-point Toronto fonts.

Before printing in the Toronto font, we also called TextFace to select boldface type. TextFace can be used to select any combination of seven typeface characteristics. To generate a TextFace parameter, you choose the characteristics you want from the TextFace list and add their numbers together. For example, to select boldface italic type, you use the command CALL TEXTFACE(3). A parameter of zero clears the TextFace characteris-

tics selection (it reverts to plain text). Listed below are the TextFace typeface characteristics:

Number	*Characteristic*
0	Plain text
1	Boldface
2	Italic
4	Underlined
8	Outlined
16	Shadow
32	Condensed
64	Extended

If you've read the descriptions of text routines in Appendix F of the Microsoft BASIC manual and the description of LPRINT in chapter 7 of the same manual, you probably think that these routines enable you to set typeface parameters for printing with the printer. They don't. LPRINT does not print with the Macintosh fonts. To print with fonts, you must use the WINDOW OUTPUT # statement (see the section entitled Controlling Windows).

Now let's run our program and see what happens. When we choose Start from the Run menu, figure 6.13 is what we get.

When the program runs, the output window overlays the List window briefly; then the interpreter stops, and the output window goes back behind the List window. The interpreter stops running the program because it's found an error, an undefined label. It draws a box around the program statement with the error.

The program tried to GOSUB to PrintText, but when we had typed in the PrintText subroutine, we forgot the colon after the subroutine name. This version of BASIC requires a colon after line labels.

Before we can correct the mistake, we need to get rid of the box with the error message by clicking the OK button in the box. When we add the colon and try again, we get another error message (figure 6.14).

This time we got much further. We actually put some text and pictures in the output window, but the program stopped when the BASIC interpreter detected a syntax error. Note the difference here between BASIC and Pascal. Pascal does some of its syntax checking when you type your program statements; BASIC does syntax checking only when you execute them.

Our error this time was to call the PRINT statement. PRINT should not be invoked with a CALL. It's not a subprogram; it's a BASIC command. To fix the program, we select the CALL statement with the mouse and use

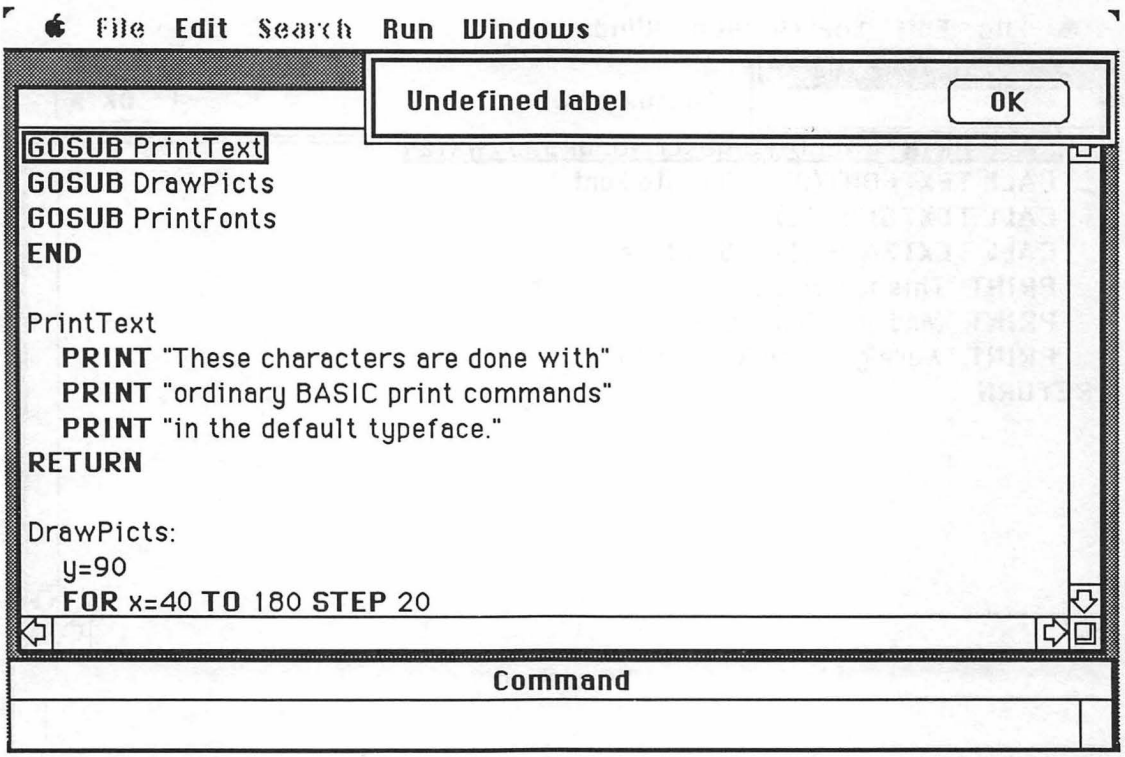


Figure 6.13 Undefined label

Cut from the Edit menu to delete it (leaving the PRINT statement). The Edit menu works in the List window just as it does everywhere else in the Macintosh.

This time our program runs to completion, draws pictures in the window, and outputs some text in the Toronto font. Since the output window went back under the List window when the program finished, now is probably the time to rearrange the windows so we can see our output. We move the windows, change their sizes, and run the program again.

This time, we got it right. Those interesting little figures below the concentric circles in figure 6.15 are actually the Cairo type font. The Cairo font is used to create little pictures instead of characters.

If our program were more complex, we might have needed to trace its execution or use some STOP statements in the program for breakpoints. To see how tracing works, we can select Trace On from the Run menu and then select Start from the Run menu. The program executes as

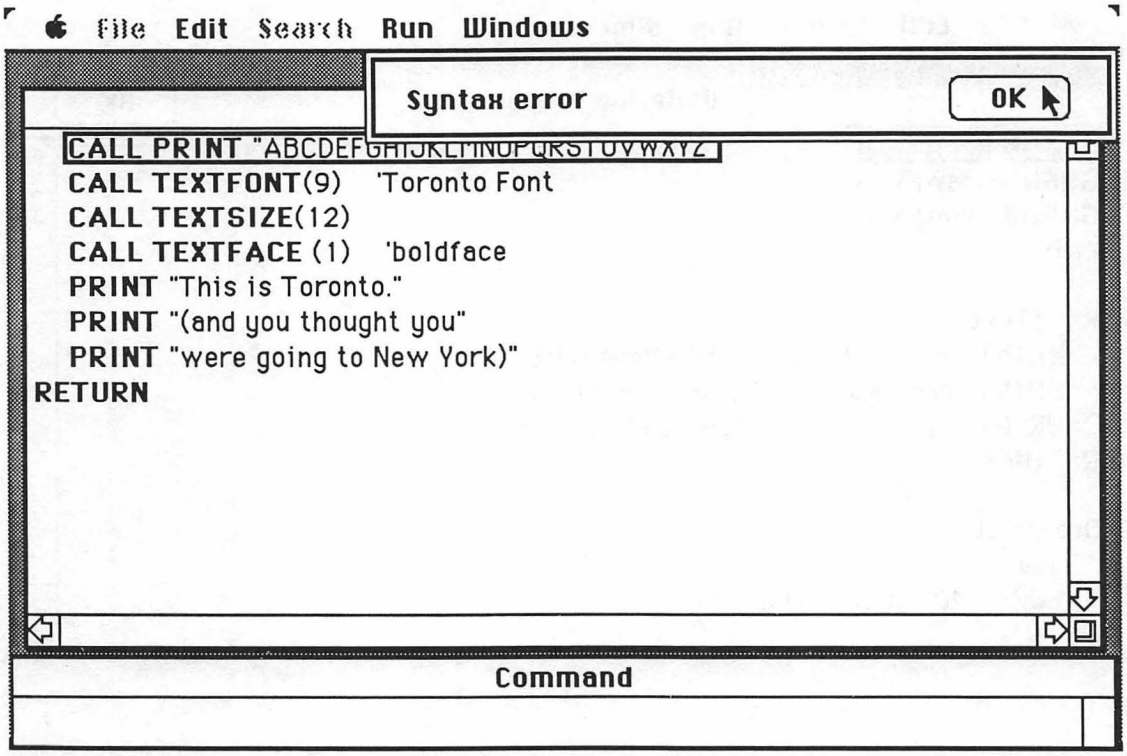


Figure 6.14 Syntax error

before but with a lot more activity in the List window. The interpreter draws a box around each line of code as it executes.

Figure 6.16 captures the screen as the program is being traced—just as the "NEXT r" statement is executing.

The Trace On option in the Run menu turns on tracing for the entire program. If we know that a problem is isolated in one subroutine, we might want to trace just that subroutine's execution. To do that, we insert TRON and TROFF statements in the code. Let's trace just the PrintFonts subroutine.

The first thing we need to do is select Trace Off from the Run menu so that the rest of the program isn't traced. We then put a TRON statement at the beginning of the subroutine to start the trace, and a TROFF statement at the end to stop tracing. Figure 6.17 shows the altered PrintFonts routine.

Figure 6.18 (page 92) catches the trace in progress as the program executes. You can see the TRON statement at the start of the subroutine and the CALL MOVETO statement as it is executing.



Figure 6.15 Graphics program output

The beauty of tracing on the Macintosh is that your trace results show up in the List window, not intermixed with your output. (Also, it's fun to watch the interpreter zipping through your program.) With ordinary BASIC interpreters, tracing prints the line number of each line of code as it executes. Thus you end up with a barrage of line numbers interspersed with your output.

If, as an exercise, you are actually running this program on your Macintosh, run with Trace On, and then pause occasionally by selecting the Run menu. This is a quick and easy way to pause for a long enough interval to see where you are and what's in the output window.

When you are debugging a large program, there are times when you want to set breakpoints—that is, have the program stop at particular places so you can examine variables and then continue execution. Sometimes you may want to set a breakpoint at a particular line in the program just to see if it gets executed. In Microsoft BASIC, you set breakpoints with the STOP statement.

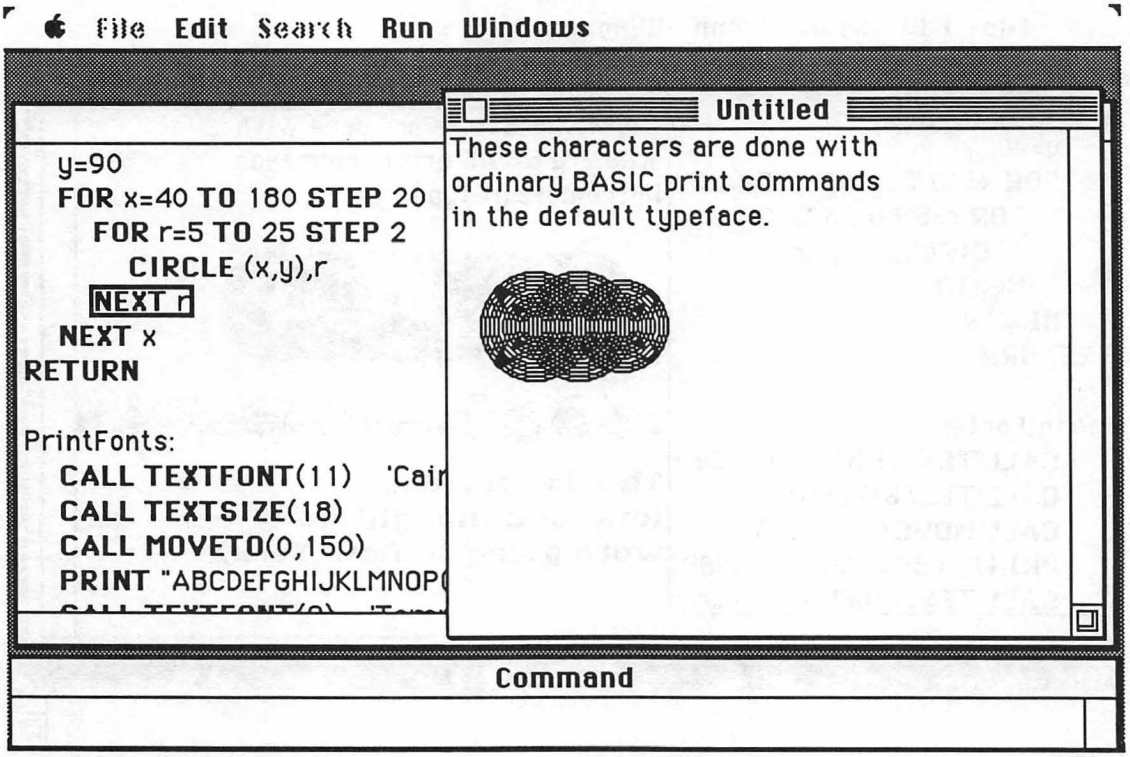


Figure 6.16 Tracing program execution

When the interpreter encounters a STOP statement in the program, it stops, leaving all files open and all variables allocated. While the program is stopped, you may examine variables, change variables, or change program statements. Let's try changing a program statement by putting a STOP in the DrawPicts routine between the "NEXT r" statement and the "NEXT x" statement. When we execute the program and it stops, the display looks like figure 6.19.

Note that we have typed a program statement in the Command window to change the value of y. Previously, the program positioned all of the circles at $y = 90$, but we have set y to 100. When we choose Continue from the Run menu, the program executes until it again reaches the STOP statement. We see in figure 6.20 (page 94) that the program has drawn another series of circles, all lower in the window, at $y = 100$. If we leave the STOP statement where it is, the program stops after drawing each set of concentric circles.

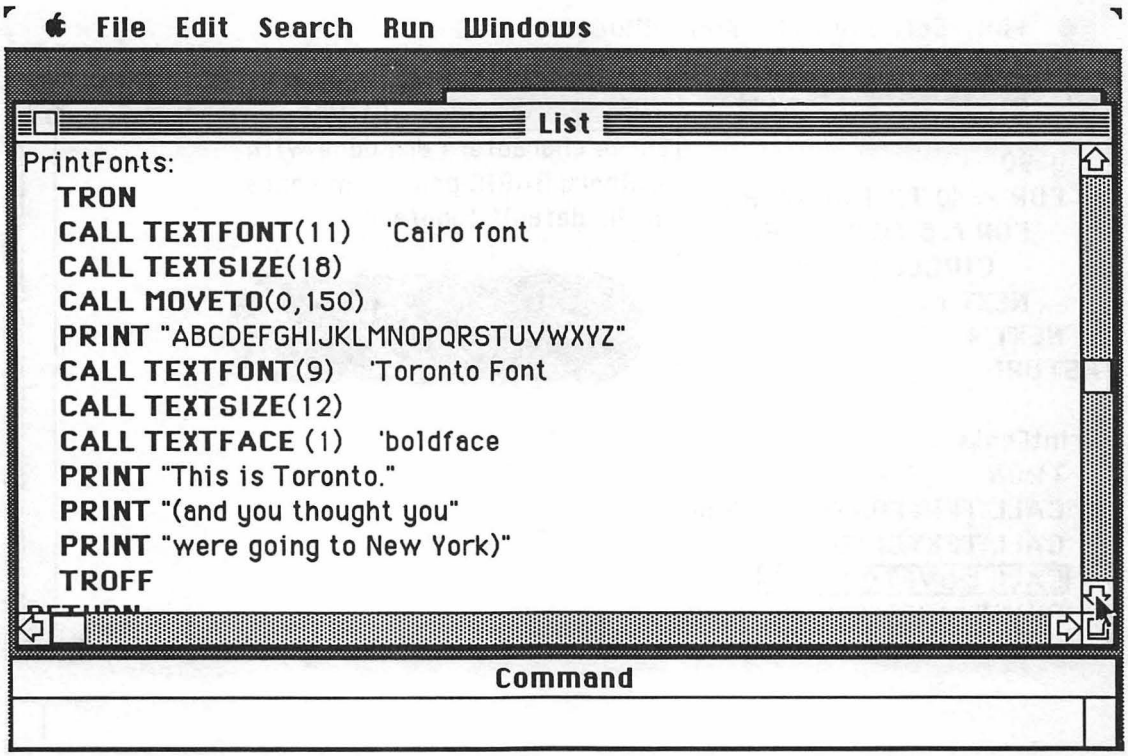


Figure 6.17 TRON and TROFF

We want to get a listing of our program, but first we remove the STOP, TRON, and TROFF statements. To get the listing (see listing 6.1, page 95), we choose Print from the File menu.

Now we want to save our program and get out of the interpreter, so we choose Close from the File menu. But the only thing that happens is that the List window disappears. Close works a little differently in Microsoft BASIC than in some other Macintosh programs; it closes the active window, not the program file.

To save our program, we choose Save or Quit from the File menu. Before the Macintosh can save our program, it needs a name for it, so the dialog box in figure 6.21 (page 96) asks us for a name. It also lets us choose the format in which to save the program: Text, Compressed, or Protected. If we choose *Text*, the program is saved in text form, suitable for use with a word processing program. *Compressed* is the format normally used by BASIC; it retains all of the program information in a compressed form that takes less disk space than the Text format. *Protected* stores the

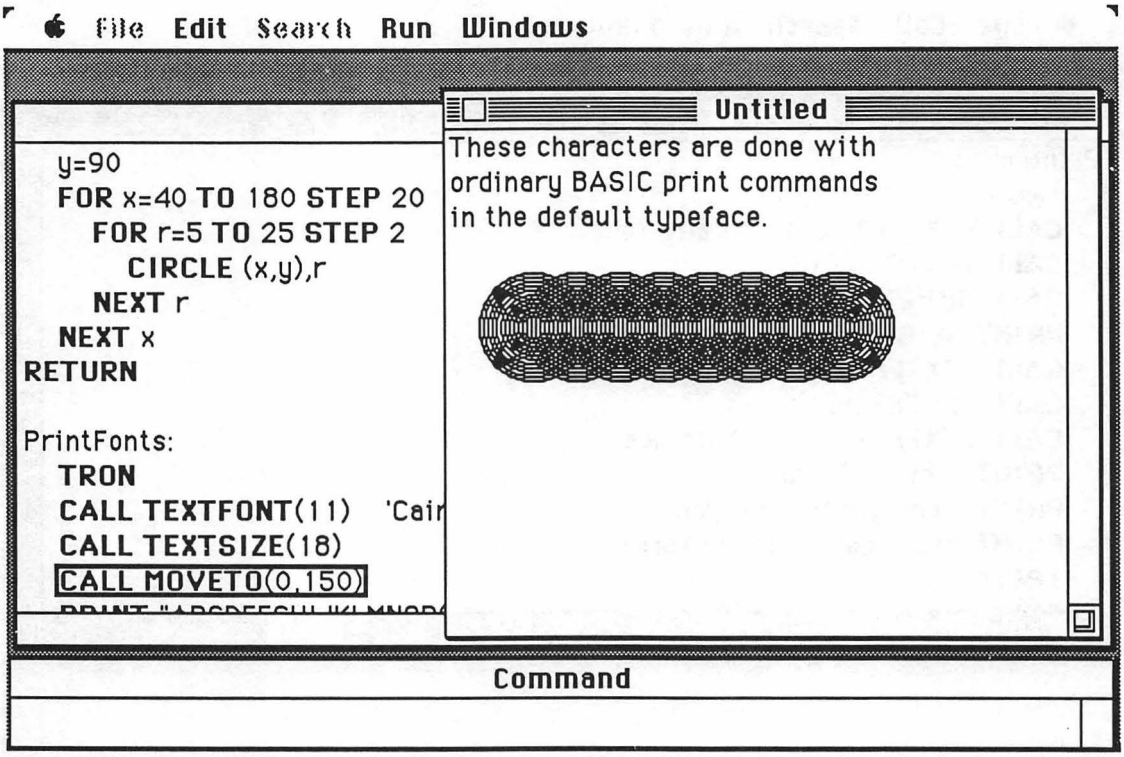


Figure 6.18 Tracing with TRON and TROFF

program in a format that the interpreter understands and can execute but that people cannot read; a program in the Protected format cannot be listed.

GRAPHICS AND FONTS ON THE PRINTER

We are going to add one more routine to our program—a routine that sends graphics and text in different type fonts to the printer. Text in type fonts is treated as graphics, so this discussion is really about how to print graphics on the printer.

As we have seen, in BASIC the PRINT and graphics statements normally send their output to a window on the Macintosh display. When we send output to the printer, we create it in the same way. We use the same BASIC graphics and PRINT statements as if they were going to display their output in a window. We even define a window for them, a window

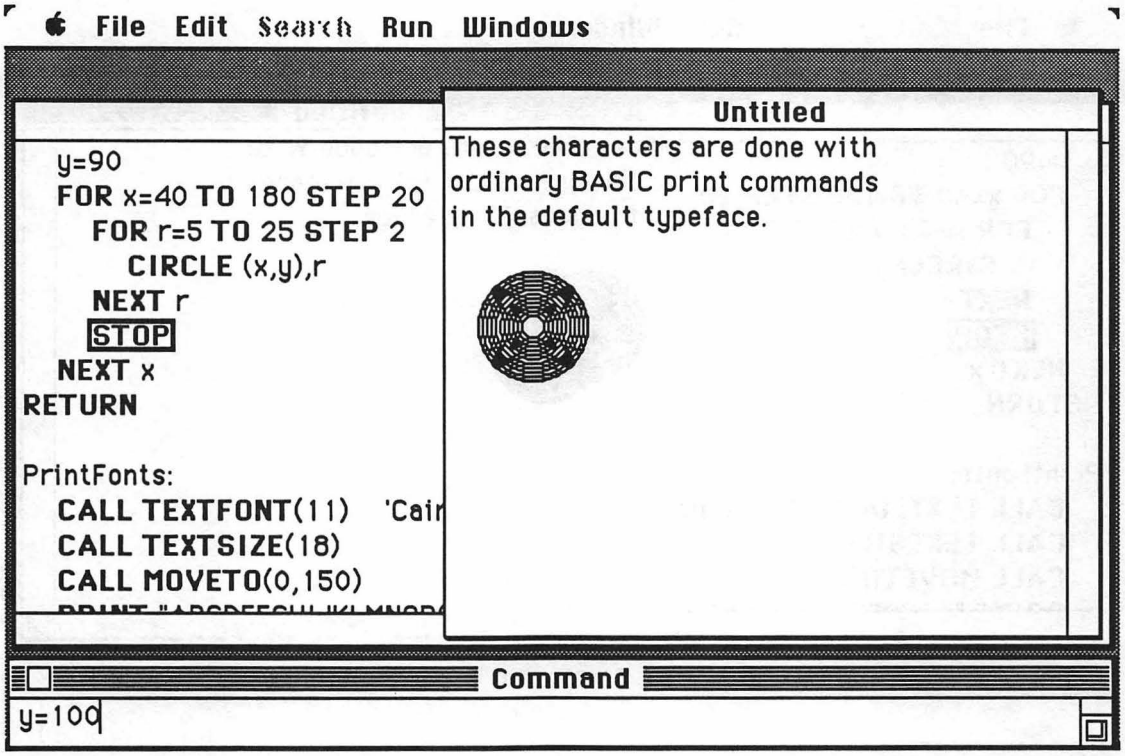


Figure 6.19 Stopped program

that is not on the screen. This "window" actually stores the output in a file; when the file is closed, the system prints its contents on the printer.

Let's take a closer look at how this system works. We'll use our graphics program and add a routine called ListFonts to print some text (in type fonts) and a circle. First we add a GOSUB to the ListFonts subroutine at the beginning of our program (see listing 6.2, page 96).

Then we add the new subroutine to the end of the program (see listing 6.3, page 97).

To print graphics on the printer, we must first open the printer for output. We open it in much the same way we open a file, but we specify "LPT1:" instead of a file name. BASIC refers to files by number, not by name. The file name is specified in the OPEN statement, which relates the name to a file number. Other BASIC statements then reference the file by its assigned number.

Now that we have a window open for output to the printer, any graphics statement we execute sends its output to a print file. When we close the print file, the system prints it.

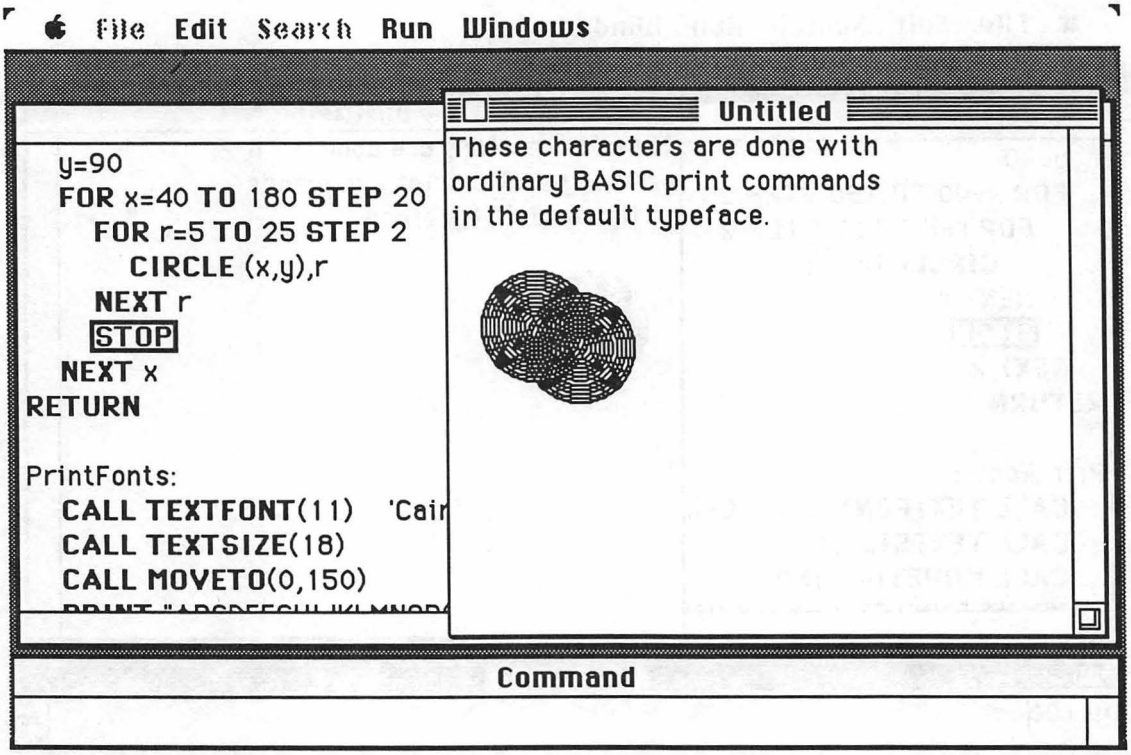


Figure 6.20 Program stopped again

In our subroutine, we have used the TextSize, TextFont, and TextFace toolbox routines to set up several different type styles for printing. Note that we can print with these fonts only if the fonts are in the system file on our BASIC disk.

The last statement before CLOSE draws a circle. The CLOSE statement closes the print file and causes it to print (figure 6.22, page 98).

THE PICTURE STATEMENT

Microsoft BASIC has a provision for storing a picture so that you can display it later. You may use this provision to create a picture and later display it at several locations in the window. You might also use it to store a picture you are forming in a window—to refresh the window after it has been overlaid by another window.

Listing 6.1 Graphics

```
REM graphics program

DEFINT j-n,x,y
GOSUB PrintText
GOSUB DrawPicts
GOSUB PrintFonts
END

PrintText:
   PRINT "These characters are done with"
   PRINT "ordinary BASIC print commands"
   PRINT "in the default typeface."
RETURN

DrawPicts:
   y=90
   FOR x=40 TO 180 STEP 20
      FOR r=5 TO 25 STEP 2
         CIRCLE (x,y),r
      NEXT r
   NEXT x
RETURN

PrintFonts:
   CALL TEXTFONT(11)   'Cairo font
   CALL TEXTSIZE(18)
   CALL MOVETO(0,150)
   PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
   CALL TEXTFONT(9)   'Toronto Font
   CALL TEXTSIZE(12)
   CALL TEXTFACE (1)   'boldface
   PRINT "This is Toronto."
   PRINT "(and you thought you"
   PRINT "were going to New York)"
RETURN
```

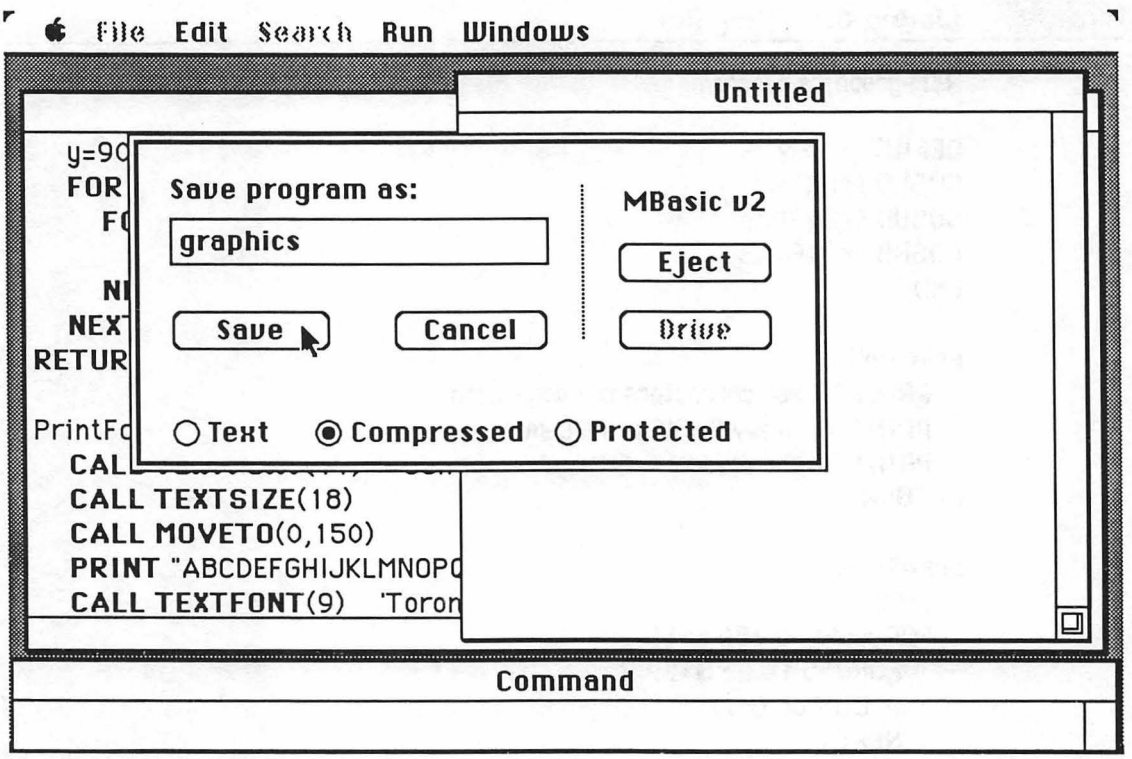


Figure 6.21 BASIC Save dialog box

Listing 6.2 Graphics Main Loop with ListFonts

```
DEFINT j-n,x,y
GOSUB PrintText
GOSUB DrawPicts
GOSUB PrintFonts
GOSUB ListFonts
END
```

In listing 6.4, we have modified the DrawPicts routine to first store our circles picture and then draw it at several locations in the window. This method is slightly faster than using the circle commands to draw each individual copy.

The PICTURE ON statement starts the recording of our picture. The FOR-NEXT loop with the CIRCLE statement creates the picture, and the

Listing 6.3 ListFonts Subroutine

```
ListFonts:  'an example of printing on the printer with type fonts
  OPEN "LPT1:" FOR OUTPUT  AS #5
  WINDOW OUTPUT #5       'send window output to the printer
  CALL TEXTSIZE(14)
  CALL TEXTFONT(5)
  PRINT
  PRINT "This is Venice"
  PRINT "Its only a short trip from Toronto"
  PRINT
  CALL TEXTSIZE (12)
  CALL TEXTFONT(2)
  CALL TEXTFACE(1)
  PRINT "and this is old, bold, New York."
  PRINT
  CALL TEXTSIZE (10)
  CALL TEXTFACE(0)
  CALL TEXTFONT(2)
  PRINT "but a little New York goes a long way."
  CALL TEXTSIZE(18)
  CALL TEXTFONT(6)
  PRINT
  PRINT "London Looks Good, AS CAPITALS GO."
  CALL TEXTFONT(11)
  PRINT
  PRINT "HGGFGJ"
  PRINT
  PRINT "MNMNMNMNMNMNMNMNMNMNMN"
  CIRCLE (200,340),50
  CLOSE #5
RETURN
```

PICTURE OFF statement stops recording it. The FOR-NEXT loop with the PICTURE(x,y) statement draws several copies of the picture. The (x,y) parameters specify where BASIC will draw the picture; they are pixel coordinates relative to the upper left-hand corner of the output window. When we now run the program, the output in the window looks just as it did before we modified the program.

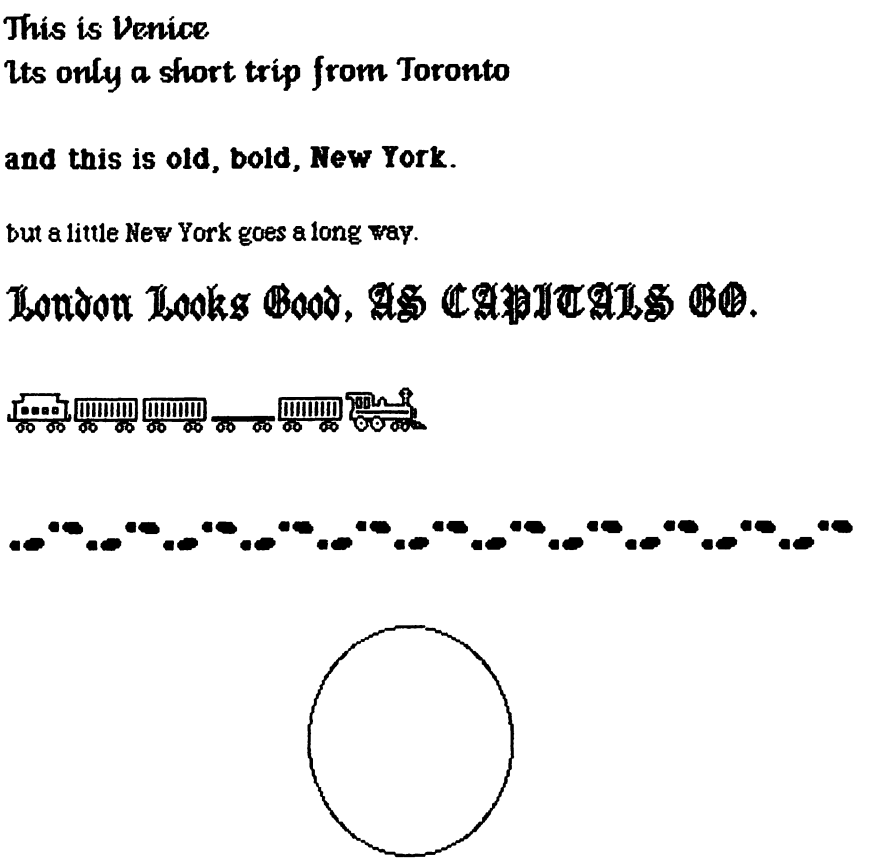


Figure 6.22 Printer output

MICROSOFT BASIC AND THE TOOLBOX

Access to the toolbox is vital to programmers writing for the Macintosh. Such access is required in order to create programs within the Macintosh user interface. Various compilers and interpreters have access to different sets of toolbox routines. Some, like assembler, can access any toolbox routine directly. Others, like the BASIC interpreter, access a commonly used subset of toolbox routines. The Pascal and two BASIC interpreters have built-in access to some toolbox functions; they can call others with an assembly-language calling procedure.

There are three ways to access toolbox functions in Microsoft BASIC. The first way is to use any of a number of built-in BASIC statements for

Listing 6.4 DrawPicts Subroutine with PICTURE

```
DrawPicts:
  PICTURE ON
  FOR r=5 TO 25 STEP 2
    CIRCLE (25,25),r
  NEXT r
  PICTURE OFF
  y=65
  FOR x=15 TO 155 STEP 20
    PICTURE(x,y)
  NEXT x
RETURN
```

creating and controlling menus, windows, and dialog boxes and for controlling the mouse. These built-in commands are easier to use than direct calls to toolbox routines because they generally perform functions that would require many calls to toolbox routines. The built-in BASIC toolbox statements include the following:

Dialog Statements

BUTTON DIALOG

Window Statements

CLS EDIT FIELD EDIT$ SCROLL WINDOW

Menu Statements

MENU ON MENU

Mouse Statements

MOUSE ON MOUSE

The second method to access toolbox routines is to use a set of toolbox-routine names that are built into BASIC. BASIC recognizes the names of these toolbox routines and knows their addresses, so the interpreter can generate assembler language calls. These routines are accessed with BASIC's CALL statement. For example, CALL TEXTSIZE(12) would call the TextSize routine in the toolbox and pass the number 12 to it as a parameter. The toolbox routines recognized by BASIC include the following:

Typeface Control

TEXTFONT TEXTSIZE TEXTFACE TEXTMODE

Drawing Pen Control

GETPEN MOVETO MOVE PENSIZE PENPAT
PENMODE PENNORMAL HIDEPEN SHOWPEN

Background Pattern

BACKPAT

Drawing

FRAMEOVAL	FRAMEARC	FRAMEPOLY	FRAMERECT	FRAMEROUNDRECT
PAINTOVAL	PAINTARC	PAINTPOLY	PAINTRECT	PAINTROUNDRECT
ERASEOVAL	ERASEARC	ERASEPOLY	ERASERECT	ERASEROUNDRECT
INVERTOVAL	INVERTARC	INVERTPOLY	INVERTRECT	INVERTROUNDRECT
FILLOVAL	FILLARC	FILLPOLY	FILLRECT	FILLROUNDRECT
LINE	LINETO			

Mouse Cursor

INITCURSOR HIDECURSOR SHOWCURSOR SETCURSOR

The third way to access toolbox routines is to call them as assembler language subroutines. This method is not an easy process and is not recommended for amateurs. Some toolbox routines are designed as functions and not subroutines, so there may be problems passing and returning parameters.

If you really need to call toolbox routines not defined in BASIC, you should probably be programming in another language. If you still require an interpretive language for developing your program, Macintosh Pascal might be a better choice. It has a method of calling toolbox functions and procedures. The parameter-passing conventions and the data structures are more likely to be compatible because the toolbox routines conform to Pascal calling specifications (see chapter 5, Macintosh Pascal).

CREATING AND CONTROLLING MENUS

If you want to write BASIC programs that fit the Macintosh user-interface model, you have to be able to create and use menus. Microsoft BASIC provides a set of functions for that purpose:

Menu Statement	*Function*
MENU	Create or control a menu
MENU RESET	Restore the BASIC interpreter's menus
ON MENU GOSUB	Specify a subroutine to execute when a menu item is selected
MENU ON	Turn on menu-event trapping, which causes execution of the subroutine specified in ON MENU GOSUB, when a menu item is selected
MENU OFF	Turn off menu-event trapping and discard any menu events that occur while trapping is off
MENU STOP	Turn off menu-event trapping but save events until the next MENU ON
MENU(0)	Return the number of the last menu selected
MENU(1)	Return the number of the last menu item selected

There are two approaches to writing a program that uses menus. One, called *polling the menus*, simply uses the MENU(0) and MENU(1) statements to periodically check the state of the menus. The other approach uses the ON MENU GOSUB to execute a routine when a menu item is selected. This method interrupts the program when the user selects a menu item. The BASIC statement that is executing when the menu item is selected completes executing. The interpreter then goes to the first line of the subroutine specified by ON MENU GOSUB. When that subroutine is finished, control is returned to the BASIC program statement following the one that was interrupted (and completed).

The MENU statement creates, enables, and disables menus and menu items. It can also be used to place a check mark beside a menu item. Each menu or menu item is created by a separate MENU statement. The MENU statement requires parameters to specify the menu number, menu item number, state of the item or menu, and menu or item title (an ASCII string that appears in the menu). See figure 6.23.

A menu exists in one of two states, disabled or enabled. If a menu is disabled, its title appears dimmed in the menu bar at the top of the screen, and clicking the title with the mouse does not cause the menu to be pulled down. When a menu is enabled, its title appears in the menu bar (undimmed), and clicking the title causes the rest of the menu to appear.

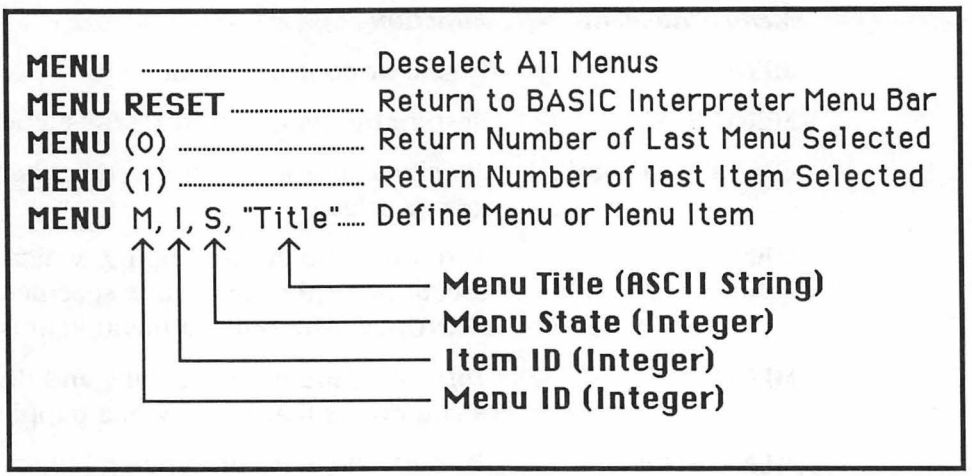


Figure 6.23 MENU statement

A menu item can be disabled, enabled, or checked. Like a disabled menu, a disabled item appears dimmed and may not be selected. An enabled menu item is not dimmed and may be selected. A checked menu item appears undimmed with a check mark beside it. Only an enabled menu item may be checked.

Let's look at the series of MENU statements (listing 6.5) that create the Window1 menu (figure 6.24) in the Memory Dump program, which is listed at the end of this chapter.

To specify a menu, you use a series of MENU statements. The first statement defines the menu, and the others define items in the menu. The first MENU statement has an item number of 0, identifying it as the statement that defines the menu number, menu state, and menu title. The remaining MENU statements each define an item in the menu. They list the menu number, item number, item state, and item title.

Listing 6.5 MENU Statements from Memory Dump Program

```
MENU 2,0,1,"Window1"
MENU 2,1,0,"Repeat"
MENU 2,2,0,"Advance"
MENU 2,3,1,"Octal"
MENU 2,4,2,"Hex"
type1 = 2
MENU 2,5,1,"Ascii"
```



Figure 6.24 Window1 menu

Note that the Hex item has a state parameter of 2 and a check mark in the Window1 menu. One of the functions of this menu is to select a hexadecimal, octal, or ASCII format for the dump. The current format selection is identified in the menu by the check mark.

When the user selects a menu with the mouse, the menu remains pulled down, selected, and displayed in reverse video until the program executes a MENU statement without any parameters. No other menu or item may be selected until the program executes that MENU statement.

CONTROLLING WINDOWS

Programs that run in the Macintosh environment display their output in windows. Microsoft BASIC lets you define up to four windows for a BASIC program. There are three forms of the WINDOW statement (see figure 6.25). The WINDOW statement—with a window number, title, window coordinates, and window type—creates and opens a window. The WINDOW CLOSE statement followed by a window number closes a window

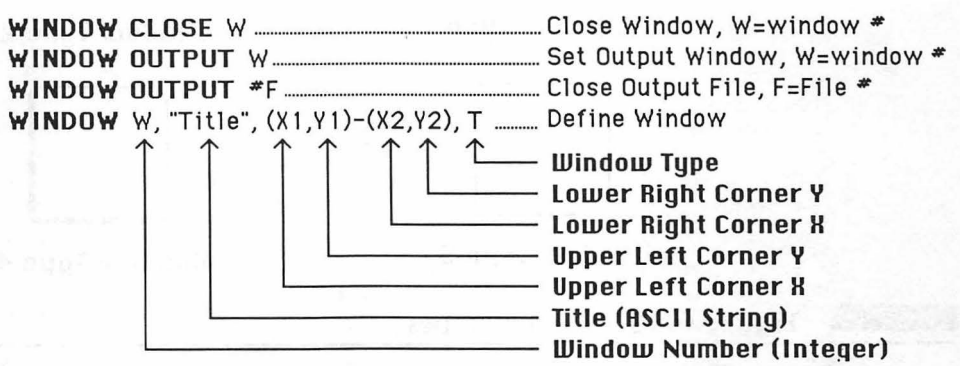


Figure 6.25 Window statements

and removes it from the screen. The WINDOW OUTPUT statement followed by a number makes a window or file the current output window. A program may have up to four windows on the screen, but only one is the current output window. When a new window is opened, it automatically becomes the current output window.

BASIC can create four window types: (1) a document window with a title bar and size box, (2) a dialog box, (3) a plain window with a thin border, and (4) a window with a shadow. All four are shown in figure 6.26.

BUTTONS

Buttons are used in a window or dialog box to let the user select an option or initiate some action. A program creates a button with the BUTTON statement, specifying the button number, state, title, coordinates, and type (see figure 6.27). The BUTTON CLOSE statement removes a button. There are three types of buttons, as shown in figure 6.28.

The DIALOG(1) statement returns the number of the button that the user most recently selected. The DIALOG statement does far more than simply report which button has been pushed, but more about that later. The other DIALOG functions report other events that the user initiates with the mouse.

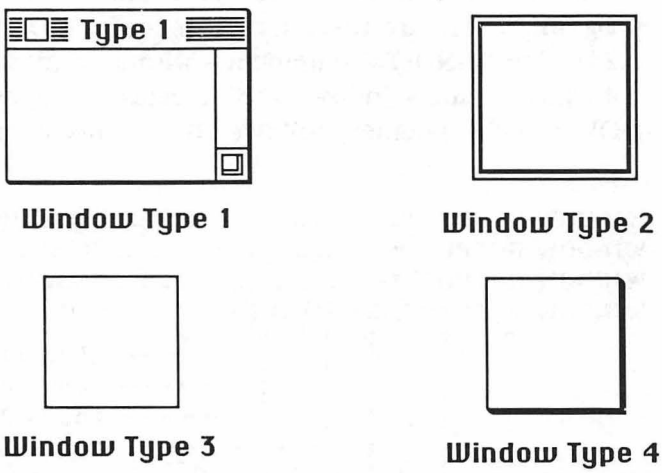


Figure 6.26 Window types

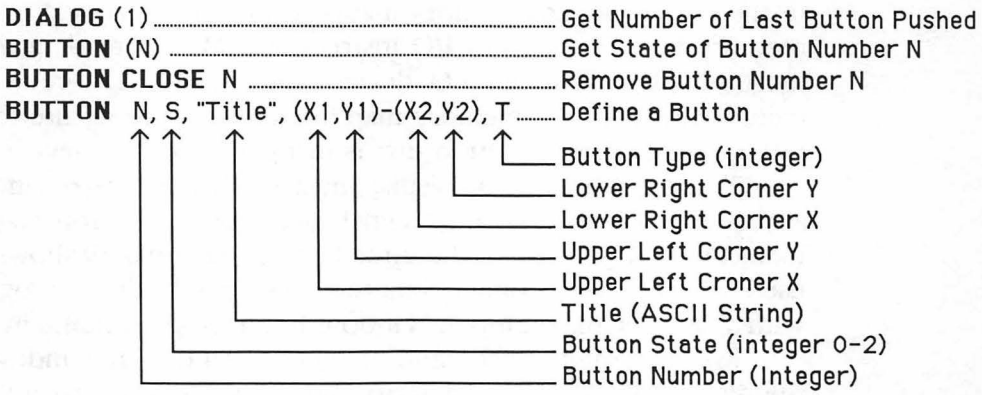


Figure 6.27 BUTTON and DIALOG statements

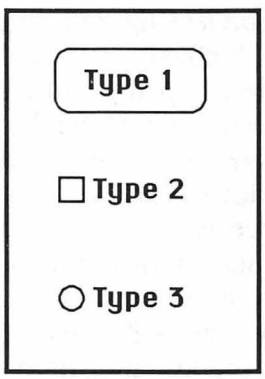


Figure 6.28 Button Types

A MENU AND WINDOW EXAMPLE—THE MEMORY DUMP PROGRAM

The Memory Dump program listed at the end of this chapter demonstrates how BASIC is used to create a program with windows, menus, and buttons. The program provides some interesting glimpses into the Macintosh memory, but the program should be used with caution. I/O devices

in the Macintosh are memory-mapped; they are accessed by read/write memory instructions, not I/O instructions. Merely reading the memory location assigned to an I/O device can affect it. Before you examine memory, refer to the memory map in chapter 10 to ensure that you stay out of any memory location that is mapped to an I/O device.

This program lets you display memory dumps in two windows and in any of three formats—hexadecimal, octal, or ASCII. You may also print memory dumps in any of the three formats. Figure 6.29 shows the menus used to control the program. Figure 6.30 shows the dump program display with a hexadecimal dump in Window 1 and an ASCII dump in Window 2.

Each Start item in the Run menu opens a display window and starts the display of an area of memory. When the user selects a Start item, a dialog box appears, asking for the starting address of the memory dump. When the user has typed the starting address, the dialog box disappears.

Selecting the Print item displays a dialog box that asks for the starting address and length of the dump; it also allows the user to select one of the three dump formats with a button (see figure 6.31). When the dialog box first appears, the buttons are disabled; after the user has entered the starting address and length of the dump, the buttons are enabled. The user must select a button in order to get out of the dialog box.

The Window1 and Window2 menus (shown in figure 6.29) select the formats for the dumps in Windows 1 and 2. When the program is first executed, the Repeat and Advance items are disabled. Once the user opens a dump window by using the Start item in the Run menu, the program enables the Repeat and Advance items for that window.

The program does its actual work in a background loop. The user selects a function by using the menus, and the background loop calls a routine to perform that function. The HandleMenu subroutine functions like a menu interrupt handler and executes when the user selects a menu item. The subroutine calls a routine specific to that menu item, which just



Figure 6.29 Memory Dump program menus

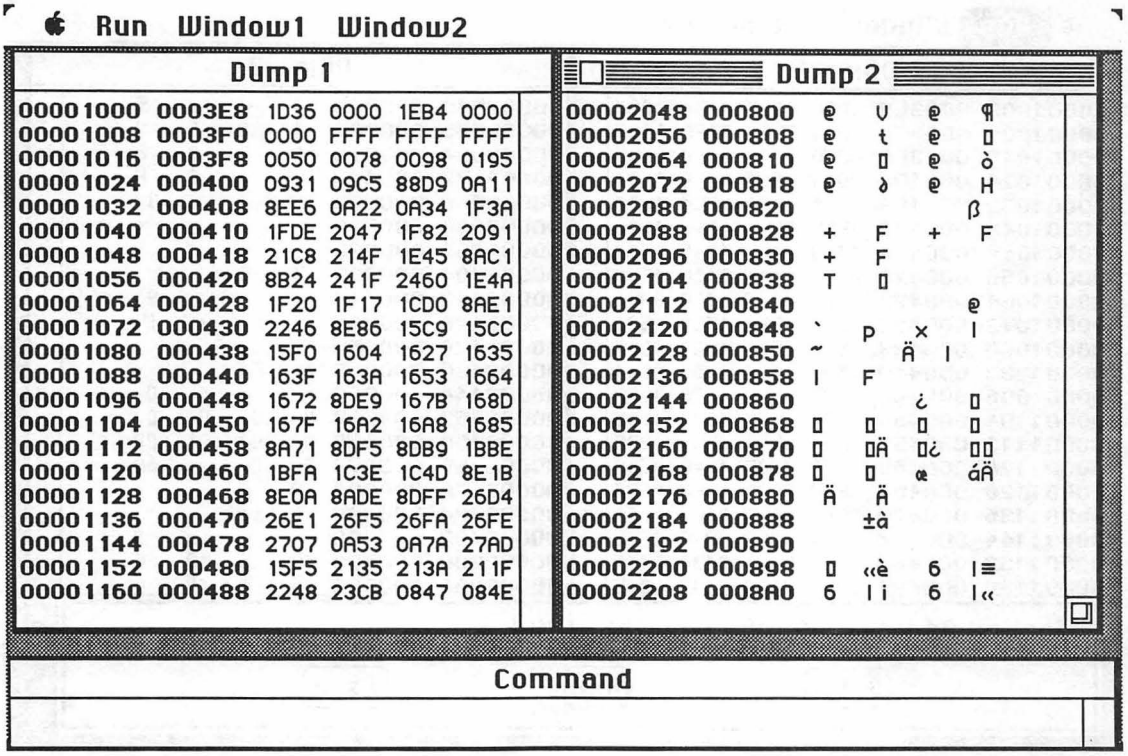


Figure 6.30 Memory Dump program display

sets a flag to indicate to the background loop what task must be done. In actuality, this interrupt-driven approach functions much like a polled control scheme.

But using the interrupt approach in this program gains us something in the handling of the End item from the Run menu and the handling of the dump-format selection items from the Window1 and Window2 menus. The program executes the END statement to end the program in the menu interrupt handler rather than in the background loop. As a result, the program ends sooner; it doesn't have to wait for the background loop to end it. The program also sets the dump-format variables, type1 and type2, in the interrupt handler.

The first things that the program does are to close any windows that were left open from a previous BASIC program; initialize the settings of the text font, size, and style; and set up the menus. Take a look at the InitMenus routine in the listing at the end of this chapter. It uses MENU

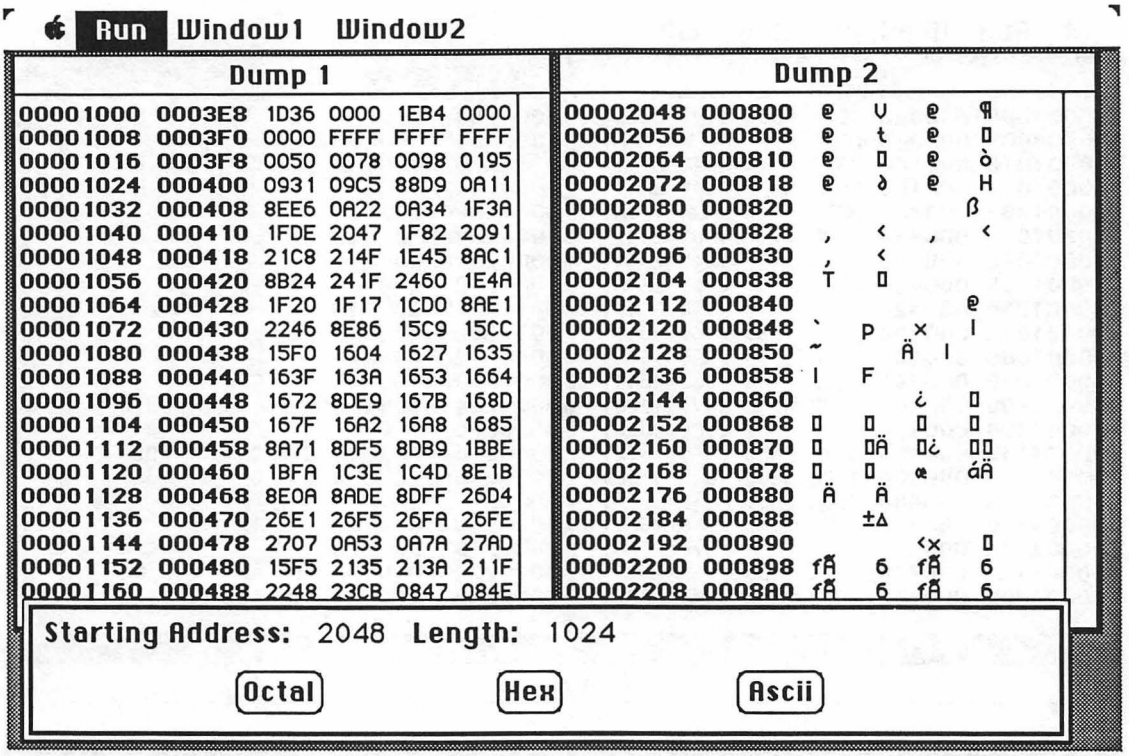


Figure 6.31 Print dialog box

statements to define all of the menus and items. Grouping menu items with their menu definitions, as is done here, is a good practice because it makes them easier to find for later modification.

The InitMenus routine also sets the type1 and type2 variables, and they select which format is used in each of the two dump windows. The program sets their default values right after the menu definitions because these values must correspond to the checked items in the menus. They could have been set elsewhere, but they are easier to find here if you later want to modify the program's default selections.

The background loop uses a TaskID variable to decide which of the background routines to execute. The routines called by the menu handler (HandleMenu) set the TaskID variable.

The Memory Dump program tasks are listed below:

Task	*Function*
Do1	Open Window 1 and start the dump in it (selected by the Start1 menu item)
Do2	Open Window 2 and start the dump
DoPrint	Print a dump (selected by the Print item in the Run menu)
DoRepeat1	Repeat the last dump in Window 1 (selected by the Repeat item in the Window1 menu)
DoAdvance1	Do another dump in Window 1, starting at the address where the last dump in this window finished (selected by the Advance item in the Window1 menu)
DoRepeat2	Repeat the last dump in Window 2
DoAdvance2	Do another dump in Window 2, starting where the last dump in this window finished

The Do1 and Do2 routines are identical except for the windows they use. The same goes for the DoRepeat1 and DoRepeat2, and DoAdvance1 and DoAdvance2, routines.

First Do1 sets up the typeface for use in the dialog box. Then it uses a WINDOW statement to create a dialog box, and PRINT and INPUT statements to create a prompt message and read data from the dialog box. When the user enters the starting address, the program reads it as a decimal number. Once the program has read the dialog-box data, it closes the dialog box with the statement WINDOW CLOSE 3.

Since there is never more than one dialog box open at a time, all dialog boxes use 3 for the window number. The two dump windows must have separate window numbers: 1 and 2. Window number 4 is used for printed output; this window never appears on the screen.

Once the dialog box is gone, the program sets up the dump variables' addr# (starting address), length# (length of dump), and MaxCol (number of columns in the display); it also calls one of the three dump subroutines OctDump, HexDump, or AsciiDump.

When the dump subroutine is finished, the Do routine turns on the Repeat and Advance items in the Windows menu, clears the TaskID, and returns to the background loop.

The Advance and Repeat routines work like the Do routines, but they do not need a dialog box to get a starting address. Nor do they need to open an output window, because it's already been opened by one of the Do routines.

All three of the dump subroutines OctDump, HexDump, and AsciiDump use a subprogram called HexConv to convert an eight-digit decimal address to a six-digit hexadecimal address for display in the dump. This subprogram uses the positional parameters doublenum# and hexnum$. (The actual variables passed by the calling programs have different names, which match the position of the subprogram's parameters in the parameter list.)

Because it uses its own private variables and is passed positional parameters, this subprogram could be copied to another program and used without modification. (Subprograms differ from subroutines in two ways: they are passed positional parameters, and they have private variables that cannot be changed by any other program or subroutine.)

When you are more familiar with the Memory Dump program (see listing 6.6) and are ready to make some modifications, you might want to program one of the following suggested improvements:

1. Move the Start1 and Start2 menu items to the Window menus.
2. Write a routine that would allow the user to enter a starting address as a hexadecimal number. The routine would convert that number to a decimal address for use by the PEEK function.
3. Write a routine that would find the size of the output window and calculate how many rows and columns of data would fit into it. The routine would set the MaxCol and length# variables before the dump subroutine is called. The user could then expand or contract the window, using its size box, and the Memory Dump program would fill the window.

Listing 6.6 Memory Dump Program

```
REM Memory Dump Program

DEFINT I-N,T

GOSUB CloseWindows
GOSUB InitText
GOSUB InitMenus
Background:
   IF TaskID = 0 THEN GOTO Background
   ON TaskID GOSUB Do1, Do2, DoPrint, DoRepeat1, DoAdvance1, DoRepeat2, DoAdvance2
   MENU
   GOTO Background

Do1:
   WINDOW 3,,(15,285)-(480,330),-2       'open dialog box
   CALL TEXTFONT(0)
   CALL TEXTSIZE(12)
   PRINT
   PRINT "Starting Address: ";
   CALL TEXTFONT(3)
   INPUT " ",Saddr1#
   WINDOW CLOSE 3                        'close dialog box
   WINDOW 1,"Dump 1",(5,40)-(250,285),1  'open output window
   addr# = Saddr1#
   GOSUB InitText
   IF type1 = 1 THEN MaxCol = 3:length#=84 ELSE MaxCol = 7:length#=168
   ON type1 GOSUB OctDump,HexDump,AsciiDump                    'do the memory dump
   Eaddr1#=addr#                         'save ending address
   MENU 2,1,1                            ' turn on repeat menu item
   MENU 2,2,1                            'turn on Advance item
   TaskID = 0
RETURN

Do2:
   WINDOW 3,,(15,285)-(480,330),-2    'open dialog box
   CALL TEXTFONT(0)
   CALL TEXTSIZE(12)
   PRINT
   PRINT "Starting Address: ";
   CALL TEXTFONT(3)
   INPUT " ",Saddr2#
   WINDOW CLOSE 3                        'close dialog box
```

Continued

Listing 6.6 *Continued*

```
  WINDOW 2,"Dump 2",(255,40)-(500,285),1    'open output window
  addr# = Saddr2#
  GOSUB InitText
  length# = 168
  IF type2 = 1 THEN MaxCol = 3:length#=84 ELSE MaxCol = 7:length#=168
  ON type2 GOSUB OctDump,HexDump,AsciiDump      'do the memory dump
  Eaddr2# = addr#                                'save ending addr
  MENU 3,1,1                                    'turn on repeat menu item
  MENU 3,2,1                                    'turn on Advance item
  TaskID = 0
RETURN

DoPrint:
  WINDOW 3,,(15,280)-(480,330),-2    'open dialog box
  CALL TEXTFACE(0)
  CALL TEXTFONT(0)
  CALL TEXTSIZE(12)
  by =25   'button top (y)
  bh = 20   'button heigth
  bx = 90   'x of first button
  bsep = 80   'button separation (x)
  b1len = 40   'button 1 length
  b2len = 30
  b3len = 40
  b1x=bx
  b2x=b1x+b1len+bsep
  b3x=b2x+b2len+bsep
  BUTTON 1,0,"Octal",(b1x,by)-(b1x+b1len,by+bh),1
  BUTTON 2,0,"Hex",(b2x,by)-(b2x+b2len,by+bh),1
  BUTTON 3,0,"Ascii",(b3x,by)-(b3x+b3len,by+bh),1
  PRINT "Starting Address: ";
  CALL TEXTFONT(3)
  INPUT ;" ",SaddrP#
  CALL TEXTFONT(0)
  PRINT "  Length: ";
  CALL TEXTFONT(3)
  INPUT ;" ",length#
  BUTTON 1,1
  BUTTON 2,1
  BUTTON 3,1
  WHILE DIALOG(0) <> 1: WEND        'wait for a button
  SelectedButton=DIALOG(1)
```

Continued

Listing 6.6 *Continued*

```
  BUTTON 1,0
  BUTTON 2,0
  BUTTON 3,0
  typeP = SelectedButton
  OPEN "LPT1:" FOR OUTPUT AS #5
  WINDOW OUTPUT #5
  GOSUB InitText
  addr#=SaddrP#
  IF typeP = 1 THEN MaxCol = 17 ELSE MaxCol = 31
  ON typeP GOSUB OctDump,HexDump,AsciiDump
  WINDOW CLOSE 3
  TaskID =0
  CLOSE #5
RETURN

DoRepeat1:
  WINDOW 1
  CALL MOVETO(2,12)
  addr#=Saddr1#
  IF type1 = 1 THEN MaxCol = 3:length#=84 ELSE MaxCol = 7:length#=168
  ON type1 GOSUB OctDump,HexDump,AsciiDump                    'do the memory dump
  Eaddr1# = addr#
  TaskID =0
RETURN

DoAdvance1:
  WINDOW 1
  Saddr1# = Eaddr1#
  addr#=Eaddr1#
  IF type1 = 1 THEN MaxCol = 3:length#=84 ELSE MaxCol = 7:length#=168
  ON type1 GOSUB OctDump,HexDump,AsciiDump        'do the memory dump
  Eaddr1# = addr#
  TaskID =0
RETURN

DoRepeat2:
  WINDOW 2
  CALL MOVETO(2,12)
  addr#=Saddr2#
  IF type2 = 1 THEN MaxCol = 3:length#=84 ELSE MaxCol = 7:length#=168
  ON type2 GOSUB OctDump,HexDump,AsciiDump                    'do the memory dump
  Eaddr2# = addr#
```

Continued

Listing 6.6 *Continued*

```
  TaskID =0
RETURN

DoAdvance2:
  WINDOW 2
  Saddr2# = Eaddr2#
  addr#=Eaddr2#
  IF type2 = 1 THEN MaxCol = 3:length#=84 ELSE MaxCol = 7:length#=168
  ON type2 GOSUB OctDump,HexDump,AsciiDump          'do the memory dump
  Eaddr2# = addr#
  TaskID =0
RETURN

InitText:
  CALL TEXTFONT(4) 'Monaco
  CALL TEXTSIZE(9) '9 point
  CALL TEXTFACE (0)
RETURN

InitMenus:
MENU 1,0,1,"Run"
MENU 1,1,1,"Start1"
MENU 1,2,1,"Start2"
MENU 1,3,1,"Print"
MENU 1,4,1,"End"
MENU 2,0,1,"Window1"
MENU 2,1,0,"Repeat"
MENU 2,2,0,"Advance"
MENU 2,3,1,"Octal"
MENU 2,4,2,"Hex"
type1 = 2
MENU 2,5,1,"Ascii"
MENU 3,0,1,"Window2"
MENU 3,1,0,"Repeat"
MENU 3,2,0,"Advance"
MENU 3,3,1,"Octal"
MENU 3,4,1,"Hex"
MENU 3,5,2,"Ascii"
type2 = 3
MENU 4,0,0," "
MENU 5,0,0," "
MENU ON
```

Continued

Listing 6.6 *Continued*

```
ON MENU GOSUB HandleMenu
RETURN

CloseWindows:
   WINDOW CLOSE 1
   WINDOW CLOSE 2
   WINDOW CLOSE 3
   WINDOW CLOSE 3
RETURN

HandleMenu:
   SelectedMenu = MENU(0)
   SelectedItem = MENU(1)
   ON SelectedMenu GOSUB RunMenu, Window1Menu, Window2Menu
RETURN

RunMenu:
   ON SelectedItem GOSUB Start1Item,Start2Item,PrintItem,EndItem
RETURN

Start1Item:
   TaskID = 1
RETURN

Start2Item:
   TaskID = 2
RETURN

PrintItem:
   TaskID = 3
RETURN

EndItem:
   END
RETURN

Window1Menu:
   ON SelectedItem GOSUB Repeat1,Advance1,Octal1,Hex1,Ascii1
RETURN
```

Continued

Listing 6.6 *Continued*

```
Repeat1:
   TaskID = 4
RETURN

Advance1:
   TaskID = 5
RETURN

Octal1:
   type1 = 1   'set type to octal
   MENU 2,3,2
   MENU 2,4,1
   MENU 2,5,1
   MENU
RETURN

Hex1:
   type1 = 2   'set type to hex
   MENU 2,3,1
   MENU 2,4,2
   MENU 2,5,1
   MENU
RETURN

Ascii1:
   type1 = 3   'set type to ascii
   MENU 2,3,1
   MENU 2,4,1
   MENU 2,5,2
   MENU
RETURN

Window2Menu:
   ON SelectedItem GOSUB Repeat2,Advance2,Octal2,Hex2,Ascii2
RETURN

Repeat2:
   TaskID =6
RETURN
```

Continued

Listing 6.6 *Continued*

```
Advance2:
  TaskID = 7
RETURN

Octal2:
  type2 = 1   'set type to octal
  MENU 3,3,2
  MENU 3,4,1
  MENU 3,5,1
  MENU
RETURN

Hex2:
  type2 = 2   'set type to hex
  MENU 3,3,1
  MENU 3,4,2
  MENU 3,5,1
  MENU
RETURN

Ascii2:
  type2 = 3   'set type to ascii
  MENU 3,3,1
  MENU 3,4,1
  MENU 3,5,2
  MENU
RETURN

Hexdump:
EndAddr#=addr#+length#
WHILE (addr# < EndAddr#)
  GOSUB addrDEC
  CALL HexConv(addr#,addr$)
  CALL TEXTFACE(1):PRINT addr10$;" ";addr$;" ";:CALL TEXTFACE(0)
   FOR col% = 0 TO MaxCol STEP 2
     GOSUB hexit2
     PRINT num$;   ' PRINT the first byte
     addr#=addr#+1          ' increment the address
     GOSUB hexit2
     PRINT num$;" ";
     addr#=addr#+1
   NEXT col%
```

Continued

Listing 6.6 *Continued*

```
   PRINT 'generate a CR
WEND
RETURN

hexit2:
   byte% = PEEK(addr#)
   temp$ = HEX$(byte%)
   IF LEN(temp$) = 1 THEN num$ = ("0" + temp$) ELSE num$ = temp$
RETURN

addrDEC:
  addr10$=""
  tempb$ = STR$(addr#)
  j = 9 - LEN(tempb$)
   IF j = 0 THEN RETURN
   FOR k=1 TO j
      addr10$=addr10$+"0"
   NEXT k
   L=LEN(tempb$)-1
   addr10$ = addr10$ + MID$(tempb$,2,L)
RETURN

Octdump:
EndAddr#=addr#+length#
WHILE (addr# < EndAddr#)
  GOSUB addrDEC
  CALL HexConv(addr#,addr$)
  CALL TEXTFACE(1):PRINT addr10$;" ";addr$;" ";:CALL TEXTFACE(0)
   FOR col% = 0 TO MaxCol STEP 1
      GOSUB octit2
      PRINT num$; " ";   ' PRINT the byte
      addr#=addr#+1            ' increment the address
   NEXT col%
   PRINT 'generate a CR
WEND
RETURN

octit2:
   byte% = PEEK(addr#)
   temp$ = OCT$(byte%)
   L=LEN(temp$)
```

Continued

Listing 6.6 *Continued*

```
  IF L = 1 THEN num$ = ("00" + temp$) ELSE IF L = 2 THEN num$=("0"+temp$) ELSE num$ = temp$
RETURN

AsciiDump:
EndAddr#=addr#+length#
WHILE (addr# < EndAddr#)
  GOSUB addrDEC
  CALL HexConv(addr#,addr$)
  CALL TEXTFACE(1):PRINT addr10$;" ";addr$;" ";:CALL TEXTFACE(0)
  FOR col% = 0 TO MaxCol STEP 2
     GOSUB Asciilt
     PRINT char$;              ' PRINT the ascii character
     addr#=addr#+1             ' increment the address
     GOSUB Asciilt
     PRINT char$; " ";             ' PRINT the ascii character
     addr#=addr#+1             ' increment the address
  NEXT col%
  PRINT 'generate a CR
WEND
RETURN

Asciilt:
  byte% = PEEK(addr#)
  IF byte% < 32 THEN byte% = 32
  char$ = CHR$(byte%)
RETURN



SUB HexConv(doublenum#,hexnum$) STATIC
hexnum$ = ""
IF doublenum# > 16777215# THEN b#=0 ELSE b# = doublenum#
FOR i = 6 TO 1 STEP -1
  a# = 16^(i-1)
  t% = INT(b#/a#)
  b# = b# - (a#*t%)
  hexnum$ = hexnum$ + HEX$(t%)
NEXT i
END SUB
```

CHAPTER

7 MACINTOSH BASIC

Apple's Macintosh BASIC
Getting into Macintosh BASIC
Using Macintosh BASIC
Variables, Constants, and Numeric Operations
Control Structures
Graphics
Sound
The CALC Program
Altering the CALC Program

APPLE'S MACINTOSH BASIC

By producing Macintosh BASIC, Apple Computer has ensured that there is a BASIC interpreter that fits the Macintosh environment. As do other Apple products, it serves a function in its own right and also demonstrates how a product can be designed to make full use of the Macintosh's capabilities.

You might ask, Why another BASIC? but an examination of Macintosh BASIC will show that it has enough unique features to make it a product distinct from Macintosh Pascal and Microsoft BASIC.

GETTING INTO MACINTOSH BASIC

Anyone who has used Microsoft BASIC in the past will feel right at home with Macintosh BASIC. Many key words are the same, and many others are just slightly different versions.

Macintosh BASIC has a familiar set of features to allow structured programming. Like Macintosh Pascal, it does some of its syntax checking when each line of the program is entered (additional checking is done when you execute the program). It has debugging capabilities comparable to those of other interpreters, and like them, it makes good use of the Macintosh environment. Among the key structured programming features of Macintosh BASIC are:

1. Indented lines
2. Comments in program lines
3. Line labels as well as numbers
4. Long variable and label names
5. User-defined functions
6. Functions with local variables

Macintosh BASIC has automatic line indentation to make programs easier to read. It can also display the program text in a variety of type fonts and sizes selected from a type font menu.

In some ways, Macintosh BASIC is a significant departure from "the other BASIC." It has the closest thing to multitasking that you will find in a Macintosh programming system—up to seven programs can be loaded at the same time. Each program runs in its own window. Only one executes at any one time, but you may switch to another and then return to the first

where you left off. For example, you could run a financial program in one window, and stop it periodically to use a calculator simulator in another window. You could run several simulations with slightly different parameters and compare results. The possibilities are limited only by your ingenuity.

Macintosh BASIC uses the same Pascal data structures that the Pascal interpreter, the Finder, the toolbox, and the Macintosh Development System use. Data structures used internally by Macintosh BASIC are the same ones used by the system, making toolbox and system interfacing much easier for language designers. The result is more access in the language to toolbox and system routines.

Access to a large number of toolbox facilities makes Macintosh BASIC a good candidate for programming one-time applications that require toolbox access or for prototyping applications designed for the Macintosh environment. Most programmers use the Menu, Window, Dialog, Event, Font, and QuickDraw routines, and only rarely need to call the others.

Another important area of compatibility between Macintosh BASIC and Apple's other languages is in numeric data types. Macintosh BASIC uses the same IEEE floating-point standard that Macintosh Pascal and Microsoft BASIC support. It also uses the very same numeric calculation routines that Macintosh Pascal uses—the Standard Apple Numeric Environment (SANE). In addition to the IEEE floating-point standard, SANE supports an important set of numeric data structures and calculations for financial applications. A calculation done in a Macintosh BASIC program should produce the same results as one done in a Macintosh Pascal program.

USING MACINTOSH BASIC

You will discover how to use Macintosh BASIC by exploring the menus and windows you use to control it. On opening the disk icon (figure 7.1), you see a familiar disk directory window with several Macintosh BASIC documents (programs) and the BASIC interpreter icon.

Once you open the Macintosh BASIC icon, the screen displays the BASIC menu bar and the program Text window. You enter the program by typing it in the Text window. The Text window uses an I-beam pointer. It has a size box and scroll bars and uses the same Edit menu as other applications (figure 7.2).

If you are familiar with MacWrite, you'll be right at home in the Macintosh BASIC Text window.

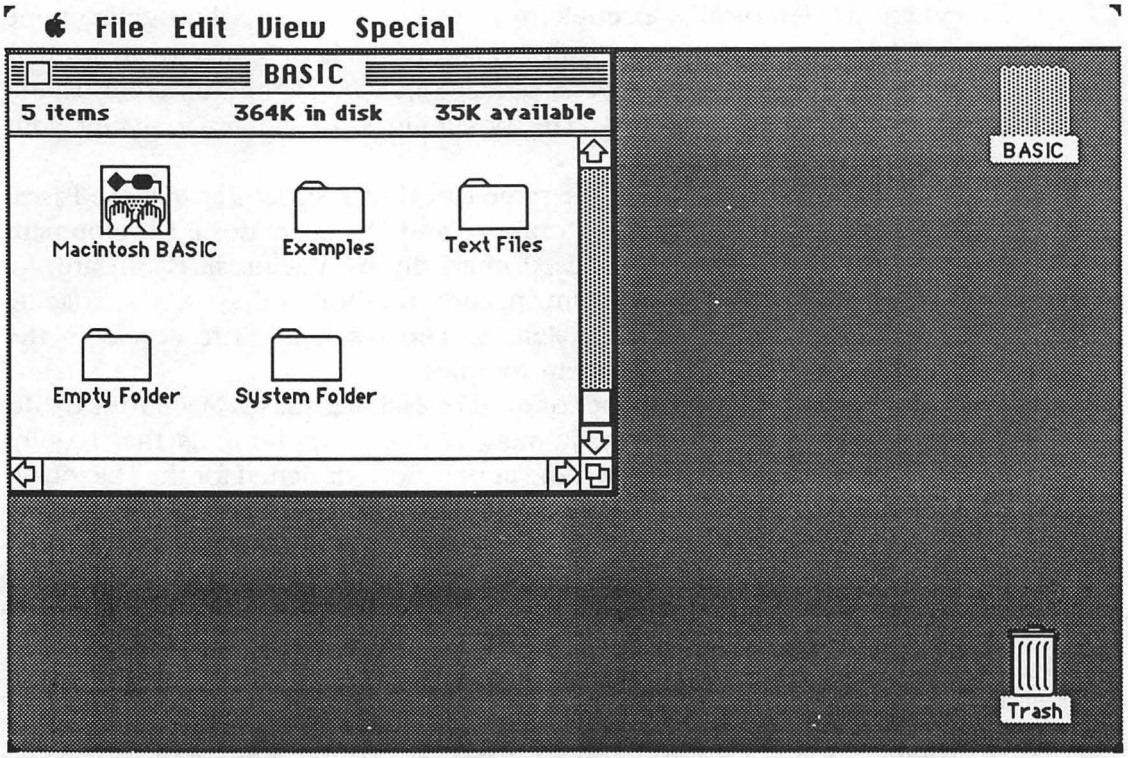


Figure 7.1 BASIC disk directory

The window displays your program text in the font style and size that you select from the Fonts menu (figure 7.3). The BASIC interpreter's syntax checking also affects the appearance of your program text. When you type RETURN at the end of a program line, the interpreter performs its syntax check. It either displays an error message in a dialog box (figure 7.4), or if your code passes the test, it sets in boldface all of the BASIC key words in the line.

Another familiar editing facility is in the Search menu (figure 7.5). It performs a search like the Search menus in other applications. The menu contains a list of search commands, so you can specify the character string you want to find, perform the search, do a search and replace, or replace all occurrences of the string.

Choosing What to Find from the Search menu gets you a dialog box that lets you specify the search string (figure 7.6, page 128). You may specify a string for which to search (and, optionally, another string to

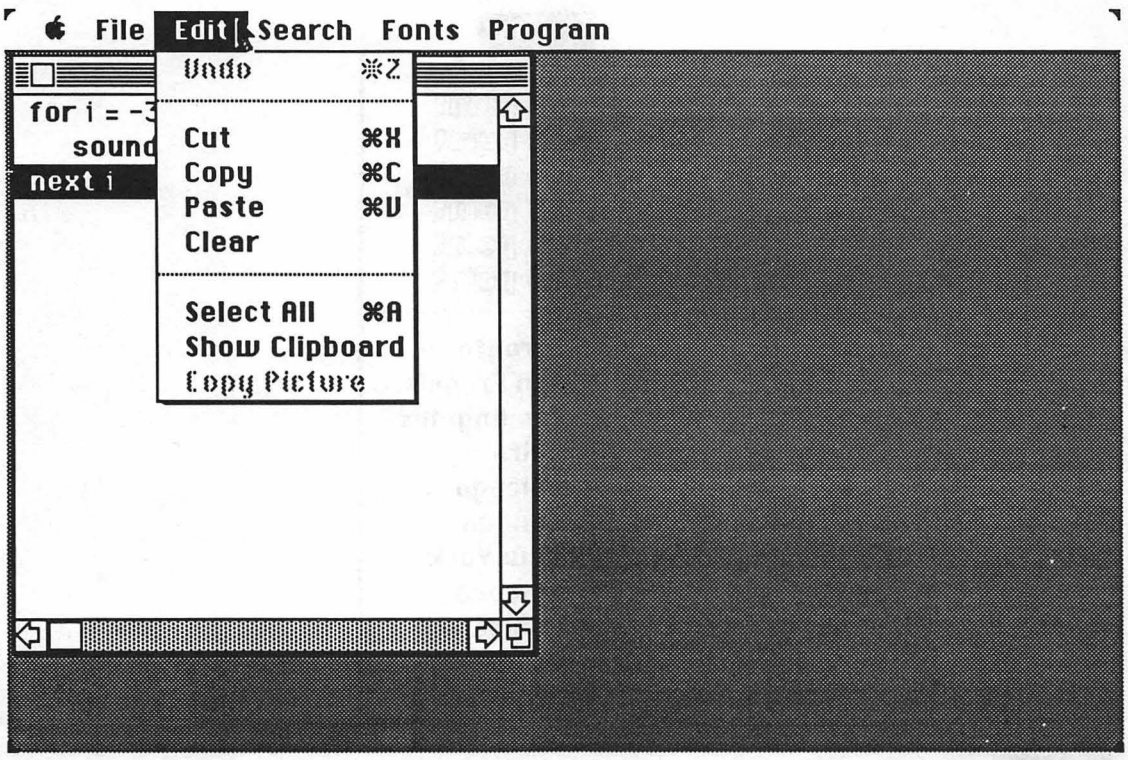


Figure 7.2 Text window and Edit menu

replace it), whether or not to ignore case, and whether or not to limit the search to whole words that stand alone or include matching strings that are imbedded in other words.

Once you've entered your program, you will want to save it. Pull down the File menu and you will see your options (figure 7.7). You may save the program (Save Text) or save it under another name (Save a Copy In . . .). If you choose to close the Text file, BASIC will ask if you want to save it and prompt you for a file name. Print Quick prints a standard-quality listing with the key words in boldface. Open Program File does just what you would expect: it opens a program's Text file in a new Text window. New opens a new Text window but doesn't close any existing windows.

Macintosh BASIC has three types of program windows—the Text window that you have already seen, the Variables window, and the output window (figure 7.8, page 129). If a program does not use the toolbox



Figure 7.3 Fonts menu

window-manager routines to create custom output windows, all of its output appears in the Macintosh BASIC output window. The Variables window shows the current values of all of a program's variables (not arrays) but can be used only when the debugger is on.

You may have multiple copies of Text, Variables, and output windows open on the screen at the same time, but only one window may be active.

Using the Program menu (figure 7.9, page 130), you can run, halt, or restart a program (Go), control the debugger, save a binary copy of the compiled program code, or control syntax checking. The Turn Checking On/Off item in the Program menu enables or disables automatic syntax checking while you are typing in a program's text. You will sometimes find it more convenient to type in the entire program and then do the syntax check. You can use the Check Syntax item in the menu to get BASIC to check the program's syntax at any time, even if automatic checking is on. BASIC always checks the program's syntax when you run the program.

The effect of selecting Run depends on which window is active. If a Text window is active, BASIC opens an output window for the program that is in the Text window and executes the program. If that program

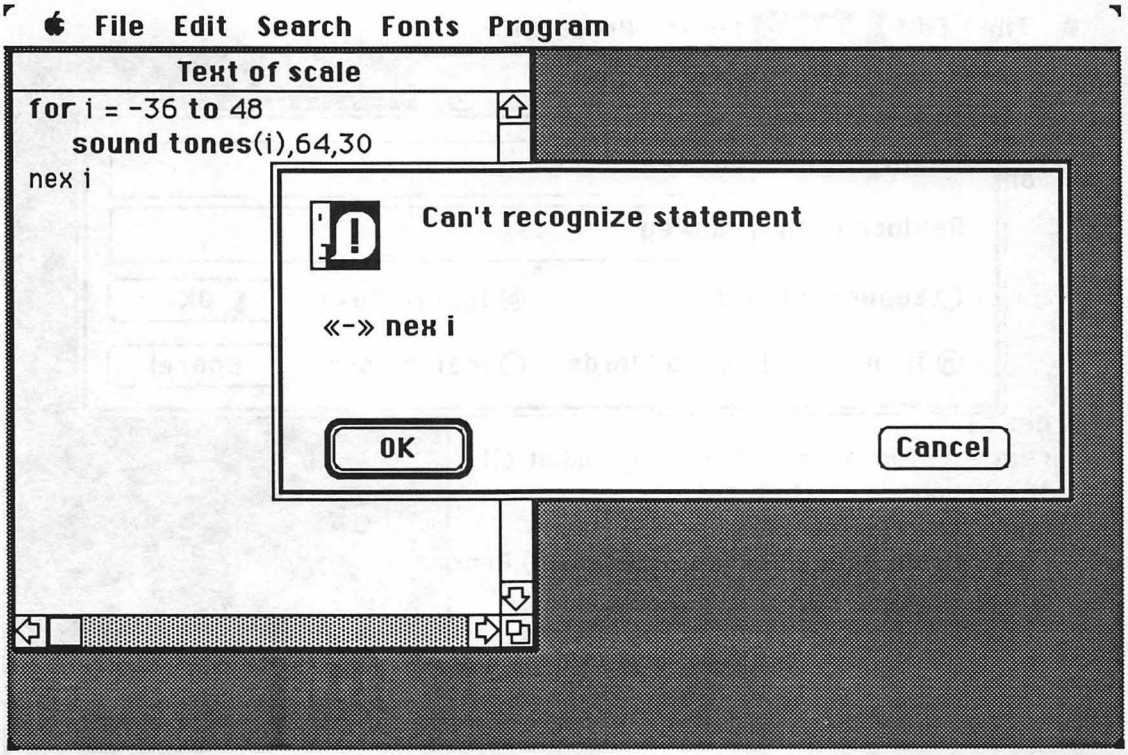


Figure 7.4 Program text with error message

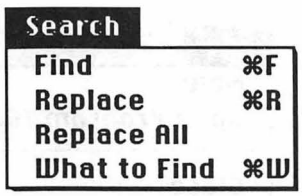


Figure 7.5 Search menu

already has an output window open, BASIC uses it and does not open another. If an output window is active when you select Run, BASIC runs the program that created the window.

Run Another is similar to Run, but it creates another output window and runs the program. The program acts like an additional copy; it has its own variables and runs independently of the original copy.

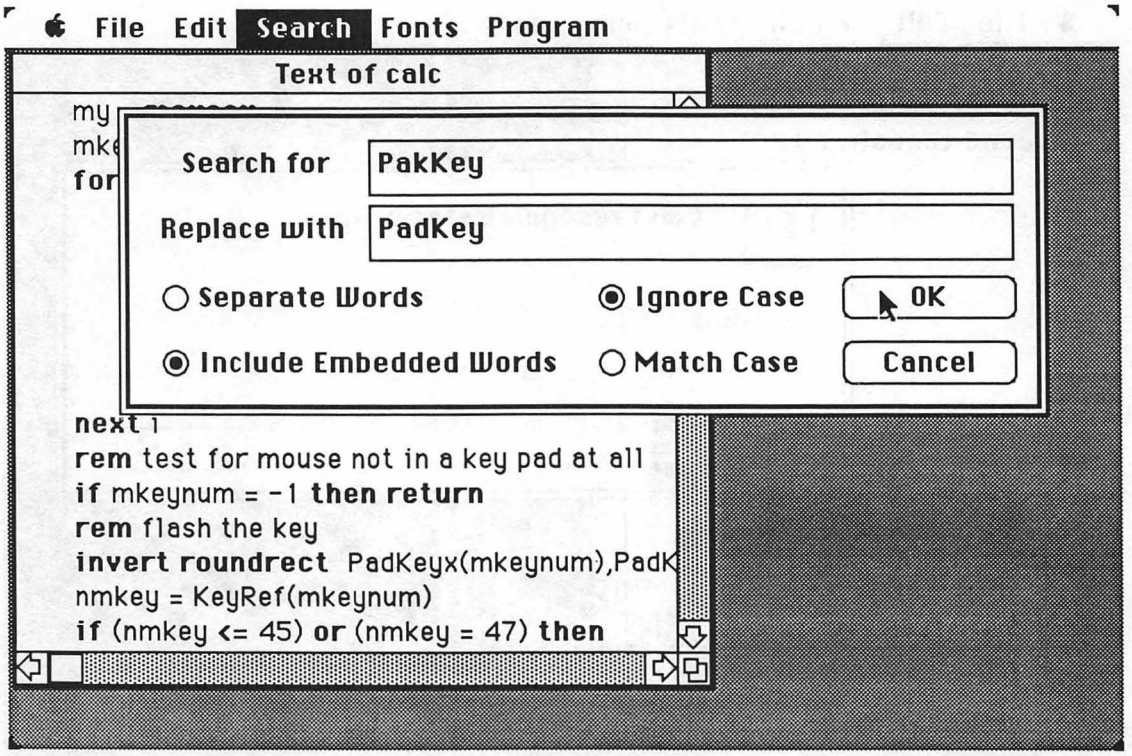


Figure 7.6 Search dialog box

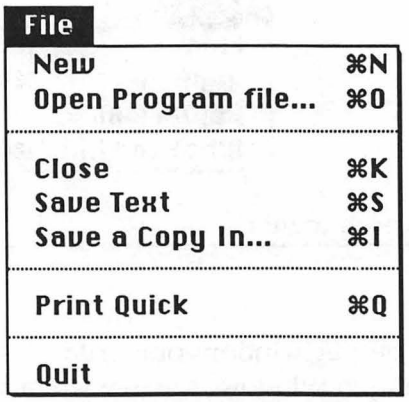


Figure 7.7 File menu

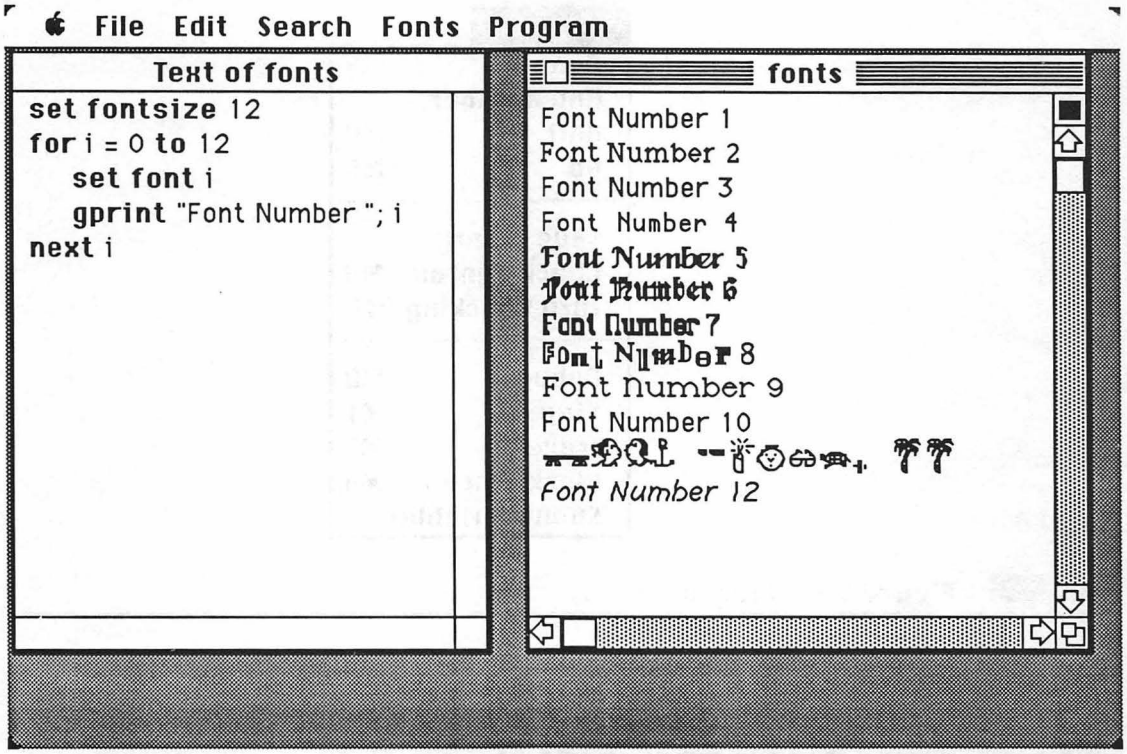


Figure 7.8 Macintosh BASIC windows

The first item (Debug) in the debug section of the Program menu turns on the debugger and enables the other debug items. The Step item executes one line of the BASIC program and advances the pointer in the Text window to the next line. The Trace item runs the program with the pointer following along, showing which line is currently executing. Block Trace combines features of Trace and Step. While the program is executing lines within a control structure block (CALL, CASE, DO, FOR-NEXT, IF-ENDIF, PERFORM, WHEN), it runs as if it were in Trace. While not in a control structure block, the program runs as if it were in Step.

Show Variables opens a Variables window for the program. The variable values are updated in the window whenever they are changed by the program.

In figure 7.10, note the icon at the top of the output window's scroll bar; it shows us the status of the program.

Figure 7.11 illustrates the different status icons.

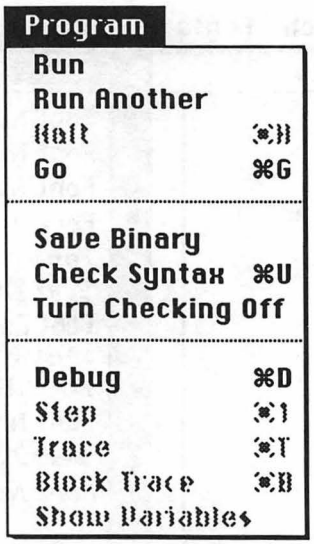


Figure 7.9 Program menu

VARIABLES, CONSTANTS, AND NUMERIC OPERATIONS

Variables, constants, and arrays must be one of the standard Macintosh BASIC data types: Boolean, string, or numeric. There are six forms of the numeric data type, each with a different storage requirement, precision, and range of values (see table 7.1). A variable, constant, or array's type and form are specified by a special character, which is the last character of its name. If the last character is not a special character, the variable, constant, or array is the numeric type with the double-precision real form.

A string constant is called a *literal* and, unlike other constants, is not identified by its ending character. It is instead enclosed in quotes. If it were otherwise, you could not include the identifying character ($) in a string. String variables and arrays, on the other hand, are identified by the ending character just like other variable and array types.

Note that the percent sign (%) defines an integer number, not a percentage. The constant 8% has a value of 8, not .08.

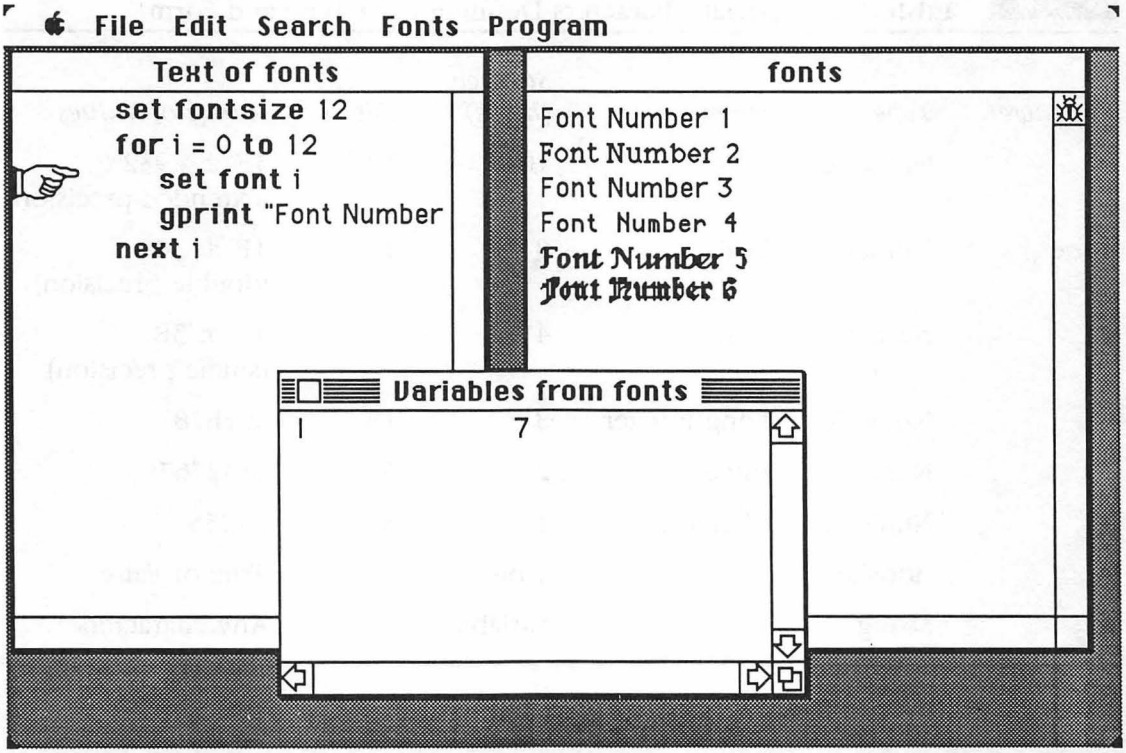


Figure 7.10 Windows with Debug on

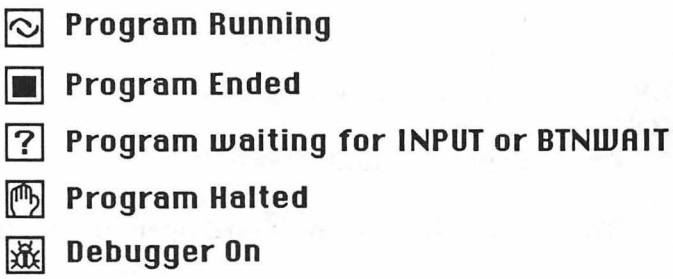


Figure 7.11 Status icons

Table 7.1 Special Characters Defining Data Type and Form

Character	*Type*	*Form*	*Storage (bytes)*	*Digits*	*Range of Values*
\	Numeric	Real	10	19	1E ± 4932 (extended precision)
none	Numeric	Real	8	15	1E ± 308 (double precision)
!	Numeric	Real	4	7	1E ± 38 (single precision)
#	Numeric	Long Integer	8	18	±1E18
%	Numeric	Integer	2	5	±32767
(c)	Numeric	Character	1	3	0–255
	Boolean		1 bit		True or False
$	String		Variable		Any characters

The Macintosh BASIC numeric operators are as follows:

Symbol	*Operation*
+	Add
−	Subtract
/	Divide
*	Multiply
DIV	Integer division
MOD	Modulo
^	Exponentiation

The Macintosh BASIC relational operators are listed below:

Symbol	*Operation*
=	Equal
<> or ><	Not equal
>	Greater than
<	Less than
>= or =>	Greater than or equal
<= or =<	Less than or equal

In addition to the usual set of BASIC numeric operators and functions, Macintosh BASIC has a set of options that control how the language deals with numeric operations. They allow you to control which floating-point exceptions cause an error event, which cause the program to stop, how numbers are rounded, at what precision level BASIC does numeric calculations, and how many digits BASIC uses to display numbers. The default precision is extended-precision real (floating-point)—the highest precision level. You may set lower precision levels for compatibility with other compilers, but they won't increase execution speed.

Macintosh BASIC also has two numeric functions for financial calculations that are not usually found in BASIC, ANNUITY and COMPOUND. Given the interest rate and period, ANNUITY returns the present value of an annuity. COMPOUND returns the balance of an interest-bearing account, given the interest rate and number of periods. COMPOUND actually returns the balance, assuming one unit (such as one dollar) was invested. You multiply the results by the number of units (dollars) to get the actual balance.

CONTROL STRUCTURES

Most of the control structures in Macintosh BASIC will be familiar to BASIC programmers, but there are a few interesting variations. FOR-NEXT loops, DO loops, GOSUB-RETURN, GOTO, and IF-THEN-ELSE can be used the way they're used in other versions of BASIC. However, the IF-THEN-ELSE construct has a variation in Macintosh BASIC that makes it better suited to structured programming. In most other BASIC languages, IF-THEN-ELSE must all be on one program line. But with the ENDIF key word, Macintosh BASIC allows blocks of code to be included in the IF-THEN-ELSE construct.

In old BASIC, IF-THEN-ELSE appears as:

```
IF condition THEN statement1:statement2 ELSE statement3:statement4
```

In Macintosh BASIC, the construct appears as:

```
IF
        statement1
        statement2
ELSE
        statement3
        statement4
ENDIF
```

BASIC recognizes the ELSE key word as ending the code block between IF and ELSE. ENDIF defines the end of the code block between ELSE and ENDIF. With ENDIF, you can write programs that are much easier to read and understand and, consequently, easier to debug.

Macintosh BASIC does not have the ON GOSUB statement found in other BASIC languages. Instead, it uses SELECT and CASE statements; they work like the CASE statement in Pascal. SELECT specifies a variable that determines which CASE statement to execute. Each CASE contains a constant or literal, and if it equals the value of the variable, CASE executes all of the program lines between it and the next CASE. Here is an example of SELECT and CASE statements.

```
SELECT FontNum
    CASE 0
        FontName$ = "System Font"
    CASE 1
        FontName$ = "Application Font"
    CASE 2
        FontName$ = "New York"
    CASE 3
        FontName$ = "Geneva"
    CASE ELSE
        FontName = "Not Installed"
END SELECT
```

Macintosh BASIC uses WHEN statements to specify what code to execute when an asynchronous event or error occurs (for instance, a keyboard event, a menu selection, or a window size-box selection).

GRAPHICS

Macintosh BASIC has several built-in graphics statements and can also access the QuickDraw routines from the toolbox. The built-in routines can plot points and lines, draw various shapes, and draw characters in the Macintosh fonts and type styles. Don't be surprised if these graphics look familiar. Macintosh Pascal, Microsoft BASIC, and Macintosh BASIC all use the same toolbox routines to create graphics. It's no accident, then, that their graphics statements are similar.

The PLOT and PENSIZE routines can plot individual points or draw lines. PENSIZE determines the size of the point or line drawn; PLOT specifies where to draw it. When supplied with the x and y coordinates of a point, PLOT plots the point. When you supply PLOT with a second set

of coordinates, it draws a line from the first coordinate pair (x, y) to the second. You can actually continue to draw connected lines by supplying additional x, y coordinates. Each coordinate pair is separated from the others by a semicolon. Here are some PLOT statements:

```
PLOT 10,15              ! Plot a point at x = 10, y = 15
PLOT 10,15;20,50        ! Draw a line from x = 10, y = 15 to x = 20, y = 50
PLOT 10,10;10,50;30,50;30,10      ! Draw a rectangle
```

The plotting routines simulate drawing with a pen. The PENSIZE (x, y) statement sets the size of the pen nib (and the size of the mark it makes) to a rectangle x pixels wide and y pixels high. To start plotting, just issue a PLOT statement with at least one set of x, y coordinates. The software simulates a pen moving to that spot and then puts the pen down (begins making a mark). As long as you keep supplying new coordinates, the pen stays down and draws a line as it moves. You can lift the pen and make it stop drawing by issuing a PLOT statement without x, y coordinates.

You can use the PLOT statement to draw a rectangle (figure 7.12), but there are easier ways to draw basic geometric figures. BASIC has a set of routines for drawing rectangles, round rectangles, and ovals. Besides drawing these figures, BASIC can erase, invert (change black pixels to white and vice versa), or fill them with a pattern. The rectangle statements look like this:

```
FRAME RECT x1,y1;x2,y2      ! draw a rectangle
ERASE RECT x1,y1;x2,y2      ! erase the rectangle
PAINT RECT x1,y1;x2,y2      ! fill a rectangular area with a pattern
INVERT RECT x1,y1;x2,y2     ! invert pixels in a rectangular area
```

The x1, y1 coordinates specify the upper left-hand corner of the rectangle, and x2, y2 specify the lower right-hand corner. The FRAME, ERASE, PAINT, and INVERT statements also work with round rectangles and ovals. When specifying an oval, you still use two sets of x, y coordinates (figure 7.13). They specify a rectangle; the oval is drawn inside the rectangle so that the oval just touches each side of the rectangle at one point.

To draw a round rectangle, you must include an extra set of x, y coordinates to specify how the corners are to be drawn. The corners' curvature is specified as if you were going to draw an oval in the upper left-hand corner (figure 7.14).

If you want to fill one of these shapes with a pattern, you use the Paint command, specify all of the coordinates necessary to draw the shape, and

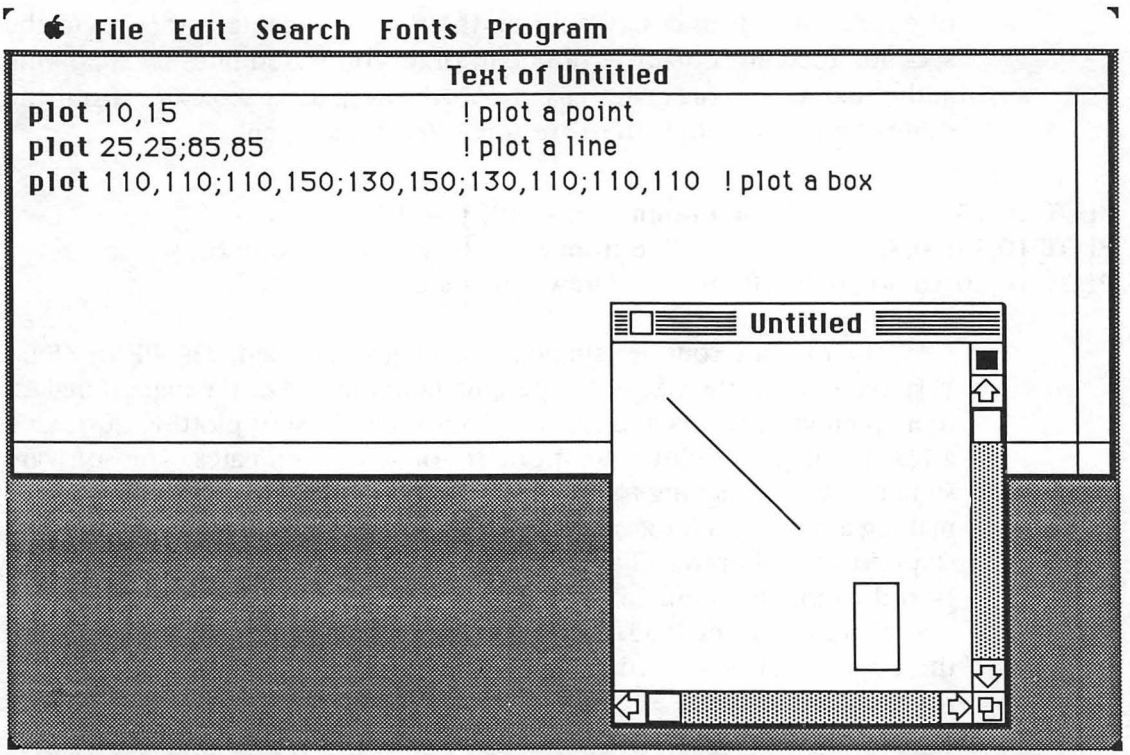


Figure 7.12 PLOT statements and results

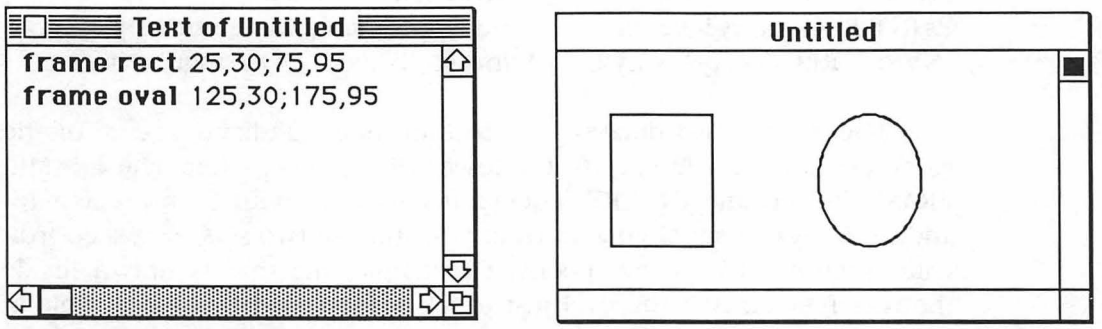


Figure 7.13 Rectangle and oval

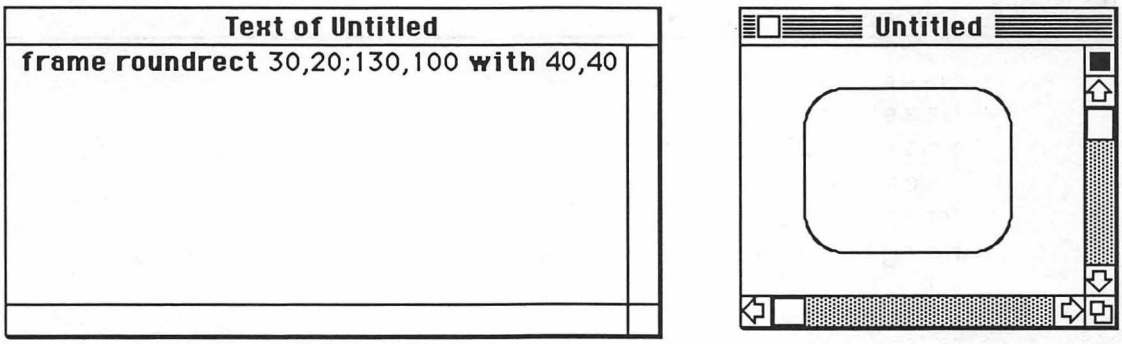


Figure 7.14 Round rectangle

specify the pattern number (listing 7.1). There are thirty-eight patterns—the same ones used by MacPaint (figure 7.15).

SET PENSIZE determines the thickness of the lines drawn by all of the FRAME statements. SET PENMODE also affects the way BASIC draws shapes against the existing picture on the screen. When you set PENMODE to a number between 8 and 15, the number determines how the shape's pixels are laid down on the screen. Only the pixels under the pen point are affected (remember, SET PENSIZE sets the size of the pen point). PENMODE numbers 8 through 15 lay down pixels in the following ways:

PENMODE	*Method*
8	Replace existing pixels
9	OR new pixels with existing pixels
10	XOR new pixels with existing pixels
11	Clear—wherever new black pixels overlay old black pixels, set pixels to white
12	Method 8; then invert all pixels
13	Method 9; then invert all pixels
14	Method 10; then invert all pixels
15	Method 11; then invert all pixels

Listing 7.1 Patterns

```
basex = 8
basey = 8
xmax = 370
xsep =8
ysep = 16
heigth = 32
width = 32
x = basex
y = basey
set fontsize 9
set font 0
for i = 0 to 38
    set pattern i
    paint rect x,y;x+width,y+heigth
    set pattern 0
    frame rect x,y;x+width,y+heigth
    set penpos x+(width/2)-4,y+heigth+10
    gprint i
    x = x + width + xsep
    if x > xmax then
        x = basex
        y = y + heigth + ysep
    endif
next i
```

GPRINT works just like the PRINT statement, but it prints in the font that you last selected. You select the font and size with SET FONT and SET FONTSIZE. GPRINT lays down characters over existing graphics. You use SET GTEXTMODE to determine how that happens; this statement works for fonts just like SET PENMODE does for graphics.

You can print in fonts (by using GPRINT) and in ordinary characters (by using PRINT) in the same output window. Each has its own insertion point, so they may be printing in different parts of the window. You use SET PENPOS to specify where GPRINT will print in the output window.

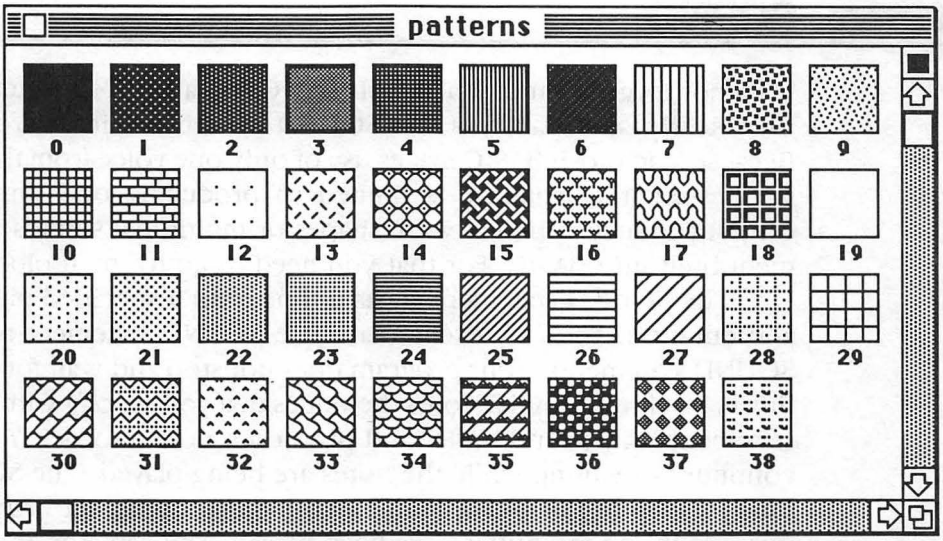


Figure 7.15 Patterns

Graphic Parameters

SET PENPOS	Set insertion point for GPRINT
SET GTEXTMODE	Set graphics mode for GPRINT
SET PENMODE	Set graphics mode for drawing
SET PENSIZE	Set size of pen nib for drawing
SET FONT	Set font for GPRINT
SET FONTSIZE	Set font size for GPRINT
SET PATTERN	Set pattern used by PAINT

Graphics Statements

PLOT	Plot a point or draw a line
FRAME	Draw the outline of a shape
ERASE	Erase a shape (set all pixels to white)
INVERT	Invert a shape (change value of all pixels)
PAINT	Fill a shape with a pattern

Graphics Shapes

RECTANGLE	Rectangle
ROUNDRECT	Rectangle with rounded corners
OVAL	Oval (ellipse)

SOUND

Let's not forget sound. Macintosh BASIC has a SOUND statement, but it isn't nearly as capable as the SOUND statements in Microsoft BASIC. Because Macintosh BASIC makes use of only one voice from the four-voice synthesizer, the language is limited to producing one simple tone. It cannot produce complex wave shapes or multitonal sounds with a statement built into BASIC. For that you need to go to the toolbox.

The SOUND statement starts a tone with a specified pitch, volume, and duration. You may execute a single SOUND statement or a series of SOUND statements. Your program does not stop and wait for the tones to finish; Macintosh BASIC stores the values that you specify with the SOUND statements in a sound buffer and plays each note in order. Your program continues executing while the notes are being played. The SOUNDOVER function returns a Boolean value that indicates whether or not all of the notes in the sound buffer have been played. You can stop the notes from being played and flush the sound buffer by executing the STOPSOUND statement.

The SOUND statement requires that you supply the frequency, amplitude, and duration of the note:

SOUND frequency, amplitude, duration

You specify the frequency in cycles per second, with a numeric expression having a value between 33 and 4186. The amplitude and duration are both integers with values between 0 and 255. An amplitude of 0 generates no sound, and an amplitude of 255 generates the loudest sound the Macintosh can make. You specify the duration in units of 1/60 second. For a tone of 2048 cycles per second at 50 percent volume and 1-second duration, you would use the statement:

```
SOUND 2048,128,60
```

It's a bit inconvenient to specify the pitch of each note by frequency. Musicians prefer to specify the pitch as a note from the diatonic scale. Macintosh BASIC has a function called TONES that, given the note, returns the frequency in cycles per second. The argument that you must specify for TONES is a numeric expression with a value between −36 and 48. The values 0 through 11 correspond to the notes of the middle C octave. Adding or subtracting 12 moves up one or down one octave. The entire range of TONES is seven octaves.

THE CALC PROGRAM

Now that you have a collection of graphics and sound statements, you can use them to create a real application program that uses graphics displays.

The CALC program emulates a four-function calculator that uses scientific notation. It uses normal algebraic expressions rather than Reverse Polish Notation (RPN). Numbers of up to nineteen digits may be entered and displayed. All numbers are stored as Macintosh extended-precision real numbers.

You can enter numbers into the calculator by any of three methods—using the mouse to click the buttons on the calculator face, pressing the corresponding keys on the Macintosh keyboard, or using the Macintosh's optional numeric keypad. The buttons on the calculator face are laid out to match the numeric keypad's key locations (figure 7.16).

The CALC program does not make use of any toolbox routines. It draws a calculator face in the BASIC output window and creates its own

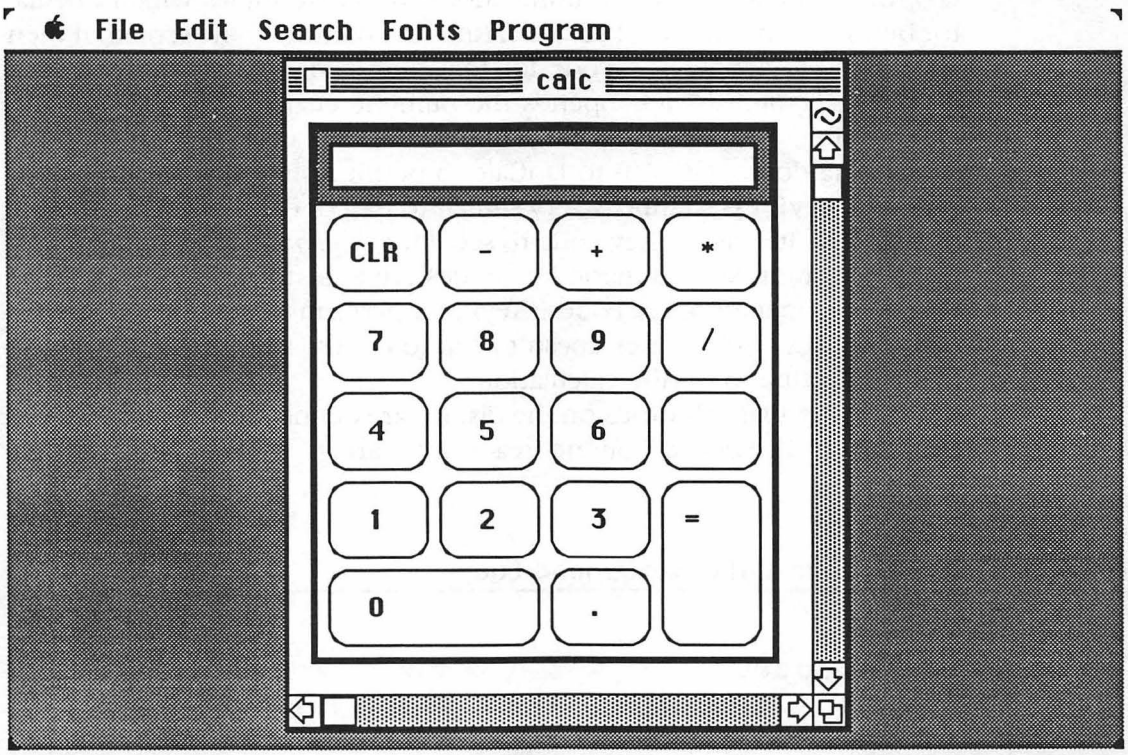


Figure 7.16 Calculator program display

buttons, using BASIC's built-in graphics statements. The program determines which button was clicked by testing the mouse position, not by using the toolbox event manager.

The calculator has a display register and an accumulator. As you enter a number, it goes into a string variable and is displayed in the calculator's display window. When you finish entering the number and press an operation key (+, −, *, /, =), the program moves the numeric value to the display register. The operation key value is saved, and the operation is performed after the next operation key is pressed.

The background loop of the program (listing 7.2) checks to see if the entry was made with the mouse button or the keyboard (by reading the keyboard buffer with the function InKey$). If the mouse button was used, the background loop uses a GOSUB to call the MousePress routine. If the entry was made with the keyboard or keypad, the program does a GOSUB to the KeyPress routine.

The MousePress routine determines whether the mouse button has been pressed or released; if it has been released, MousePress does a GOSUB to the MouseTrap routine. MouseTrap determines which calculator button (if any) was clicked and flashes the button on the screen. It then generates the proper key code for that button. If the key code is for a number key, MouseTrap appends the numeric character to the number-entry string, KeyNum$. If the key code is for an operation key, the subroutine does a GOSUB to DoCalc to perform the calculation.

The KeyPress routine is very similar to part of the MouseTrap routine (listing 7.3). It tests the key code to see if it corresponds to a numeric key or an operation key. Numeric key codes are appended to the KeyNum$ string. For operation key codes, KeyPress performs a shift of the keypad operation keys (so the user doesn't need to use the shift key) and calls the DoCalc routine to do the calculation.

The calculator buttons on the display are defined in DATA statements that the SetUpKeyPad routine reads into arrays before the DrawCalc

Listing 7.2 CALC Background Loop

```
Background:
    inmouse~=mouseb~
    if inmouse~<>oldmouse~ then gosub MousePress
    key$ = inkey$
    if key$ <> "" then gosub KeyPress
    goto Background
```

Listing 7.3 KeyPress Routine

```
KeyPress:
    nKey=asc(key$)
    if nKey > 57 then return
    rem fix the key code so you don't have to shift for -+*/
    if nKey = 28 then Key$ ="+"
    if nKey = 29 then Key$ = "*"
    if nKey = 30 then Key$ = "/"
    if nKey = 31 then key$ = ","
    nKey = asc(Key$)
    gosub FlashKey
    if (nKey <= 45) or (nKey = 47) then
        dnKey = nKey
        gosub DoCalc
        return
    endif
    if keys + 1 > 19 then
        gosub beep
        return
    endif
    if nKey <> 46 then keys = keys+1
    if KeyNum$ = "" then gosub ClearReg
    gprint Key$;
    OldKeyNum$ = KeyNum$
    KeyNum$=OldKeyNum$ & Key$
return
```

routine draws the calculator face. The button locations, sizes, and values may be changed by changing the PadData and RefData statements.

You operate the calculator just as you would any four-function calculator. Just enter numbers and press operation buttons as if you were entering an algebraic expression. Pressing the CLR (clear) key once clears the display register and the number being displayed. That erases the number you were entering without destroying any intermediate results that might still be in the accumulator. Pressing the CLR key twice in a row clears the display register and the accumulator.

You can have up to seven calculators active at any one time because Macintosh BASIC lets you run up to seven programs (or seven copies of the same program). You may find, however, that fitting seven calculator displays on the screen can be difficult. If you are entering data from the keyboard or the numeric keypad, you may shrink each calculator display until it shows just the numeric display of the calculator (figure 7.17).

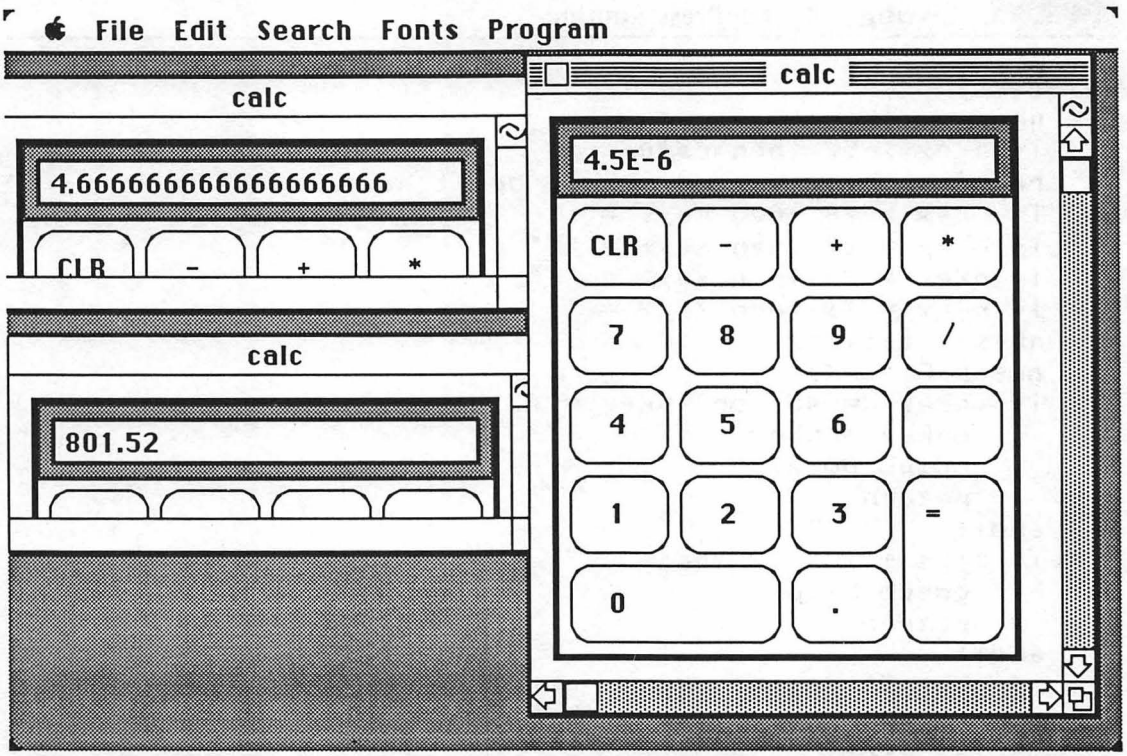


Figure 7.17 Multiple calculators

ALTERING THE CALC PROGRAM

There are several interesting things you can do with the CALC program (listing 7.4). One of the keys in the numeric keypad display on the calculator face has been left unused. You could add a square-root function on that key. Make sure you know whether you want to calculate the square root of the number in the accumulator immediately or allow another number to be entered. Another possibility would be to make the unused key a general *n*th-root function. The user would follow the square-root key with a number *n*, and the calculator would calculate the *n*th root of the number in the accumulator.

You can also make a specialized calculator with a larger face to allow more keys. It could be a financial calculator using the Macintosh BASIC financial functions ANNUITY and COMPOUND, a scientific calculator using the trigonometric and logarithmic functions, or a hexadecimal calculator that uses your own number-conversion routines.

Listing 7.4 CALC Program

```
REM calculator program

REM Calculator Key Definitions
rem the keys are numbered left to
rem right starting at the top of
rem the keypad
dim PadKey$(17)
dim PadKeyx(17)
dim PadKeyy(17)
dim PadKeyW(17)
dim PadKeyH(17)
dim KeyRef(17)
restore
set showdigits 19
KeyNum$ = ""
oldop = 0
gosub SetUpKeyPad
gosub DrawCalc
Background:
    inmouse~=mouseb~
    if inmouse~<>oldmouse~ then gosub MousePress
    key$ = inkey$
    if key$ <> "" then gosub KeyPress
    goto Background

MousePress:
    if inmouse~ then gosub MouseTrap
    oldmouse~=inmouse~
return

MouseTrap:
    mx = mouseh
    my = mousev
    mkeynum = -1
    for i = 0 to 17
        if (mx > PadKeyx(i)) and (mx < PadKeyx(i)+PadKeyW(i)) then
            if (my > PadKeyy(i)) and (my < PadKeyy(i)+PadKeyH(i)) then
                mkeynum = i
                exit
            endif
        endif
    next i
    rem test for mouse not in a key pad at all
    if mkeynum = -1 then return
    rem flash the key
    invert roundrect  PadKeyx(mkeynum),PadKeyy(mkeynum);PadKeyx(mkeynum)+PadKe
yW(mkeynum),PadKeyy(mkeynum)+PadKeyH(mkeynum) with 20,20
    nmkey = KeyRef(mkeynum)
    if (nmkey <= 45) or (nmkey = 47) then
        dnkey=nmkey
        gosub DoCalc
        goto release
    endif
    if keys + 1 > 19 then
        gosub beep
        goto release
    endif
    if nmkey <> 46 then keys = keys+1
```

Continued

Listing 7.4 *Continued*

```
    if KeyNum$ = "" then gosub ClearReg
    gprint chr$(nmkey);
    OldKeyNum$ = KeyNum$
    KeyNum$=OldKeyNum$ & chr$(nmkey)
release:
    if mouseb~ then goto release
    invert roundrect  PadKeyx(mkeynum),PadKeyy(mkeynum);Padkeyx(mkeynum)+PadKe
yW(mkeynum),PadKeyy(mkeynum)+PadKeyH(mkeynum) with 20,20
return

KeyPress:
    nkey=asc(key$)
    if nkey > 57 then return
    rem fix the key code so you don't have to shift for -+*/
    if nkey = 28 then Key$ ="+"
    if nkey = 29 then Key$ = "*"
    if nkey = 30 then Key$ = "/"
    if nkey = 31 then key$ = ","
    nkey = asc(Key$)
    gosub FlashKey
    if (nkey <= 45) or (nkey = 47) then
        dnkey = nkey
        gosub DoCalc
        return
    endif
    if keys + 1 > 19 then
        gosub beep
        return
    endif
    if nkey <> 46 then keys = keys+1
    if KeyNum$ = "" then gosub ClearReg
    gprint Key$;
    OldKeyNum$ = KeyNum$
    KeyNum$=OldKeyNum$ & Key$
return

beep:
    sound 2096,100,15
return

FlashKey:
    mkey = ASC(Key$)
    for  k = 0 to 17
        if KeyRef(k) = mkey then
            knum = k
            exit
        endif
    next k
    invert roundrect  PadKeyx(knum),PadKeyy(knum);Padkeyx(knum)+PadKeyW(knum),
PadKeyy(knum)+PadKeyH(knum) with 20,20
    invert roundrect  PadKeyx(knum),PadKeyy(knum);Padkeyx(knum)+PadKeyW(knum),
PadKeyy(knum)+PadKeyH(knum) with 20,20
return

DrawCalc:
    set pensize 3,3
    frame rect 10,10;225,260
    set pensize 1,1
```

Continued

Listing 7.4 *Continued*

```
    set pattern 3
    paint rect 13,13;222,47
    set pattern 19
    paint rect 20,20;215,40
    set pattern 0
    frame rect 20,20;215,40
    plot 13,47;222,47
    charx = 10
    chary = 22
    set font 0
    set fontsize 12
    for i = 0 to 17
)       frame roundrect PadKeyx(i),PadKeyy(i);(PadKeyx(i)+PadKeyW(i)),(PadKeyy(i
+PadKeyH(i)) with 20,20
        set penpos PadKeyx(i)+charx, PadKeyy(i)+chary
        if i = 15 then set penpos PadKeyx(i)+charx-4, PadKeyy(i)+chary
        gprint PadKey$(i)
    next i
    set penpos 25,35
return

SetUpKeyPad:
    Heigth = 36
    Width = 46
    for i = 0 to 17
        read PadKey$(i)
        PadKeyH(i) = Heigth
        PadKeyW(i) = Width
    next i
PadData:
    data  CLR, "  -" , "  + ", "  * ","  7","  8","  9","  /","  4","  5","  6
","  ","  1","  2","  3",,"  0", "  ."
    for i = 0 to 17
        read KeyRef(i)
    next i
RefData:
    data 27,45,43,42,55,56,57,47,52,53,54,44,49,50,51,3,48,46
    separationx = 5
    separationy = 5
    PadKeyH(15)=2*Heigth+separationy
    PadKeyW(16)=2*Width+separationx
    PadKey$(15) = " ="
    x1 = 13
    y1 = 47
    rowy = y1+separationy
    for row= 0 to 4
        colx = x1+separationx
        for col = 0 to 3
            n=(row*4)+col
            PadKeyy(n) = rowy
            PadKeyx(n) = colx
            colx = colx+separationx+PadKeyW(n)
            if n = 17 then exit
        next col
        rowy = rowy+separationy+Heigth
    next row
return
```

Continued

Listing 7.4 *Continued*

```
DoCalc:
  ! Here we do the actual calculation (finally)
  ! The operation code is the numeric value of the calculator
  ! key pressed
  ! the enter key is used as an enter key and as the equals key
  ! the calculator uses normal algebraic notation. to add 1 and 3
  ! you press 1, then + then 3 then enter
  ! The operation codes are:
  !   code     index    key operation
  !   43       1        +       add
  !   45       2        -       subtract
  !   42       3        *       multiply
  !   47       4        /       divide
  !   3        5        =       calculate last operation
  !   27       6        CLR clear the input register
  !                     2nd time pressed succesively, clear the accumulator
  !
  !   `the calculator has an entry register and an accumulator
  ! the accumulator stores the result of the last calculation
  ! the algorithm is :
  !       store the op code
  !       perform the previous op code on reg\ and accum\
  !       results in accum\
  ! the op code is passed in dnkey
    reg\ = val(KeyNum$)
    keys = 0
    if dnkey = 27 then
            gosub ClearOp
    else
            nclr = 0
            if oldop = 0 then
                gosub EqualOp
            else if oldop = 47 then
                    accum\ = accum\ / reg\
                else if oldop = 42 then
                        accum\ = accum\ * reg\
                    else if oldop = 45 then
                            accum\ = accum\ - reg\
                        else if oldop = 43 then
                                accum\ = accum\ + reg\
                            else if oldop = 44 then return
    oldop = dnkey
    !  RETURN key, = key and default (0) op code all do the same thing
    if (oldop = 13 or oldop = 3) then oldop = 0
    gosub DisplayAccum
    endif
return

EqualOp:
    if KeyNum$ = "" then
        accum\ = accum\ + reg\
    else
        accum\ = reg\
    endif
return
```

Continued

Listing 7.4 *Continued*

```
ClearOp:
    if nclr = 0 then
        nclr = 1
        gosub ClearReg
    else
        nclr = 0
        oldop = 0
        accum\ = 0
    endif
return

DisplayAccum:
    Gosub ClearReg
    gprint accum\;
return

ClearReg:
    set pattern 19
    paint rect 21,21;214,39
    reg\ = 0
    set penpos 25,35
    KeyNum$ = ""
return
```

CHAPTER

8 CREATING AND USING MENUS

Designing Menus

The Anatomy of a Menu

Building Menus

Using Menus

A Skeleton Menu Program

Running the Skeleton Menu Program

DESIGNING MENUS

One of the most obvious features of Macintosh applications is their use of pull-down menus to control the program. We talked about the use and design of menus in chapters 2 and 3 and saw how to create menus with Microsoft BASIC in chapter 6. In this chapter (after a brief review of good menu-design principles) we will explore in detail how programs written in languages other than BASIC create, manipulate, and use menus.

We will use Macintosh Pascal for the programming examples in this chapter. A programmer creates and uses menus by calling various routines in the menu-manager, event-manager, and window-manager sections of the toolbox. Descriptions of selected event-, menu-, and window-manager routines are in Appendix E, Toolbox Routines. The same programming techniques and toolbox calls that we will demonstrate in Macintosh Pascal could be used in other languages, such as assembler or C.

The program's menu titles appear in the menu bar at the top of the screen and are always visible (figure 8.1). When a user presses the mouse button with the cursor on a menu title, the menu appears below the title, possibly overlaying part of an existing display. When the user moves the mouse pointer down the menu, different items in the menu are highlighted (in reverse video). Releasing the mouse button with the pointer on a highlighted item selects that item from the menu.

A convention adopted by most software developers dictates that the first three menus are always the desk accessory (apple symbol), File, and Edit menus. Good menu design says that a menu title should always be in the same place on the menu bar and that each menu should always contain

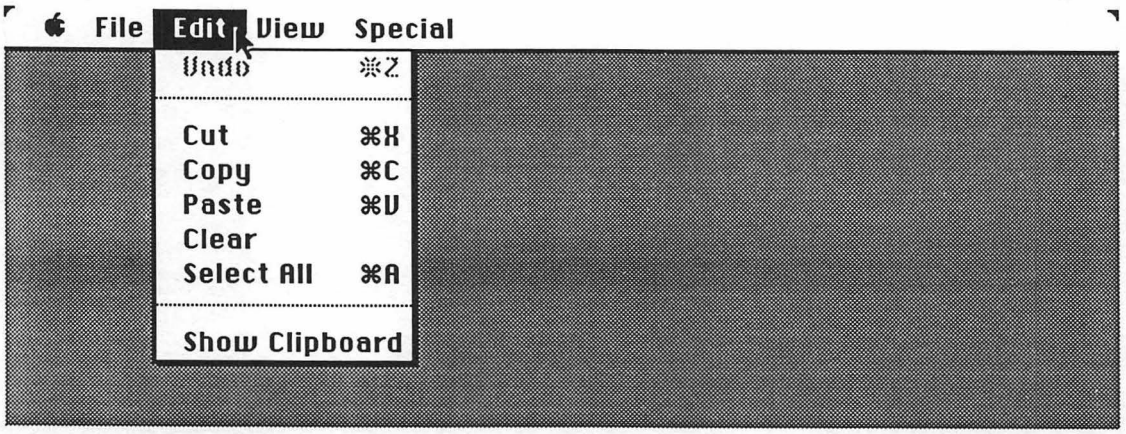


Figure 8.1 Menu bar and menu

the same items. You may disable an entire menu or individual menu items when they are not appropriate, but you should not remove them from the menu bar.

A menu that performs tasks similar to tasks in another Macintosh application should be as much like the menu of the other application as possible. Consistency in menu design is important both within a program and among separate programs.

A menu can be used to select an option or to cause the program to perform a particular task. A program may indicate a selected option or some operational state by placing a check mark beside a menu item (as most programs do in their font selection menu).

The menu bar can contain up to sixteen menu titles. Most applications will not be able to fit that many menus on the screen simply because of the limited space in the menu bar for menu titles. Menu items may be longer than the menu title. It is not unusual to see a menu extend farther to the right than the text of its title in the menu bar.

THE ANATOMY OF A MENU

You know what a menu looks like on the screen, but what does it really consist of? How does a program define a menu? A menu definition consists of a data structure and a procedure. The data structure defines the contents of the menu; the procedure draws the menu on the screen. The programmer uses toolbox procedures to manipulate the menu, put it in the menu bar, remove it, enable it, and disable it. The programmer also uses toolbox routines to find out about menu selections that the user has made.

The standard menu format that you see in almost all Macintosh applications is drawn by a procedure in the system resource file. You may create custom menu designs by supplying your own menu procedure, but only a very unusual program would require a custom menu design. You may create menus with your own text, or icons for menu items, without supplying a custom menu procedure. The standard menu procedure in the system resource file handles all of that.

A programmer defines a menu bar and the contents of its menus with a hierarchy of data structures. A data structure called the *menu list* defines a menu bar. A program may define more than one menu bar, but only one can be in use and active at any given time. A system global variable points to the menu list that defines the currently active menu bar.

A menu list consists of a set of menu records; each record defines one menu (figure 8.2). A menu record specifies the text for the menu title and the contents of the menu. A menu may have up to eighteen items. Each

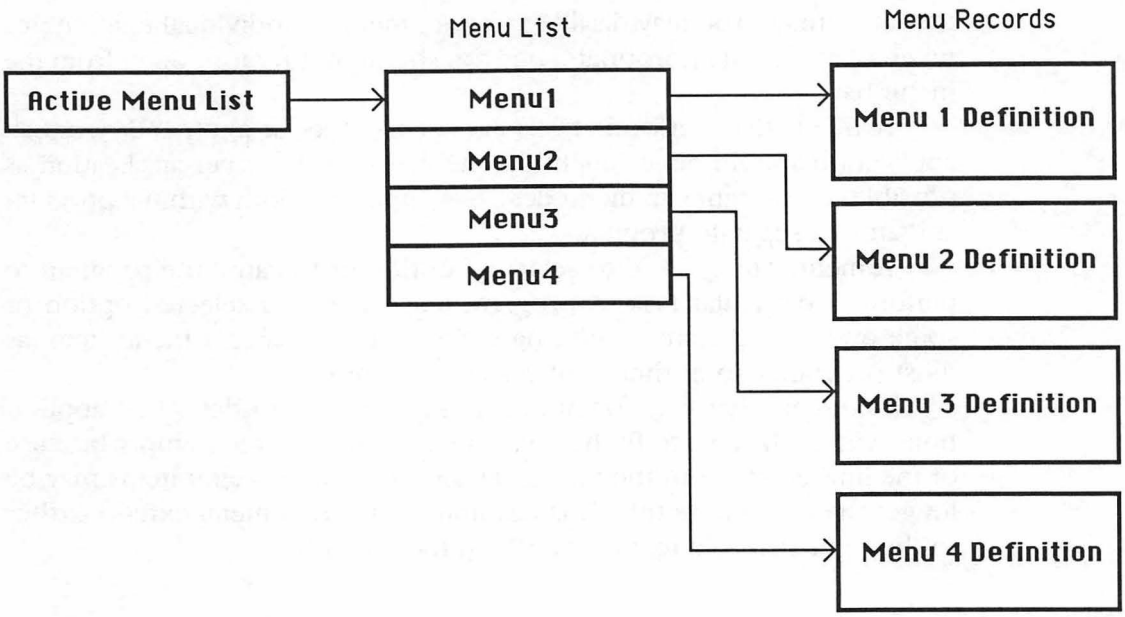


Figure 8.2 Menu data structures

item may contain text, an icon, and special characters (like a check mark). Each item may also have a command key equivalent that is displayed as part of the item. The text of a menu's title and the text of the menu's items must be in the system font (Chicago), but the programmer may specify any type style (bold, italic, outline, and so on).

Listing 8.1 shows a menu-record data structure. It contains fields for the menu ID and the width and height of the menu (in pixels). The

Listing 8.1 Menu Record

```
type
 MenuRecord = record
   menuID : integer;
   menuWidth : integer;
   menuHeight : integer;
   menuProc : handle;
   enableFlags : packed array[0..31] of boolean;
   menuData : str255
  end;
```

menuProc field is a handle to the menu definition procedure. The enableFlags array contains enable flags for the menu and menu items, and the menuData string defines the text and attributes of menu items.

To review, a menu consists of a procedure and a data structure. Most programs use the procedure provided by the Macintosh system resource file and provide their own data structures.

BUILDING MENUS

In order to display a menu bar, a program must first define a menu list, make it the currently active menu list, and redraw the menu bar. A program can define menu lists, menus, or menu items by building them with a series of calls to toolbox routines, or it may get the menu definitions from a resource file. Using resource files enables you to change a menu later by supplying a modified resource file; you don't need to recompile the program. Resource files and the resource manager are not covered in this book, so we will stick to defining menus with menu-manager subroutine calls.

It takes a series of menu-manager subroutine calls to define a set of menus, get them installed in the menu bar, and draw the menu bar on the screen. The program must first call the NewMenu routine to allocate memory space for the menu's data structure. NewMenu does not fill in menu items in the data structure—it merely fills in the menu title and returns a handle to the space it allocated for the data structure.

When you are through with that menu, you can deallocate the memory space used by its menu record and make the space available for other uses. The DisposeMenu subroutine would deallocate the space.

The example in listing 8.2 allocates space for a menu and defines its title to be Menu1. We defined the menu's ID to be the value of Menu1ID, which we previously set to 101. NewMenu returns a handle to the menu, and we store the handle in the variable called Menu1.

A handle is similar to a pointer (in fact, it is a pointer to a pointer). Programs use handles to access relocatable data structures. Menus can be accessed either by a menu handle or by a menu ID number. Some

Listing 8.2 NewMenu

```
Menu1 := Pointer(LInLineF(NewMenu, Menu1ID, 'Menu1'));
```

menu-manager routines require a handle to identify the menu; others require a menu ID. The menu's handle is related to its menu ID when the program calls the NewMenu routine.

The Pascal program example in listing 8.2 didn't really call NewMenu; it called a function named LInLineF, and one of the parameters passed to LInLineF was the trap number of the routine NewMenu. If you look through the Macintosh Pascal manuals and the *Inside Macintosh* documentation, you won't find any mention of LInLineF. The documentation for LInLineF and a whole set of similar Pascal functions and procedures is in a text file on your Macintosh Pascal disk.

The InLine functions and procedures allow you to call a procedure or function in the Macintosh ROM, whether Pascal knows about it or not. The last letter in the InLine function or procedure's name indicates whether it calls a function or a procedure. If it calls a function, the first letter of the name identifies the function type (the data type of the value returned by the function).

When you use InLine procedures and functions, Macintosh Pascal turns off all type checking. A type error, such as assigning a pointer value to an integer variable, results in a system crash, not a Pascal error message. That circumstance, coupled with the fact that InLine procedures and functions can access critical toolbox, operating-system, and device-driver routines, means that you must be very, very careful when using InLine. Specifying parameters incorrectly, calling the wrong routine, or specifying the wrong trap number can produce catastrophic results, such as crashing the program or destroying data on any disks that are in the disk drives.

InLine functions and procedures include:

InLineP	Procedure
LInLineF	Function—returns a long integer
WInLineF	Function—returns a word (integer)
BInLineF	Function—returns a Boolean value

You cannot single-step a program that has InLine statements. You should always save your source text before executing the program because a program error will not get you back to the interpreter; it will result in a system error. You can rarely recover from a system error. Usually, you must boot the system—press the reset button (the rearmost button of the programmer's switch) and then press the boot button (the frontmost button of the programmer's switch). If you don't have the programmer's switch installed, turn the power off and back on again.

The program must supply a list of parameters to the InLine procedure or function. The first parameter is the trap number of the ROM routine you want to access. In our examples, we don't put the trap number in the

InLine call. Instead, we use a constant named after the routine we are calling. We set the constants equal to the trap values of their routines, at the beginning of the program where global constants are defined. (The trap numbers for toolbox routines we have used can be found in Appendix E, Toolbox Routines.)

The remaining parameters are the parameters that the ROM routine expects to be passed to it. A trap number is used as a vector to the routine we want to call. If the routine is replaced by an updated RAM routine, we use the same trap number to access it. The system has changed the vector at the trap address, however, so we actually get the new routine instead of the old one.

A program can fill in a menu's data structure by a single call or a series of calls to the AppendMenu subroutine. AppendMenu appends items to the menu. An entire menu—including all of its items—can be defined with a single call to AppendMenu, or you may define the menu and each item with separate calls to AppendMenu.

In listing 8.3, we are defining four menus. We define the first with a call to AppendMenu for each menu item. For each of the other menus, we define all of the items with a single call to AppendMenu. When we define multiple items with a single AppendMenu call, we separate items with a semicolon.

Note that in Menus 3 and 4 there are some extra characters in the menu-item text. These are *metacharacters*; they define various attributes of menu items, as follows:

Listing 8.3 AppendMenu

```
Menu1 := Pointer(LInLineF(NewMenu, Menu1ID, 'Menu1'));
InLineP(AppendMenu, Menu1, 'Item1');
InLineP(AppendMenu, Menu1, 'Item2');
InLineP(AppendMenu, Menu1, 'Item3');
InLineP(AppendMenu, Menu1, 'Item4');
InLineP(AppendMenu, Menu1, 'Exit');
Menu2 := Pointer(LInLineF(NewMenu, Menu2ID, 'Menu2'));
InLineP(AppendMenu, Menu2, 'Item1;Item2;Item3;Item4');
Menu3 := Pointer(LInLineF(NewMenu, Menu3ID, 'Menu3'));
InLineP(AppendMenu, Menu3, '(Item1/A;(Item2/B;(Item3/C;(Item4/D');
Menu4 := Pointer(LInLineF(NewMenu, Menu4ID, 'Menu4'));
InLineP(AppendMenu, Menu4, 'Item1/E;Item2/F;Item3/G;Item4/H');
```

Metacharacter	*Attribute*
^	Icon
!	Check mark
<	Type style
/	Keyboard command character equivalent
(	Disable item

The icon metacharacter is followed by a single decimal digit that is the icon number. The icon number identifies an icon defined in a resource file.

The type-style item must be followed by a letter that indicates the type style for its menu item. The menu-item type styles are listed below:

Metacharacters	*Type Style*
<B	Boldface
<I	Italic
<U	Underline
<O	Outline
<S	Shadow

Programs use the type style of a menu item to indicate some special information about the item. A good example is the font size menu in many applications (figure 8.3).

Note that the 9- and 12-point font sizes are shown in the outline type style, while the others are displayed in a plain type style. This application (MacWrite) uses the outline type style to indicate that a particular font size is on the system disk. The plain type style is used for font sizes that are not on the system disk and must be approximated by scaling another size.

We noted in chapters 2 and 3 that good program and menu design sometimes requires that users be able to execute frequently used commands by typing on the keyboard rather than by pulling down a menu. You define the keyboard command character for a menu item with the slash (/) character followed by the command character. In our example, the items in Menus 3 and 4 have keyboard command-character equivalents. Typing F while holding down the command key performs the same function as selecting Item 2 from Menu 4.

When the system displays the menu, it shows the keyboard command characters to the right of each menu item, along with the symbol for the command key (figure 8.4).

You can use menu-manager subroutine calls to enable or disable menu items. Or you can use the disable-item metacharacter in the menu-item definition to specify that an item be disabled when the menu is first

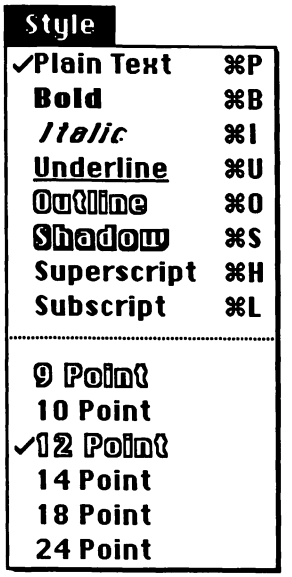


Figure 8.3 Font size menu

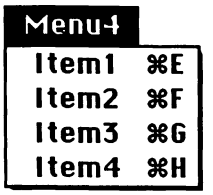


Figure 8.4 Menu 4

drawn. Note the AppendMenu statement in listing 8.3. Each item is preceded by the disable-item metacharacter (the open parenthesis). When the system first displays Menu 3, all of its items will be disabled and will appear in the dimmed type style (figure 8.5). Note that the command key symbol looks a little strange when it is dimmed. This is an artifact of the way the system shows dimmed icons. Some other symbols and icons also have peculiar appearances when dimmed.

Now we're ready to put the menus we just defined into the menu list that defines the menu bar. We follow the menu definition (NewMenu and AppendMenu) with a call to InsertMenu. When we give InsertMenu the menu's handle and menu ID number, it inserts the menu into the menu list.



Figure 8.5 Menu 3

Now the menu list defines a menu bar that we would like to make active and draw on the screen. To do this, we can follow the menu definitions with a call to the DrawMenuBar procedure. Suppose, though, that we want to be able to restore the old menu bar when our program is completed. If we use the GetMenuBar subroutine to get a handle to the old menu list before we change menu bars, we can save the old menu bar and restore it later.

In our menu-definition program, we did store the handle to the old menu bar first (listing 8.4). Right after that, we called ClearMenuBar to clear the current menu list. If we hadn't cleared the menu list before our calls to AppendMenu, we would just have added our new menus to the old menu bar, instead of creating a new menu bar. ClearMenuBar did not destroy the old menu list or menu definitions; it just created a fresh menu list. When we are ready to exit the program, we will call SetMenuBar and pass it the variable OldMenuBar, which has a pointer to the old menu list. A call to DrawMenuBar after that will restore the old menu bar and menu list.

If this were an assembly language or C program, we would need to call InitMenus before defining any menus. It initializes the menu manager's data structures and system globals used for menu operations. In this case, we didn't need to call InitMenus because the Pascal interpreter did the initialization for us.

When we finish with the menus we have defined, we might want to deallocate the memory space used by their data structures. We can do that by calling DisposeMenu for each menu record we want to deallocate. Our menu-manager calls at the end of our program will then look like listing 8.5.

The menu-manager routines we have used so far are sufficient to build menu data structures that define menus, but there are other routines that can manipulate these data structures. What follows is a brief list of the menu-manager routines for defining and manipulating menus. You can find more information about these and other toolbox routines in Appendix E, Toolbox Routines.

Listing 8.4 Drawing the New Menu Bar

```
{Save pointer to the old menu bar so we can restore it later }
OldMenuBar := Pointer(LInLineF(GetMenuBar));
{Define New Menus}
InLineP(ClearMenuBar);
Menu1 := Pointer(LInLineF(NewMenu, Menu1ID, 'Menu1'));
InLineP(AppendMenu, Menu1, 'Item1');
InLineP(AppendMenu, Menu1, 'Item2');
InLineP(AppendMenu, Menu1, 'Item3');
InLineP(AppendMenu, Menu1, 'Item4');
InLineP(AppendMenu, Menu1, 'Exit');
Menu2 := Pointer(LInLineF(NewMenu, Menu2ID, 'Menu2'));
InLineP(AppendMenu, Menu2, 'Item1;Item2;Item3;Item4');
Menu3 := Pointer(LInLineF(NewMenu, Menu3ID, 'Menu3'));
InLineP(AppendMenu, Menu3, '(Item1/A;(Item2/B;(Item3/C;(Item4/D');
Menu4 := Pointer(LInLineF(NewMenu, Menu4ID, 'Menu4'));
InLineP(AppendMenu, Menu4, 'Item1/E;Item2/F;Item3/G;Item4/H');
InLineP(InsertMenu, Menu1, 0);
InLineP(InsertMenu, Menu2, 0);
InLineP(InsertMenu, Menu3, 0);
InLineP(InsertMenu, Menu4, 0);
{Draw the new menu bar}
InLineP(DrawMenuBar);
```

Listing 8.5 Close of Menu Example Program

```
InLineP(SetMenuBar, OldMenuBar);
InLineP(DrawMenuBar);
InLineP(DisposeMenu, Menu1);
InLineP(DisposeMenu, Menu2);
InLineP(DisposeMenu, Menu3);
InLineP(DisposeMenu, Menu4);
```

Initializing and Building Menus

InitMenus	Initialize toolbox menu data structures
NewMenu	Create a new menu
DisposeMenu	Give up a menu's storage space
AppendMenu	Add a menu to the menu bar

Building the Menu List (Menu Bar)

InsertMenu	Insert a menu in the menu bar
DrawMenuBar	Draw the menu bar
DeleteMenu	Delete a menu from the menu bar
ClearMenuBar	Delete all menus from the menu bar
GetMenuBar	Make a copy of the menu list
SetMenuBar	Make the menu bar (menu list) the current menu bar

Selecting Menu Items

MenuSelect	Pull down and highlight menus tracking the mouse
MenuKey	Select a menu item based on a command key
HiliteMenu	Highlight the menu's title

Manipulating Menus and Menu Lists

SetItem	Set a menu item's text
GetItem	Get a menu item's text
DisableItem	Disable a menu item
EnableItem	Enable a menu item
CheckItem	Check a menu item
SetItemIcon	Set a menu item's icon
GetItemIcon	Get a menu item's icon
SetItemStyle	Set a menu item's style
GetItemStyle	Get a menu item's style
SetItemMark	Set a menu item's mark character
GetItemMark	Get a menu item's mark character
SetMenuFlash	Set a menu item's flash period
CountMItems	Get the number of items in a menu
GetMHandle	Get a handle to the menu
FlashMenuBar	Flash a menu's title in the menu bar

USING MENUS

Compared to the way we define menus in Microsoft BASIC, the process we have just gone through to define menus in Macintosh Pascal seems com-

plicated. Actually, we've just done the easy part. Using the menus is a little more complex. We must call a window-manager routine and a number of event-manager and menu-manager routines in order to find out which menu item, if any, the user has selected.

Our program should be organized so that it waits in a background loop until the user causes an event. An event could be the user's clicking the mouse or typing a key on the keyboard. What we do with the event depends on the current state of the program—whether it has any windows open, is accepting keyboard input, or has any menus enabled. Our program will usually be organized to interpret the event (keyboard or mouse) and then pass it to the part of the program that is waiting for it. If the program does more than one thing with a given event type, the program must maintain a state variable, or *active task ID* variable, so it will know which of its procedures should handle the event.

An example would be a program that has a number of menus active and two windows open. The keyboard event could be a command key that selects a menu item, or it could be keyboard input for the active window (the program interprets the character from the keyboard). If it's a command key, the program calls a menu-manager routine to select the proper menu item. If the keyboard event is keyboard input, the program checks its state variable to see which window is active and what it should be doing in the window. The program then passes the keyboard character to the subroutine that handles input for that window.

While it might seem that the program is interrupt-driven, it isn't. It is actually polling the event manager to find out what events have occurred. The event manager does not interrupt the program when an event occurs.

There are many more event types than the ones we will discuss here. Some are related to controlling windows: activating, deactivating, changing window size, scrolling, and so on. Others are events generated by peripherals like the disk drives, the printer port, and the communications port. For now, we will restrict our interest to events related to menus.

In the background loop of our program, we need to call the event-manager routine GetNextEvent to see what kind of event (if any) has occurred. Once we have an event, we determine whether it is a mouse event or a keyboard event. We have separate procedures to handle each of these event types.

The operating system and its interrupt handlers notify the event manager of the events caused by the user or I/O devices. The event manager queues events in priority order, as shown in the following list, and passes one event to the application program each time the program calls GetNextEvent.

1 Activate-window events

2 Mouse, keyboard, disk-insert, abort, network, and I/O-driver events, plus events defined by application programs

3 Auto-key events

4 Window update events

5 Null event

The application program can determine which types of events it wants to receive by setting an event-type mask in the event manager. The program calls the SetEventMask routine in the event manager and passes it an event mask. Figure 8.6 shows the event mask and event types. The event types used in controlling menus are shown in boldface. Note that the data type of the event mask is integer. Figure 8.6 shows how the bits in the integer are assigned to event types in the mask.

A call to GetNextEvent gets us an event record—a data structure that describes the event in detail (listing 8.6). The What field in the event record tells us the event type. The Message field contains data about the event (figure 8.7). For keyboard events, the Message field contains the key code and character code for the key that caused the event.

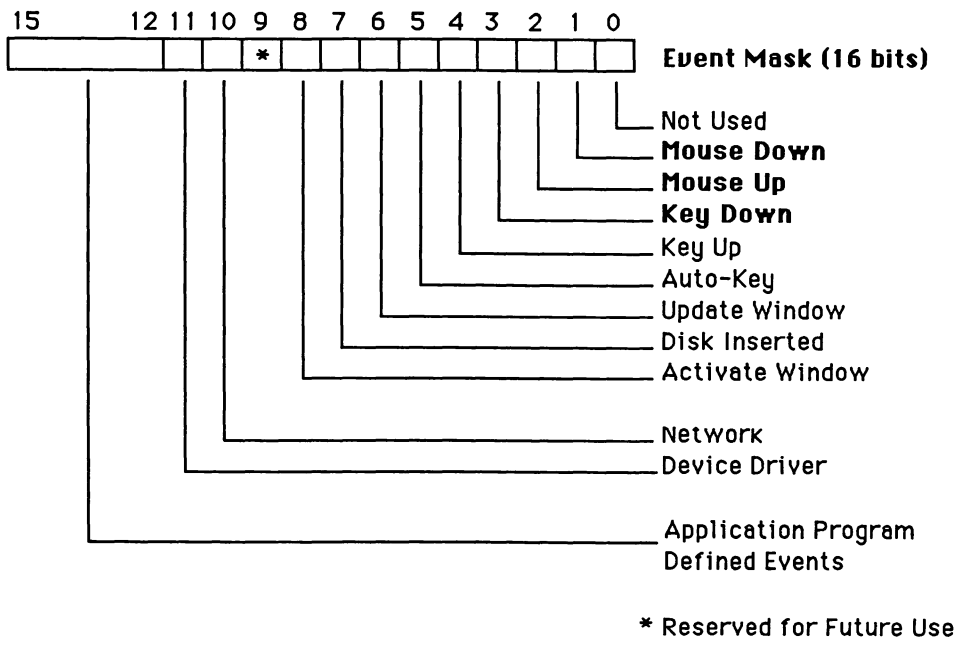


Figure 8.6 Event Mask

Listing 8.6 Event Record

```
EventRecord = record
  What : Integer;
  Message : LongInt;
  When : LongInt;
  Where : LongInt;
  Modifiers : Integer;
 end;
```

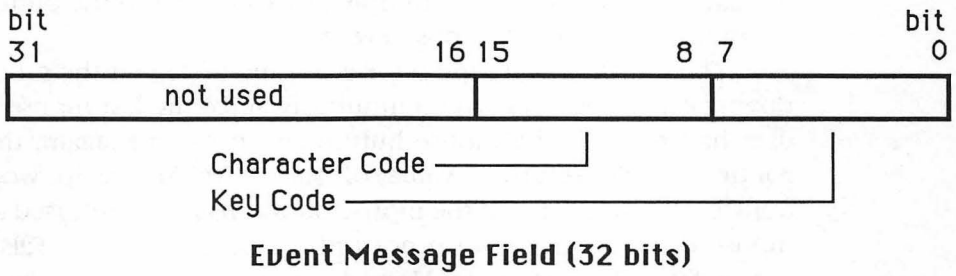


Figure 8.7 Keyboard-event Message field

The Where field contains the mouse position at the time of the event. The When field contains the time of the event in number of clock ticks (1/60 second each) since the system was started. The Modifier field contains additional information about the event, such as the states of the mouse button and the command, shift, caps lock, and option keys:

Bit	*Definition*
0	= 1, Activate
	= 0, Deactivate
1	= 1, Application window activated, system window deactivated
7	Mouse button down
8	Command key down
9	Shift key down
10	Caps lock key down
11	Option key down

A number of event-manager routines will be useful to us in controlling menus and tracking mouse movement. We have already discussed the GetNextEvent routine; it gets the next event from the event queue but gets only events that we have not masked off. We can find out what the next event in the queue is with the EventAvail routine. It returns the event type and event record of the next event in the queue, but it leaves the event in the queue. GetNextEvent does the same thing but removes the event from the queue.

At the beginning of any program, we should call the FlushEvents routine. It clears the event queue and thus discards any events that occurred before our program executed.

The Button routine returns the state of the mouse button. It returns the current state, regardless of how many changes in the button state have occurred since the last mouse event.

The StillDown routine returns a value of True if the mouse button is down and has been down continuously since the last mouse event. If the user had released the mouse button and pressed it again, the StillDown routine would return a value of False. WaitMouseUp works like the StillDown routine, but if the mouse button has been released since the last mouse event, WaitMouseUp not only returns a value of False, it also removes from the queue any MouseUp events that have occurred since the last MouseDown event.

Event- and Window-Manager Routines for Controlling Menus

```
InLineP(SetEventMask, EventMask: Integer)
InLineP(FlushEvents, EventMask: Integer, StopMask: Integer)
BInLineF(GetNextEvent, EventMask: Integer, var Event: EventRecord)
BInLineF(EventAvail, EventMask: Integer, var Event: EventRecord)
BInLineF(Button)
BInLineF(StillDown)
BInLineF(WaitMouseUp)
WInLineF(FindWindow)
```

If we get a keyboard event and we determine that the command key was pressed, we call the menu-manager routine MenuKey. It determines which menu and item have been selected with the command key and returns the menu and item ID numbers.

If we get a mouse event, we still have some work to do. We must first call the window-manager routine FindWindow to determine if the mouse pointer is in the menu bar. If it is, we call the menu-manager routine MenuSelect, which pulls down the menu and tracks mouse movement. If

the user releases the mouse button on a menu item, the routine returns the selected menu and item ID numbers.

Whether it's a mouse or keyboard event, we now know which menu and item were selected, and we can execute a procedure to do whatever that menu item calls for. When we're all done, we call the menu-manager routine HiLiteMenu. The MenuSelect routine left the menu title in the menu bar in reverse video (white letters on black) to indicate that the menu item was still selected while the program was running. Our call to HiLiteMenu returns the menu title to normal video, notifying the user that the program has finished the processing started by the menu item selection.

To review, for both mouse and keyboard events, we identify which menu item has been selected, execute the procedure to do that function, and then return the menu title to normal video. For command key events, we call the MenuKey routine to identify which menu item was selected.

For mouse events, we get the mouse position from the event-manager routine FindNextEvent, and call FindWindow to see if the cursor is in the menu bar. If the cursor is in the menu bar, we call MenuSelect and pass it the mouse location. MenuSelect tracks the mouse from that point, pulling down menus and highlighting menu items until the user releases the mouse button. MenuSelect returns the menu and item number of the selected item. We continue our processing, just as in the command key case, and call HiLiteMenu when done.

A SKELETON MENU PROGRAM

Even though it looks like a lot of code, our skeleton menu program is really not at all complicated when you look at what each procedure is doing (listing 8.7). Let's see how the program sets up menus and handles menu events. We have written the program to display four menus with the menu names Menu1, Menu2, Menu3, and Menu4. When the program initializes the menus, all of the items in Menu 3 are disabled, and the items in the other menus are enabled.

We need to assign some actions to the selected menu items, so the items in Menu 2 enable and disable the corresponding items in Menu 3. Each of the items in Menu 4 may have a check mark to indicate selection. Selecting a Menu 4 item toggles its check mark. Menu 1 and Menu 3 items merely write a line to the Text window with the menu and item names. However, Menu 1 does have one additional item that does something useful. The Exit item causes the program to end in an orderly fashion,

Listing 8.7 Skeleton Menu Program

```
program Menus;
 const
{ToolBox Trap Numbers}
   {Window Manager Traps}
  FindWindow = $A92C;
  {ToolBox Utility Traps}
  BitAnd = $A858;
  HiWord = $A86A;
  LoWord = $A86B;
  SysBeep = $A9C8;
  {Menu Manager Traps}
  EnableItem = $A939;
  DisableItem = $A93A;
  CheckItem = $A945;
  GetMenuBar = $A93B;
  ClearMenuBar = $A934;
  NewMenu = $A931;
  AppendMenu = $A933;
  InsertMenu = $A935;
  DrawMenuBar = $A937;
  SetMenuBar = $A93C;
  DisposeMenu = $A932;
  MenuSelect = $A93D;
  MenuKey = $A93E;
  HiLiteMenu = $A938;
  {Quickdraw Traps}
  InitCursor = $A850;
  {Event Manager Traps}
  GetNextEvent = $A970;
  StillDown = $A973;
{Event Manager Constants}
  EventMask = $000F;
  MouseDown = 1;
  KeyDown = 3;
  CmdKey = 256;
{Other Constants}
  Menu1Id = 101;
  Menu2Id = 102;
  Menu3Id = 103;
```

Continued

Listing 8.7 *Continued*

```
 Menu4Id = 104;
 M1Item1 = 1;
 M1Item2 = 2;
 M1Item3 = 3;
 M1Item4 = 4;
 M1ExitItem = 5;
 M2Item1 = 1;
 M2Item2 = 2;
 M2Item3 = 3;
 M2Item4 = 4;
 M3Item1 = 1;
 M3Item2 = 2;
 M3Item3 = 3;
 M3Item4 = 4;
 M4Item1 = 1;
 M4Item2 = 2;
 M4Item3 = 3;
 M4Item4 = 4;
type
 Ptr = ^LongInt;
 Handle = ^Ptr;
 WindowRecord = array[1..78] of Integer;
 WindowPtr = ^WindowRecord;
 Rectangle = array[1..4] of Integer;
 EventRecord = record
    What : Integer;
    Message : LongInt;
    When : LongInt;
    Where : LongInt;
    Modifiers : Integer;
   end;
var
 OldMenuBar, Menu1, Menu2, Menu3, Menu4 : Handle;
 EventInfo : EventRecord;
 Abort : Boolean;
 SelectedMenuID, SelectedItemID : Integer;
 M3Toggle : array[1..4] of boolean;
 M4Toggle : array[1..4] of boolean;
```

Continued

Listing 8.7 *Continued*

```
procedure Initialize;
  const
    SystemFont = 0;
    ApplicationFont = 1;
    NewYork = 2;
    Geneva = 3;
    Monaco = 4;
  var
    index : Integer;
begin
  for Index := 1 to 4 do
    begin
      M3Toggle[Index] := False;
      M4Toggle[Index] := False;
    end;
  Abort := False;
  InLineP(InitCursor);
  FlushEvents(EventMask, 0);
{Save pointer to the old menu bar so we can restore it later }
  OldMenuBar := Pointer(LInLineF(GetMenuBar));
{Define New Menus}
  InLineP(ClearMenuBar);
  Menu1 := Pointer(LInLineF(NewMenu, Menu1ID, 'Menu1'));
  InLineP(AppendMenu, Menu1, 'Item1');
  InLineP(AppendMenu, Menu1, 'Item2');
  InLineP(AppendMenu, Menu1, 'Item3');
  InLineP(AppendMenu, Menu1, 'Item4');
  InLineP(AppendMenu, Menu1, 'Exit');
  Menu2 := Pointer(LInLineF(NewMenu, Menu2ID, 'Menu2'));
  InLineP(AppendMenu, Menu2, 'Item1;Item2;Item3;Item4');
  Menu3 := Pointer(LInLineF(NewMenu, Menu3ID, 'Menu3'));
  InLineP(AppendMenu, Menu3, '(Item1/A;(Item2/B;(Item3/C;(Item4/D');
  Menu4 := Pointer(LInLineF(NewMenu, Menu4ID, 'Menu4'));
  InLineP(AppendMenu, Menu4, 'Item1/E;Item2/F;Item3/G;Item4/H');
  InLineP(InsertMenu, Menu1, 0);
  InLineP(InsertMenu, Menu2, 0);
  InLineP(InsertMenu, Menu3, 0);
  InLineP(InsertMenu, Menu4, 0);
{Draw the new menu bar}
```

Continued

Listing 8.7 *Continued*

```
 InLineP(DrawMenuBar);
end;

procedure DoMenu1;
begin
 case SelectedItemID of
  M1Item1 :
   writeln('Menu 1 Item 1');
  M1Item2 :
   writeln('Menu 1 Item 2');
  M1Item3 :
   writeln('Menu 1 Item 3');
  M1Item4 :
   writeln('Menu 1 Item 4');
  M1ExitItem :
   Abort := True;
  otherwise
 end;
end;

procedure DoMenu2;
begin
 M3Toggle[SelectedItemID] := not M3Toggle[SelectedItemID];
 if M3Toggle[SelectedItemID] then
  InLineP(EnableItem, Menu3, SelectedItemID)
 else
  InLineP(DisableItem, Menu3, SelectedItemID);
end;

procedure DoMenu3;
begin
 case SelectedItemID of
  M3Item1 :
   writeln('Menu 3 Item 1');
  M3Item2 :
   writeln('Menu 3 Item 2');
  M3Item3 :
   writeln('Menu 3 Item 3');
  M3Item4 :
```

Continued

Listing 8.7 *Continued*

```
    writeln('Menu 3 Item 4');
   otherwise
  end;
end;

procedure DoMenu4;
begin
  M4Toggle[SelectedItemID] := not M4Toggle[SelectedItemID];
  InLineP(CheckItem, Menu4, SelectedItemID,
       M4Toggle[SelectedItemID]);
end;

procedure HandleMenu;
  var
    SelectedMenu : LongInt;
    Ch : Char;
begin
  EventInfo.Message := LInLineF(BitAnd, EventInfo.Message, $FF + 0);
  Ch := Chr(EventInfo.Message);
  if EventInfo.What = MouseDown then
    SelectedMenu := LInLineF(MenuSelect, EventInfo.Where)
  else
    SelectedMenu := LInLineF(MenuKey, Ch);
  SelectedMenuID := WInLineF(HiWord, SelectedMenu);
  SelectedItemID := WInLineF(LoWord, SelectedMenu);
  case SelectedMenuID of
    Menu1ID :
      DoMenu1;
    Menu2ID :
      DoMenu2;
    Menu3ID :
      DoMenu3;
    Menu4ID :
      DoMenu4;
    otherwise
  end;
  InLineP(HiLiteMenu, 0);
end;
```

Continued

Listing 8.7 *Continued*

```
 procedure TrackMouse;
  const
   InMenuBar = 1;
  var
   AWindow : WindowPtr;
 begin
  if WInLineF(FindWindow, EventInfo.Where, @AWindow) = InMenuBar
       then
   HandleMenu
  else
   InLineP(SysBeep, 5)
 end;

 procedure HandleKey;
 begin
  if LInLineF(BitAnd, EventInfo.Modifiers + 0, CmdKey + 0) = CmdKey
       then
   HandleMenu
  else
   InLineP(SysBeep, 5)
 end;

{The Outer Loop}
begin
 initialize;
 begin
  repeat
  {wait for an event, then handle it}
   if BInLineF(GetNextEvent, EventMask, @EventInfo) then
    case EventInfo.What of
     MouseDown :
      TrackMouse;
     KeyDown :
      HandleKey;
     otherwise
    end
  until Abort;
  InLineP(SetMenuBar, OldMenuBar);
  InLineP(DrawMenuBar);
```

Continued

Listing 8.7 *Continued*

```
    InLineP(DisposeMenu, Menu1);
    InLineP(DisposeMenu, Menu2);
    InLineP(DisposeMenu, Menu3);
    InLineP(DisposeMenu, Menu4);
  end
end.
```

releasing the space allocated for menu data structures and restoring the old menu bar.

All of the toolbox trap numbers that we use are defined in the constant-definition section of the main program. The event mask, event type, Menu ID, and Item ID constants are also defined there.

The initialization section is pretty much the same as the examples we used earlier in our discussion about how to define menus. In addition to defining the menus, the initialization section flushes preexisting events from the event queue and initializes the toggle variable arrays for Menus 3 and 4. Note that the initialization section saves the handle to the old menu bar so we can restore it at the end of the program.

The outer loop of the program repeatedly checks to see if an event has occurred and, if so, handles the event. We are checking for only two event types: a (mouse button) MouseDown event or a (keyboard) KeyDown event. If we get a MouseDown event, we call the TrackMouse routine to handle it. If we get a KeyDown or auto-key event, we call the HandleKey routine.

Take a look at the HandleKey routine in the listing. The only keyboard keys that interest us are the command keys. If the key was a command key, we call HandleMenu to take care of it. If not, we call SysBeep to beep the speaker.

The TrackMouse routine isn't any more complicated. We first find out if the mouse cursor is in the menu bar. If it is, we call HandleMenu; if it's not, SysBeep. We find out where the cursor is by calling the window-manager routine FindWindow. We pass it the mouse location that we got in the event record, and FindWindow returns a value that tells us if the mouse is in the menu bar.

With HandleMenu, the first thing we want to do is find out which menu was selected. If the key that got us to HandleMenu was a command key, we call the menu-manager routine MenuKey. It maps the command key to the proper menu and item, selects the menu item, and highlights the

menu title in the menu bar. MenuKey returns the ID numbers of the selected menu and item.

If we are handling a MouseDown event, we call the menu-manager routine MenuSelect. It pulls down the selected menu and tracks the mouse movement, highlighting menu items as the mouse drags over them. The MenuSelect routine doesn't return control to us until the user releases the mouse button. MenuSelect returns the menu and item ID numbers of the selected menu and item.

Once we know which menu the user has selected, we just do a case statement to call a different routine (DoMenu1 through DoMenu4) for each menu. After the DoMenu routine returns, we call HiLiteMenu to restore the selected menu's title in the menu bar to normal video. It's good program design practice to always leave a selected menu highlighted while the program is performing the function that the user has selected. That way the user knows what the system is doing.

RUNNING THE SKELETON MENU PROGRAM

To start the program we choose Go from the Run menu. When the program goes through its initialization section, it replaces the existing menu bar with its own. Once the new menu bar is drawn, we pull down Menu 3 and see that all of its items are disabled (figure 8.8).

Now we pull down Menu 2 and select Item 1. We pull it down again and select Item 3. When we pull down Menu 3, we see that Items 1 and 3 have been enabled (figure 8.9). Selecting an item from Menu 2 enables or disables the corresponding item in Menu 3.

Looking at the DoMenu2 procedure, we can see that it uses an array of Boolean variables, one for each menu item, to see if a Menu 3 item is enabled or disabled. The procedure first changes the state of the Boolean variable for the selected menu item. If the resulting variable value is True,

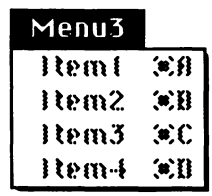


Figure 8.8 Menu 3 with all items disabled

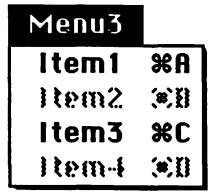


Figure 8.9 Menu 3 with Items 1 and 3 enabled

it calls the menu-manager routine EnableItem to enable the corresponding item in Menu 3. If the variable value is False, it calls DisableItem.

We now select Item 1 from Menu 1, and Item 3 from Menu 3, and we see the resulting output in the Text window. When we select items from Menu 1 or Menu 3, the program writes a line in the Text window to identify which menus and items were selected (figure 8.10).

Let's try doing the same thing with a command key instead of a menu selection. When we hold down the command key and type the letter A, the result is that Menu 3 Item 1 is selected. We see the menu title highlighted briefly, and the program writes another line in the Text window, identifying Item 1 from Menu 3.

The DoMenu1 and DoMenu3 procedures are nearly identical. They both do a case statement on the selected item ID number. For each case (item), they execute one line of code that writes a line of text in the Text window.

Let's take a look at Menu 4 (figure 8.11). We have written the program to toggle a check mark in a Menu 4 item when we select it. Pulling down Menu 4, we see that none of its items have check marks yet.

Now we pull down Menu 4 and select Item 1. Next we hold down the command key and type the letter H, selecting Item 4. If we pull down Menu 4 now, we see that Items 1 and 4 have check marks (figure 8.12). Programs use check marks to indicate menu selections that set program flags or to select items from a list (like type fonts).

Looking at the DoMenu4 procedure, we see that it uses the M4Toggle array. The array contains a Boolean value (True or False) for each item in the menu. The procedure changes the state of that Boolean value for the selected item and then calls the menu-manager routine CheckItem. CheckItem uses the Boolean value in M4Toggle to determine whether to put a check mark next to the menu item or to remove a check mark. That's all there is to setting and removing check marks.

To exit the program, we pull down Menu 1 and select Exit. The program returns the old menu bar to the screen and ends. Selecting Exit

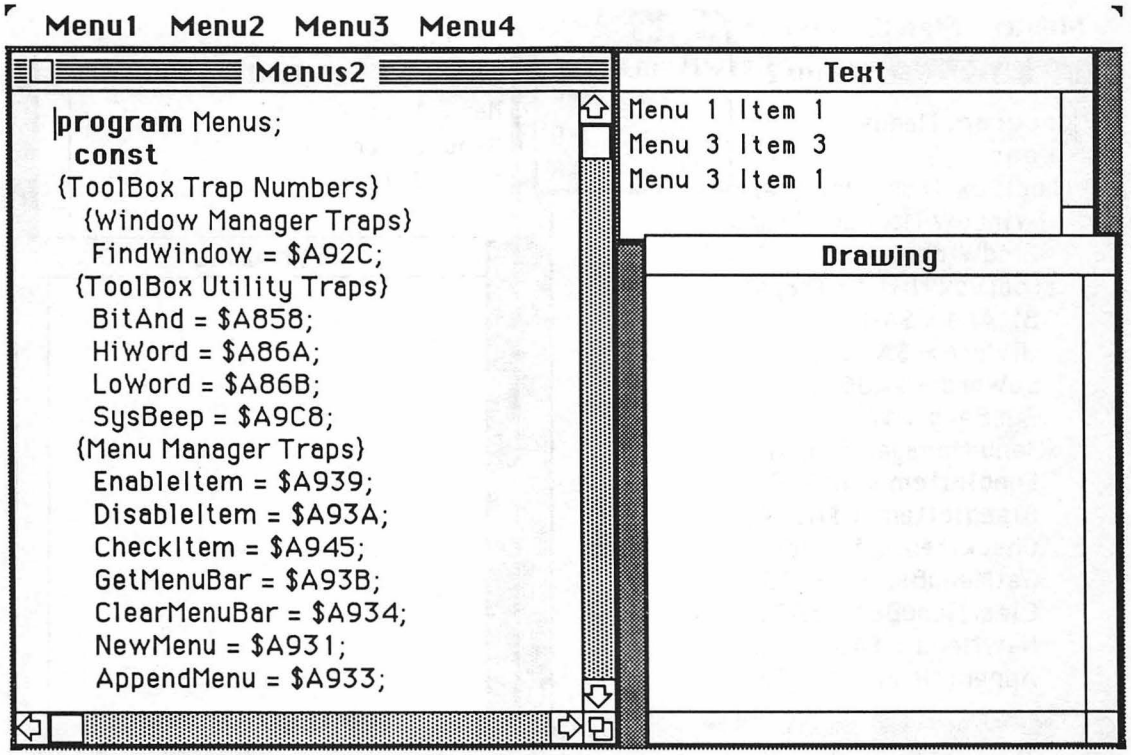


Figure 8.10 Menu selections traced in the Text window

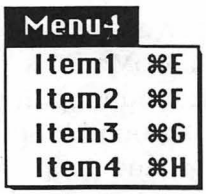


Figure 8.11 Menu 4 with no items checked

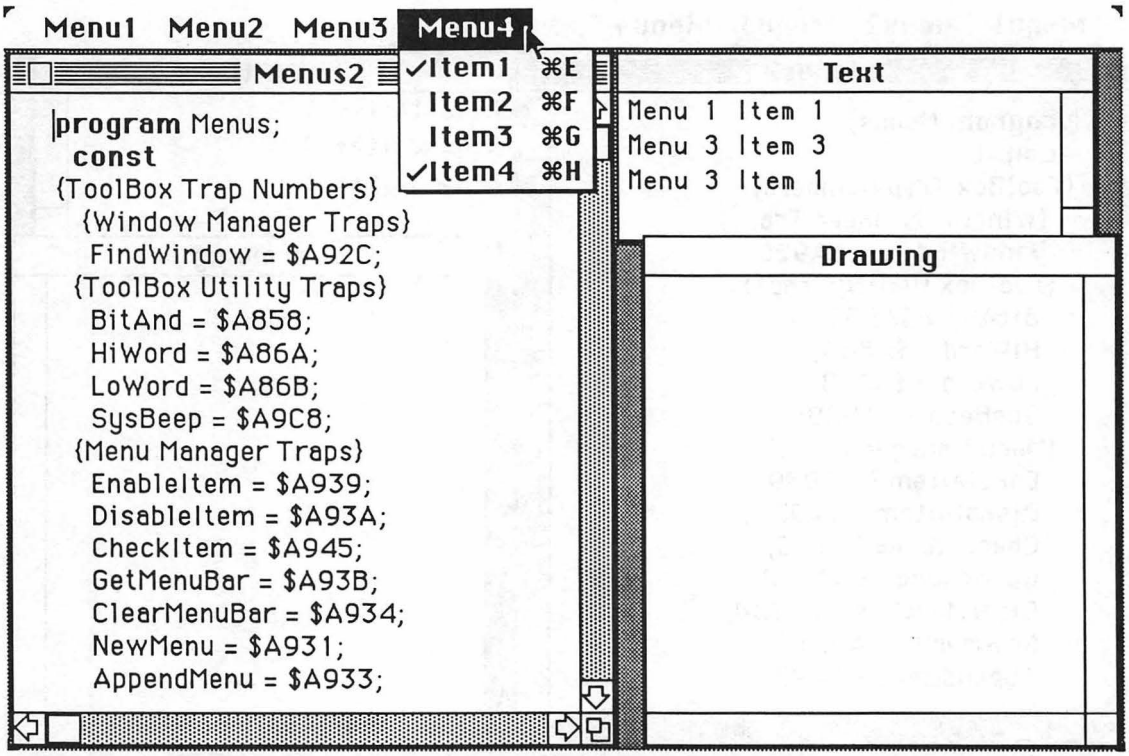


Figure 8.12 Menu 4 with checked items

gets us to the DoMenu1 procedure. DoMenu1 does a case statement on the selected item ID number and executes one line of code for each case (item). For the Exit item, DoMenu1 sets the variable Abort to True. When we return to the background loop, it checks the Abort variable. If Abort is True, we exit the background loop defined by the Repeat-Until block. After exiting the background loop, we do some cleanup work before ending the program: we restore the old menu bar and remove the storage space allocated for our menu data structures.

CHAPTER

9 DO IT IN A WINDOW

What's in a Window

Creating Windows

Writing and Drawing in Windows

Mouse Events

Activate Events

Update Events

WHAT'S IN A WINDOW

Everything that a properly designed application displays on the Macintosh screen is displayed in a window. A window is smaller than the screen, and multiple windows may appear on the screen at any given time. Windows may overlap or completely obscure others. The window manager keeps track of where the windows are, which ones overlap others, and what window regions need to be updated (a region is a QuickDraw data structure that describes an arbitrary shape).

An application creates and controls windows by calling various window-manager, event-manager, and QuickDraw routines (for descriptions of those routines, see Appendix E, Toolbox Routines). The window manager keeps track of information about the windows and provides routines to manipulate them—remove them, change their size, change their attributes, update their contents, and the like. The event manager notifies the program of window-related events, and the QuickDraw package provides routines to draw the windows' contents.

A window usually displays only part of a picture or text. If the window has a size box, you may change the window's size and thus the amount of the document it displays. If the window has scroll bars, you can determine what part of the picture or text the window displays. When you scroll the window, it appears to move over the document, displaying a portion the same size as the window.

Windows come in many types and styles (figure 9.1). Some are used to display documents; others display dialog or alert boxes. Most applications programmers will use one of the standard predefined window types and will define custom windows only for special uses. We will stick to predefined window types in our discussion of how to create and use windows.

In figure 9.2 we see a document window with its parts identified. Not all document windows have all of the features shown there; every document window has a title bar, but the close box, size box, and scroll bar are optional. The appearance of the title bar varies, depending on whether the window is active or inactive.

An application program may have multiple windows on the screen, but only one window may be active (figure 9.3). The program is not required to have an active window; it may have a number of windows displayed that are all inactive.

When windows overlap, the window manager displays them as if they were in different planes. We think of some windows as being behind others. While we may not be able to see all of a window that is behind another window, its contents still exist and will be displayed if we bring

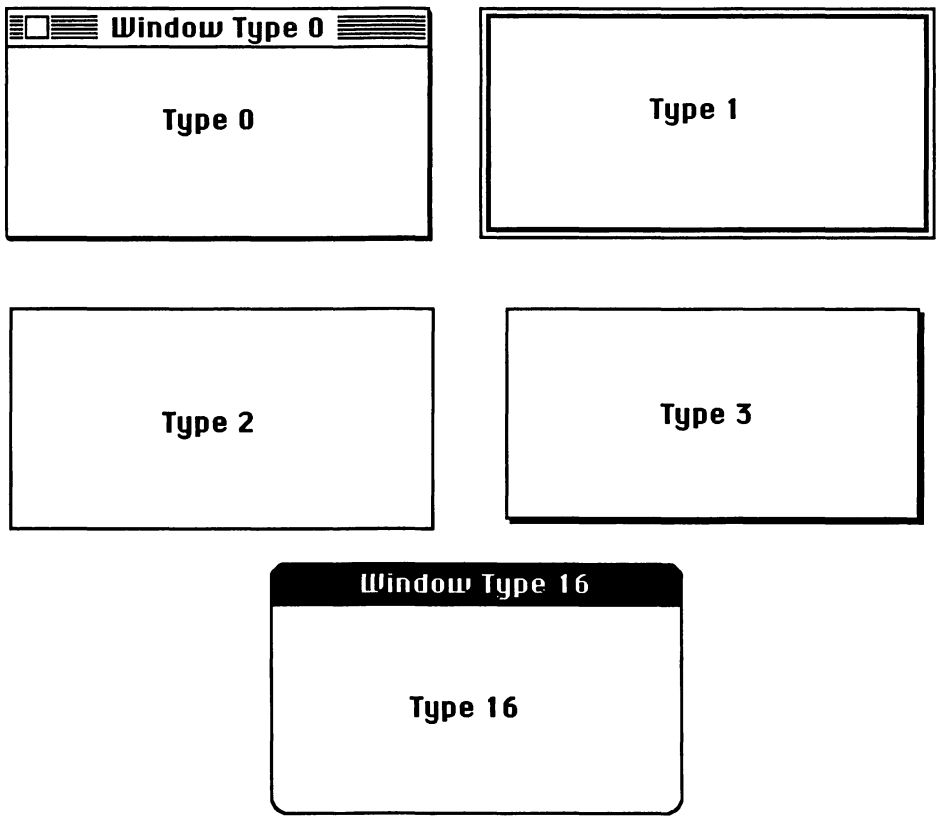


Figure 9.1 Window types

the window to the front. The active window is always the frontmost window. While it is possible for a program to write new data in an inactive window, most applications won't require that. A better practice is to keep your design simple and write new data into the active window only.

So far we've identified two attributes of windows: windows are active or inactive, and overlapping windows appear in different planes. Windows may also be either open or closed, and open windows may be visible or invisible.

When a program creates a new window, it must first open the window. Opening a window creates its data structures and adds it to the window manager's window list. When the program is finished with a window, it may close the window, removing it from the window manager's list and disposing of its data structures.

An open window may be visible or invisible. An invisible window is not drawn on the screen. Windows that are visible are drawn on the

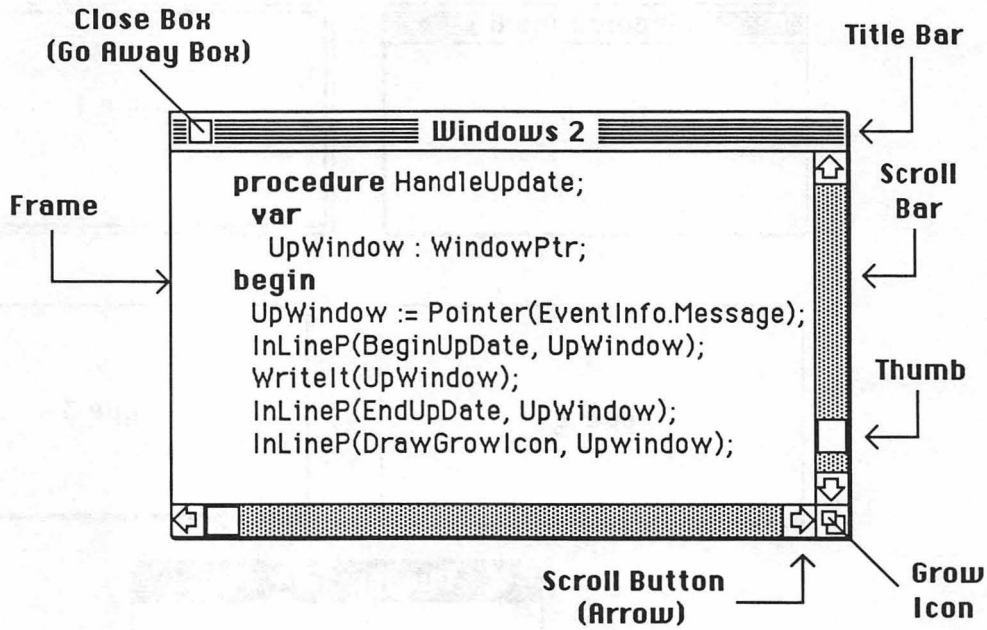


Figure 9.2 Standard document window

screen—that is, the parts of them that are not obscured by other windows. If a window is completely covered by other windows, it may be technically visible (marked as visible in its data structure) but not displayed. The active window is always visible and cannot be obscured by other windows, since it is the front window.

Window Attributes

Open/Closed
Plane
Visible/Invisible
Active/Inactive

The same part of the window is sometimes called a region and sometimes called a box. For instance, *size box* and *go-away region* are both terms that describe the box that you click to make a window disappear. A *box* is the actual drawing on the screen. A *region* is a QuickDraw data structure that defines an area of arbitrary shape. Sometimes the Apple Macintosh documentation uses the term *region* to refer to an area of the actual drawing on the screen, such as part of a window.

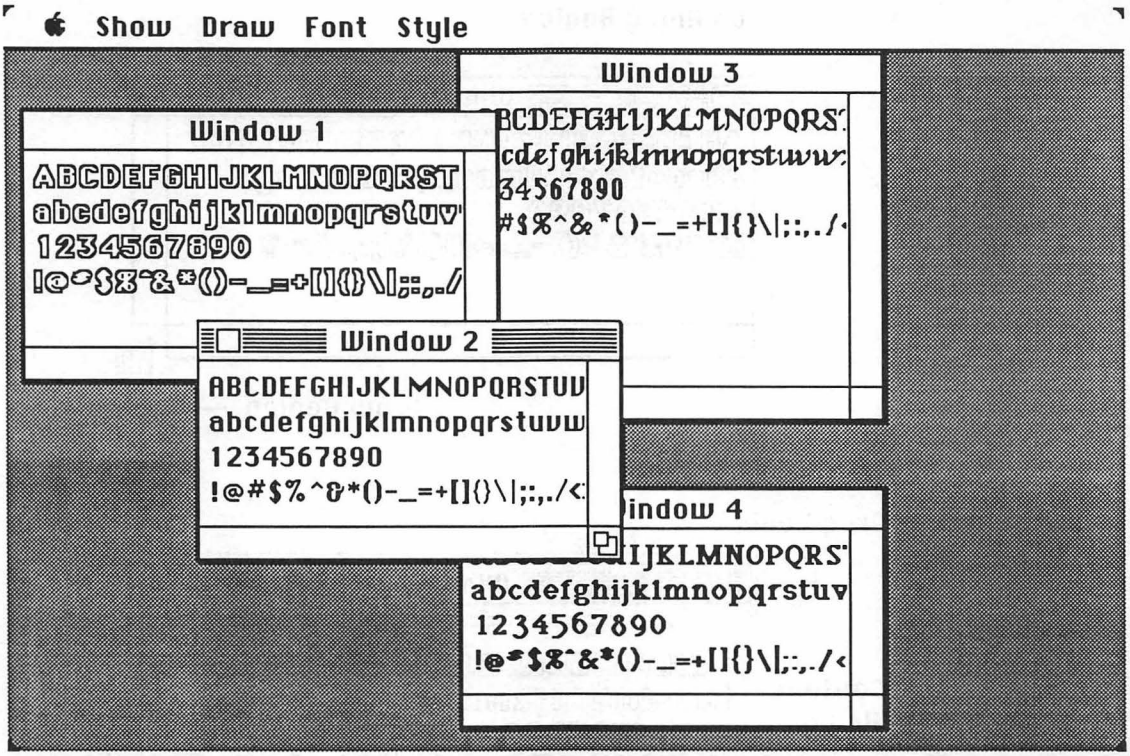


Figure 9.3 Multiple windows

A window has various regions that are used for different purposes (figure 9.4). The most obvious region is the content region, used to display part of a document or other data. The rest of the window consists of the window frame, which includes the title bar and the lines that define the window edges. The window frame and content regions are further divided into additional regions.

The drag and go-away regions are both part of the window frame. Clicking the mouse in the drag region and dragging the mouse drags an outline of the window around on the screen. When the user releases the mouse button, the window is moved to the new location, defined by the outline that the user dragged. Clicking the mouse in the go-away region causes the window to become invisible.

The grow region is part of the content region and is located in the lower right-hand corner of the window. It works like the drag region, but the upper left-hand corner of the window doesn't move. Clicking the mouse in the grow region and dragging the mouse drags an outline that

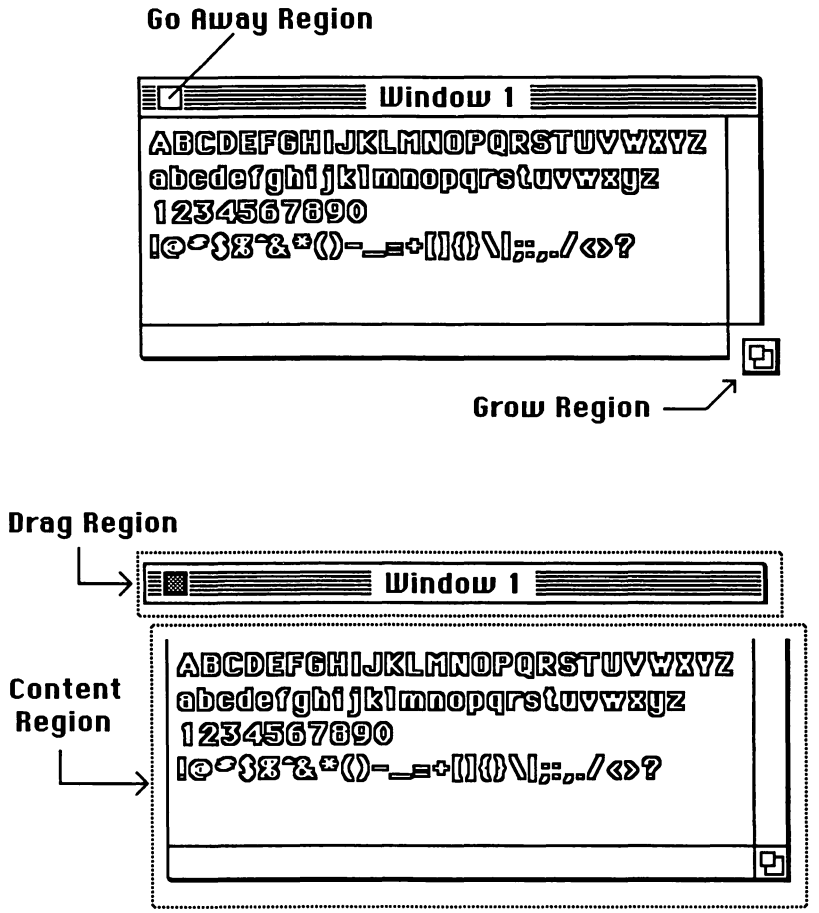


Figure 9.4 Window regions

defines a new size for the window. When the user releases the mouse button, the window changes size.

When a window that was partially obscured becomes the front window, part of it needs to be redrawn; the window management software in the Macintosh keeps track of which parts need updating. When a program creates a window or updates it, the window manager draws the window frame, and usually the application program draws the window contents.

A program can update the window contents in several ways. The application program may do it, or the application can give the window manager a "picture" data structure and let the window manager take care

of updating the window contents (a "picture" is a list of QuickDraw calls that created the drawing in the window).

A portion of a window might need updating because it was previously overlaid and now needs to be displayed, or because the window has been enlarged and the newly visible portions need to be drawn. The window manager keeps track of which window portions need to be updated and accumulates them in the update region. The update region can encompass various separate parts of the window. It's really just a data structure that lists areas that need updating. It only accumulates areas of the content region.

The window manager keeps track of which parts of the window frame need to be updated, but it doesn't pass that information to the application program. The window manager itself updates the frame.

An application program finds out that a window needs to be updated by calling an event-manager routine and getting an update event in return. After that, the application is responsible for drawing the parts of the window that are in the update region, but it gets help from the window manager. The application calls a window-manager routine that restricts drawing in the window to the update region. Then, when the application executes code to draw the entire window contents, only the update region is actually drawn on the screen.

In this chapter, we will use many of the window-manager and other toolbox routines that create and control windows. A demonstration program called Windows will show you how to use many of the toolbox routines to create and control windows. The program is written in Macintosh Pascal and uses InLine statements to call toolbox routines. It is very similar in structure to the skeleton menu program in chapter 8. In fact, it was developed as a modification of that program.

You will find listings of parts of the program throughout this chapter. The complete Windows program listing is at the end of the chapter.

Like other Macintosh programs, Windows is controlled by menus. You use the menus to individually make each of the four windows visible or invisible and to draw their contents—samples of several type fonts and styles (figure 9.5). You select the current type font and style, using the Font and Style menus. The next window drawn displays part of the Macintosh character set in that font and style (figure 9.6).

CREATING WINDOWS

Like a menu, a window consists of a data structure and a program. The program can be a standard window-definition program that is part of the Macintosh toolbox or a custom window program that you write yourself.

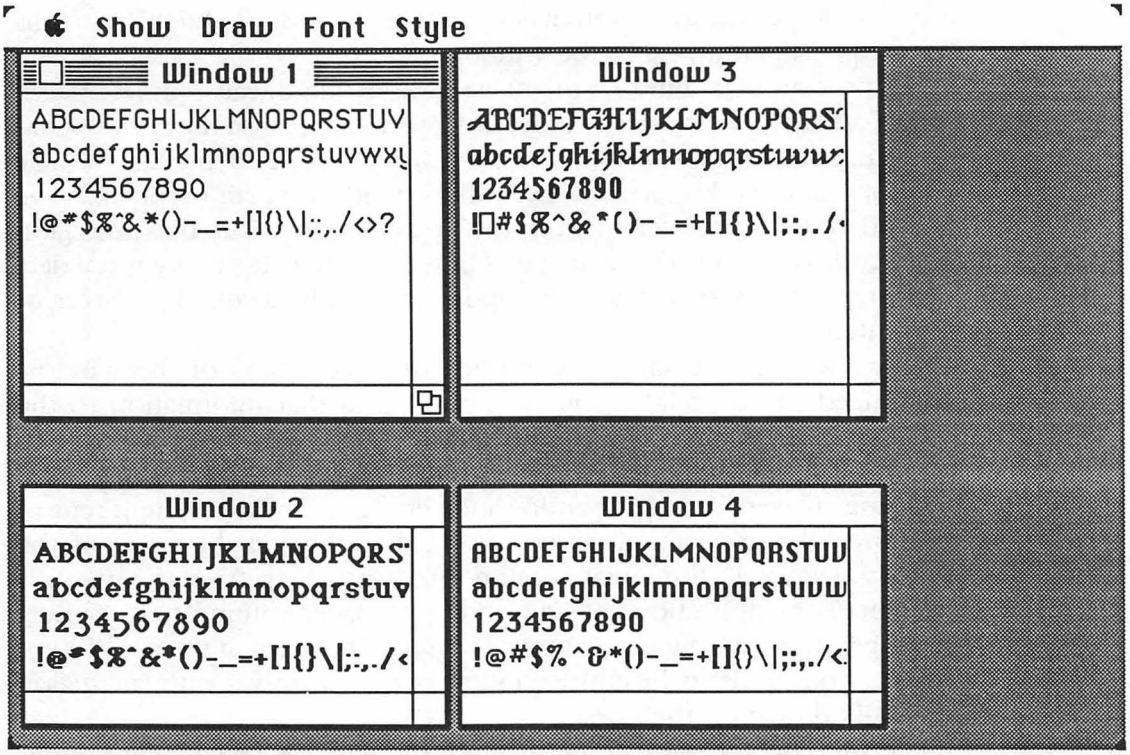


Figure 9.5 Windows program display

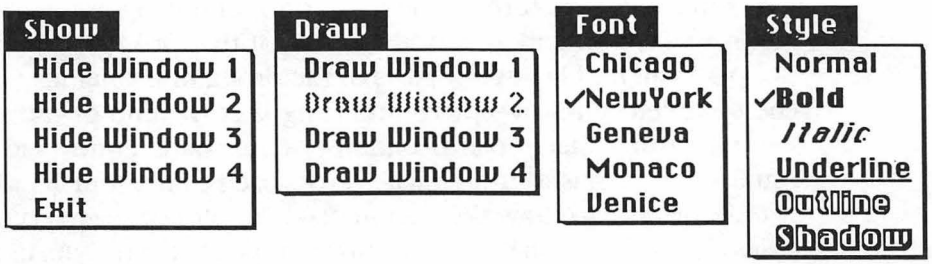


Figure 9.6 Menus for Windows program

Once you have the window program, you can use it with toolbox routines and window data structures to define specific windows that your application program will display.

We may define a specific window by using a series of calls to window-manager routines or by getting the window definition from a

resource file. In this book, we will limit our discussion of window creation to the methods that do not use resource files.

A window's data structure is called a *window record*. The window manager maintains a pointer that always points to the window record of the front window. Each window record contains a pointer to the next window behind it. In technical terms, that means the window records are a singly linked list of data structures, linked in the order of the window planes. The window manager thus knows which window is in front and the order of the remaining windows.

The first element of a window record is a graph port record, defining the window's content region for the QuickDraw routines that will draw in it. (For information about QuickDraw and the graph port, see *The Magic of the Macintosh*, the second book in this series, or Appendix B in the *Macintosh Pascal Technical Appendix*.) The remainder of the window record contains various flags and pointers to other data structures that define the current attributes and appearance of the window.

Window Record

```
TYPE WindowRecord = record
    port:           GrafPort;
    windowKind:     INTEGER;
    visible:        BOOLEAN;
    hilited:        BOOLEAN;
    goAwayFlag:     BOOLEAN;
    spareFlag:      BOOLEAN;
    structRgn:      RgnHandle;
    contRgn:        RgnHandle;
    updateRgn:      RgnHandle;
    windowDefProc:  Handle;
    dataHandle:     Handle;
    titleHandle:    StringHandle;
    titleWidth:     INTEGER;
    controlList:    Handle;
    nextWindow:     WindowPeek;
    windowPic:      PicHandle;
    refCon:         LongInt
END;
```

Most of the elements of the window record are definitions of window features we have discussed. The windowKind field defines the type of the predefined or custom window. The hilited flag determines whether the window is highlighted; only the active window gets highlighted. For

standard document windows, highlighting consists of drawing a series of horizontal lines in the title bar of the window.

The structure region (structRgn) defines the entire window region, including the window frame and the content region. The windowDefProc field points to the set of procedures that define the window. The dataHandle is used by the window-definition procedure and is of no concern to the application program. The controlList points to a list of controls (buttons, dials, and so on) that are maintained by the control manager. NextWindow is the pointer to the next window to the rear. The windowPic field points to a QuickDraw picture that the window manager can use to update the window.

Finally, refCon is a field for use by the application program. The application can use it to indicate any information about the window that it needs to maintain. The application program writes to and reads from refCon by using two window-manager routines.

There are two ways that a program can reference a window record: with a window pointer or with another pointer called WindowPeek. The window pointer is a GraphPtr type—a pointer to a graphics port. This is the most convenient pointer to use because the majority of things the application program wants are in the graph port. (Pascal lets us get away with that because the graph port is the first element of a window record, and a pointer to the window record actually does point to a graph-port data structure.)

The window manager needs direct access to the other parts of the window record, but the application program usually does not. The application can call window-manager routines to access those fields. If an application program needs to do something special with a window, it may access the rest of the window record directly with the WindowPeek pointer. WindowPeek's type identifies it as a pointer to a window record, not to a graph port.

```
TYPE GraphPtr = ^GraphPort;
TYPE WindowPtr = GraphPtr;
TYPE WindowPeek = ^WindowRecord;
```

Let's take a look at the window data structures in our program, starting with the data type definitions (listing 9.1). First we have the standard definitions of the pointer and handle types. Next we define the window record type—it's something of a cop-out. All we did was allocate space for the window record; we didn't define the record format (we used this shortcut to avoid having to define a legion of other data types and records referenced in the window record definition). WindowPtr is defined as a pointer to a window record.

Listing 9.1 Window Data Types

```
Ptr = ^LongInt;
Handle = ^Ptr;
WindowRecord = array[1..78] of Integer;
WindowPtr = ^WindowRecord;
Rectangle = array[1..4] of Integer;
EventRecord = record
  What : Integer;
  Message : LongInt;
  When : LongInt;
  Where : LongInt;
  Modifiers : Integer;
 end;
```

In the program listing, Rectangle is an array of integers that specify the bounds of a rectangle. The event record is the same one we used in the skeleton menu program. Following the event record is a series of type definitions that allow us to define the first part of the QuickDraw graph port record (GrafPort). We don't need all of the port definition because we only need to get to the portRect field. The entire GrafPort record appears later in this chapter. It is fully explained in Appendix B of the *Macintosh Pascal Technical Appendix*.

In listing 9.2, in the variable definition section of the program, we define Window1 through Window4 to be pointers to window records. We define Window1Rec through Window4Rec to be the window records. We could have let the window manager allocate space for the window records from the heap, but we decided to allocate the space in our program instead.

In the initialization section, immediately following the menu intialization, we create the four windows. The first three program statements don't really have anything to do with creating windows. The HideAll procedure hides all of the windows created by the Pascal interpreter, and the next two lines of code initialize two rectangle data structures.

A rectangle is defined by four integers that locate the upper left-hand corner (x, y) and the lower right-hand corner (x, y) of the rectangle. The two rectangles in question define: (1) the area of the screen in which we will permit windows and (2) the maximum size of a window. We need to set these parameters because we don't want a window to extend into the menu bar, nor do we want to create a menu larger than the screen.

Listing 9.2 Window Initialization

```
{initialize Windows}
HideAll;
InLineP(SetRect, @GlobalBounds, 4, 24, 508, 338);
InLineP(SetRect, @MaxWSize, 0, 0, 508, 338);
InLineP(SetRect, @BoundsRect1, 8, 40, 200, 189);
InLineP(SetRect, @BoundsRect2, 8, 240, 200, 330);
InLineP(SetRect, @BoundsRect3, 208, 40, 400, 189);
InLineP(SetRect, @BoundsRect4, 208, 240, 400, 330);
Visible1 := FALSE;
Visible2 := FALSE;
Visible3 := FALSE;
Visible4 := FALSE;
ProcID1 := 0;
ProcID2 := 0;
ProcID3 := 0;
ProcID4 := 0;
Behind1 := Pointer(-1);
Behind2 := Pointer(-1);
Behind3 := Pointer(-1);
Behind4 := Pointer(-1);
GoAway1 := True;
GoAway2 := True;
GoAway3 := False;
GoAway4 := False;
Window1 := Pointer(LInLineF(NewWindow, @Window1Rec, BoundsRect1,
     'Window 1', Visible1, ProcId1, Behind1, GoAway1, 0 + 0));
Window2 := Pointer(LInLineF(NewWindow, @Window2Rec, BoundsRect2,
     'Window 2', Visible2, ProcId2, Behind2, GoAway2, 0 + 0));
Window3 := Pointer(LInLineF(NewWindow, @Window3Rec, BoundsRect3,
     'Window 3', Visible3, ProcId3, Behind3, GoAway3, 0 + 0));
Window4 := Pointer(LInLineF(NewWindow, @Window4Rec, BoundsRect4,
     'Window 4', Visible4, ProcId4, Behind4, GoAway4, 0 + 0));
```

For the most part, the rest of the window intialization code assigns values to all of the window parameters. The BoundsRect1 through BoundsRect4 variables define the locations and dimensions of the four windows at the time they are first displayed. The size of a window may be changed later by the program or by the user if the window has a size box. The SetRect procedure takes a set of four values and puts them into a rectangle data structure.

Visible1 through Visible4 are Boolean variables that determine whether the window will be visible or invisible when defined. If a window is not made visible at the time it is defined, the program must call the ShowWindow procedure at the time it wants to make the window visible.

In our program, we decided to make all of the windows invisible at definition time and give the user menu choices to make them visible (see the Show menu in figure 9.6).

The ProcID identifies the procedure that defines the window and thus the window type. A ProcID of zero makes the window a document window.

The Behind parameter selects the window's plane. Behind is a pointer to a window record; it identifies the window that the new window will be behind. By assigning a value of −1 to the parameter, we make a window the front window. A value of Nil makes the window the last (rearmost) window.

GoAway is a Boolean variable that determines whether or not the window has a close box in the title bar.

The last four lines of the initialization section each call NewWindow, the window-manager procedure that creates a window. With the exception of three, we have already defined all of the parameters that must be passed to NewWindow. The @WindowRec parameter is a pointer to the storage area for the window record. The text string in single quotes (for example, 'Window 1') is the window title that appears in the title bar.

The last parameter is the refCon parameter. We did not use a variable to define refCon but passed a constant instead. Remember refCon from the window-record data structure? It is a field in the window record put there for use by the application program; the window manager never reads it. RefCon is a long integer (longint), so we needed to pass a long integer constant. To do that, we used a Macintosh Pascal programming trick: since the result of any arithmetic operation in Macintosh Pascal is a long integer, we added two zeros together to get a long integer with the value zero.

The NewWindow routine is actually a Pascal function and returns a pointer to a window record. Since we provided a window-record storage area to NewWindow, it returns a pointer to that area.

At the end of the program, we call the CloseWindow procedure for each window. CloseWindow removes a window's record from the linked list of windows, deallocates all data structures that it allocated for that window on the heap, and erases the window from the screen. The window manager maintains some of its own window data structures on the heap. Our program cannot access them.

The NewWindow routine sets the window's graph-port data-structure fields to the same default values as the QuickDraw graph-port open routine. Each graph port (and hence each window) has a font

variable in the graph port record. The font field identifies the font that QuickDraw routines use to draw text on the screen. NewWindow sets the font type to the default application font (usually Geneva), the font style to plain, and the font size to 12 points. Later, our program changes those settings in response to selections in the Font and Style menus.

When we run the program, we see that the first menu (the Show menu) has selections to show Windows 1 through 4. When the user selects an item from the Show menu, the program calls the window manager's ShowWindow procedure. ShowWindow makes the window visible and displays it on the screen. We pass it a pointer to the window record of the window we want to make visible.

Once the Show menu is visible, we change the menu selection to read Hide Window. If the user selects Hide Window, we call the window manager HideWindow procedure, passing it the window record pointer. The routine marks the window as invisible and erases it from the screen. HideWindow does not close or destroy the window or its contents; it merely hides them from view. We can put the same window back on the screen later by calling ShowWindow.

You can see the statements that call the ShowWindow and HideWindow routines from the DoMenu1 procedure in the complete program listing at the end of this chapter.

WRITING AND DRAWING IN WINDOWS

This section could be called just Drawing in Windows because there's no distinction between writing text and drawing. When you write text in a window, each character is individually drawn, just like any picture you might draw in the window. An application draws text in a window by calling QuickDraw routines.

QuickDraw uses the window's graph-port data structure to control how it draws text or other pictures. Three of the fields in the graph-port data structure that control drawing text are text font, text style, and text size. The application program sets those fields by calling QuickDraw routines and then calls other QuickDraw routines that draw characters in the selected font, style, and size.

The graph-port data structure also defines the drawing rectangle—the size and location of the area that QuickDraw may draw in. The application program needs to access the graph port rectangle in order to erase window contents, erase scroll bars, draw scroll bars, and erase the grow icon.

Graph Port Record

```
GrafPort = RECORD
    device:      Integer;
    portBits     BitMap;
    portRect:    Rect;
    visRgn:      RgnHandle;
    clipRgn:     RgnHandle;
    bkPat:       Pattern;
    fillPat:     Pattern;
    pnLoc:       Point;
    pnSize:      Point;
    pnMode:      Integer;
    pnPat:       Pattern;
    pnVis:       Integer;
    txFont:      Integer;
    txMode:      Integer;
    txSize:      Integer;
    spExtra:     LongInt;
    fgColor:     LongInt;
    bkColor:     LongInt;
    colrBit:     Integer;
    patStretch:  Integer;
    picSave:     QDHandle;
    rgnSave:     QDHandle;
    polySave:    QDHandle;
    grafProcs:   QDProcsPtr;
END;
```

Most of the routines in our program that draw in the window's content region require that we pass them a pointer to the graph port. We pass a window pointer (remember, it actually points to the window's graph port). If the routine we called needs to access fields in the graph port record, it usually defines the pointer to be a graph port pointer.

Our routines in the demonstration program access the graph port record directly only to read the portRect field. In the program, our definition of the graph port record defines only the fields up to and including the portRect (listing 9.3).

Our program has two routines that draw in the content region of the window, EraseScrollBars and WriteIt. EraseScrollBars does just what its name implies and does not alter any other area of the window. The WriteIt procedure draws several lines of text in the window in whatever type style and font were last selected. For changing the style and font, we have a

Listing 9.3 Graph-Port Record Definition

```
PortRecord = record
  device : integer;
  portBits : BitMap;
  portRect : Rectangle;
 end;
```

procedure called SelectType. Before calling WriteIt, we always call SelectType to set the graph port's style and font fields to the values that the user last selected with the Font and Style menus.

EraseScrollBars uses two other routines that access the graph port record, GetVertBar and GetHorizBar. They each return a rectangle data structure that contains the location of one of the scroll bars. The rectangle that GetHorizBar returns extends into the grow icon because we want EraseScrollBars to erase the grow icon as well as the scroll bars.

QuickDraw always draws in the current graph port and maintains a pointer to the current graph port. When we want QuickDraw to draw in a particular window, we must set the current graph port pointer to point to that window. It's good programming practice both to save the current graph port pointer before changing it and to restore the pointer when you are through using QuickDraw.

We pass the EraseScrollBars, SelectType, and WriteIt procedures a pointer to a window record, so each of those procedures saves the pointer to the current graph port and sets it to point to the window's graph port. When each procedure is done, it restores the graph port pointer to its original value.

Looking at the WriteIt procedure, we see an InLine call to GetPort to save the old graph port pointer. The last statement in the procedure calls SetPort to restore the pointer (listing 9.4).

The procedure calling WriteIt passes it a pointer to a window record. Since a window record pointer is actually a pointer to a graph port, WriteIt can define that parameter as a graph port pointer. After setting QuickDraw's current graph port pointer to the value of the window's graph port pointer, WriteIt uses the pointer to access the graph port's rectangle field to find the dimensions of the drawing rectangle. WriteIt uses that information to erase the drawing area before doing its own drawing.

The window's scroll bars and grow icon are part of the content region and hence are included in the graph port's drawing rectangle.

Listing 9.4 WriteIt Procedure

```
procedure WriteIt (WritePort : PortPtr);
 var
  x, y, LineHeigth : integer;
  OldPort : PortPtr;
  theRect : Rectangle;
begin
 InLineP(GetPort, OldPort);
 InLineP(SetPort, WritePort);
 theRect := WritePort^.portRect;
 InLineP(EraseRect, theRect);
 InLineP(ShowPen);
 x := 4;
 y := 16;
 LineHeigth := 16;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
 y := y + LineHeigth;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, 'abcdefghijklmnopqrstuvwxyz');
 y := y + LineHeigth;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, '1234567890');
 y := y + LineHeigth;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, '!@#$%^&*()-_=+[]{}\|;:,./<>?');
 EraseScrollBars(WritePort);
 InLineP(DrawGrowIcon, WritePort);
 InLineP(SetPort, OldPort);
end;
```

Erasing the drawing area erases the scroll bars, so we must restore them when we are through drawing. Another result of the scroll bars' being part of the drawing rectangle is that when we ask QuickDraw to draw in the window, it may draw in the scroll-bar and grow-icon areas. Looking at the end of the WriteIt procedure, you will see that we erase the scroll bars again before WriteIt is finished.

MOUSE EVENTS

The application program decides how to control its menus. Most of the control it exerts over menus is the result of actions taken by the user, so the application is really serving as a vehicle to allow the user to control the window. When the user does something with the mouse, the application program finds out what the user did by calling event-manager routines.

The event manager reports many different types of events, but we are interested in three types only: window update events, activate events, and mouse events that control windows. Update and activate events are generated by the window manager, but mouse events are generated by the user's clicking the mouse in a window region.

To change the location of a window, the user presses the mouse button with the pointer in the title bar of the window (the drag region) and then drags an outline of the window to a new location. By clicking the mouse in the close box, the user causes the application to hide the window. To change the size of the window, the user drags the grow icon in the window's lower right-hand corner. Clicking the mouse anywhere in the content region of a window causes it to become the active window. The event manager reports each of these events as a mouse event. The application must then call a window-manager routine to find out where the mouse was clicked and, consequently, which type of mouse event it is dealing with.

In the background loop of our program, we call GetNextEvent to get the next event in the event queue and do a case statement based on the event type. For mouse events, we call the procedure HandleMouse (listing 9.5).

HandleMouse calls the window-manager function FindWindow to find out where the mouse pointer was when the user pressed the mouse button. Depending on where the mouse pointer was in the window, HandleMouse then calls one of the following procedures: HandleMenu, HandleContent, HandleDrag, HandleGrow, or HandleGoAway.

If the mouse pointer was in the content region of the window, the user was selecting the window to make it the active window. The HandleContent routine is pretty simple; it calls the SelectWindow toolbox procedure to select the window and calls a routine to update the state of the items in Menu 2.

HandleDrag is even simpler. It just calls the window-manager procedure DragWindow. DragWindow tracks the mouse movement (as long as the mouse button is down) by moving an outline of the window around the screen, following the mouse pointer. When the user releases the mouse

Listing 9.5 HandleMouse

```
procedure HandleMouse;
 const
  InDesk = 0;
  InMenuBar = 1;
  InSysWindow = 2;
  InContent = 3;
  InDrag = 4;
  InGrow = 5;
  InGoAway = 6;
 var
  AWindow : WindowPtr;
  WhereItIs : Integer;
begin
 WhereItIs := WInLineF(FindWindow, EventInfo.Where, @AWindow);
 case WhereItIs of
  InMenuBar :
   HandleMenu;
  InContent :
   HandleContent(AWindow);
  InDrag :
   HandleDrag(AWindow);
  InGrow :
   HandleGrow(AWindow);
  InGoAway :
   HandleGoAway(AWindow);
  otherwise
   InLineP(SysBeep, 5);
 end
end;
```

button, DragWindow moves the window, generates an update event, and returns to its caller.

The HandleGoAway procedure calls the window-manager procedure HideWindow to make the window invisible. HandleGoAway also updates the items in Menu 1.

The most complicated mouse-event handler for window events is the HandleGrow procedure, and even it isn't very complex. It has three jobs

to do: find out the new dimensions of the window, change the size of the window, and include the scroll bars in the update region.

HandleGrow calls the window-manager function GrowWindow to find out the new window size. Like DragWindow, GrowWindow tracks the mouse movement while the mouse button is down. It follows the mouse-pointer movement with an outline of the window, showing its changed size. When the user releases the mouse button, GrowWindow returns the new window size to our HandleGrow routine. HandleGrow then calls SizeWindow, which redraws the window frame in the new size and generates an update event (listing 9.6).

SizeWindow keeps track of the areas of the screen that need to be updated when the window changes size. The routine that handles the update event should redraw those areas of the window. SizeWindow includes in the update region any changed areas of the content region, but it does not recognize anything special (like scroll bars) in the scroll-bar areas of the content region.

Listing 9.6 HandleGrow

```
procedure HandleGrow (WindowIn : WindowPtr);
 var
  WSize : LongInt;
  OldPort : WindowPtr;
  VBar, HBar : Rectangle;
begin
 VBar := GetVertBar(WindowIn);
 HBar := GetHorizBar(WindowIn);
 InLineP(GetPort, OldPort);
 InLineP(SetPort, WindowIn);
 WSize := LInLineF(GrowWindow, WindowIn, EventInfo.Where, MaxWSize);
 InLineP(InvalRect, VBar);
 InLineP(InvalRect, HBar);
 InLineP(SizeWindow, WindowIn, WSize, True);
 VBar := GetVertBar(WindowIn);
 HBar := GetHorizBar(WindowIn);
 InLineP(InvalRect, VBar);
 InLineP(InvalRect, HBar);
 InLineP(SetPort, OldPort);
end;
```

If the window is made larger, the program has to update the old scroll-bar areas to erase the scroll bars and fill that area with window contents. If the window is made smaller, the program has to draw new scroll bars. Since SizeWindow does not include scroll-bar areas in the update region, the HandleGrow procedure uses the window-manager procedure InvalRect to add the scroll-bar areas to the update region accumulated by GrowWindow. InvalRect always works with the current graph port, so we save and restore the old graph port just as we did in the WriteIt procedure.

ACTIVATE EVENTS

When a window becomes the active window, the window manager redraws the window frame, highlights it, and generates an activate event. A window's becoming inactive also generates an activate event. If you have at least two windows on the screen and click the mouse in the content region of the inactive window, you generate two activate events—one for the active window's becoming inactive and one for the inactive window's becoming active.

The application program must know when a window becomes active or inactive because it must make the necessary changes to the content region when the window changes state. These changes usually include filling in the scroll bars if the application uses scrolling, and redrawing the grow icon if the window has one.

When a window that does scrolling is active, it has the scroll bars filled with a pattern and has scrolling buttons shaped like arrows at each end of the scroll bar. Each scroll bar may also have a *thumb*, a box indicating the relative position of the window over the document it is displaying. When that window is inactive, the scroll bars are blank (completely white) with no scroll buttons (arrows) or thumbs. See figure 9.7.

The appearance of the grow icon is different in active and inactive windows. In an active window, the grow icon is in the lower right-hand corner at the intersection of the scroll bars. In an inactive window, the scroll-bar outlines are still there, but the grow icon is not. In both situations, the application program can call the window-manager procedure DrawGrowIcon to draw the icon and scroll-bar outlines. If the window is inactive, DrawGrowIcon draws just the scroll-bar outlines; if the window is active, the procedure also draws the grow icon at the intersection of the scroll bars.

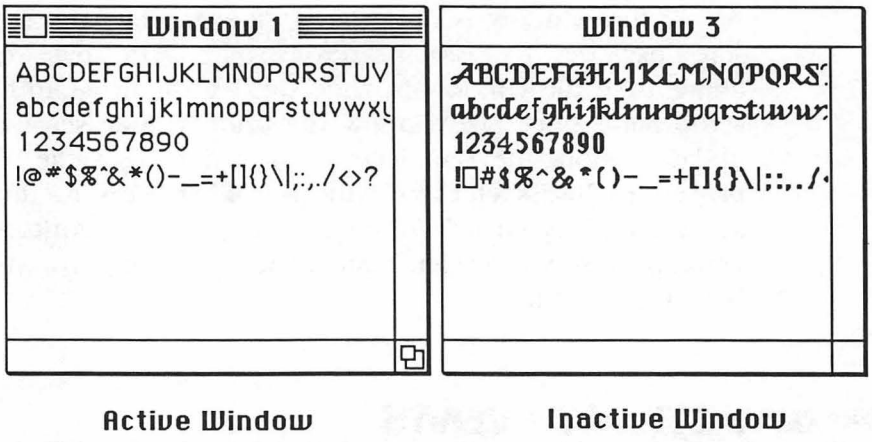


Figure 9.7 Active and inactive windows

The HandleActivate routine in our Windows program takes the same action for a window's becoming active as it does for a window's becoming inactive. It erases the scroll bars, calls DrawGrowIcon to draw the scroll-bar outlines (and possibly the grow icon), and calls a procedure to update the state of the items in Menu 2.

Our program does not do scrolling. If it did, the HandleActivate routine would need to draw the scroll arrows, the scroll-bar pattern, and the scroll-bar thumbs when the window becomes active.

UPDATE EVENTS

Some of the things that we can do to a window, like altering its size or uncovering part of it that was obscured by another window, require that we update the part of the content region that is newly visible. The window manager detects situations that require updating part of a window and generates an update event.

An application program actually has two ways to update the window's contents: (1) it can redraw the content area when it gets an update event, or (2) using a QuickDraw picture data structure, it can avoid having to redraw the content region.

A picture data structure defines a picture in terms of a set of QuickDraw calls, and the picture in this case defines the window contents. The application program stores a handle to a picture data structure in the window record. If there is a handle to a picture in the WindowPic field of

the window record, the window manager does not generate an update event. Instead, it uses the picture data structure to update the window without generating an event.

Our demonstration program uses update events, not handles, to QuickDraw pictures.

When it detects an update event, the application program draws the portion of the window that is affected by the change. The application needs to draw only the content region of the window; the window manager draws any part of the frame that needs updating.

While the user is moving a window, changing its size, or changing its plane, the window manager accumulates a description of the areas that need updating in a QuickDraw data structure called a *region*. When the application program gets an update event, it must redraw the update region. If the application called the routine that normally draws in the graph port, the routine would draw the entire area included in the graph port's rectangle. We can, however, force QuickDraw to draw just in the areas included in the update region.

Let's take a look at the HandleUpdate routine in listing 9.7. Before calling WriteIt to update the window, the procedure calls the window-manager routine BeginUpdate. BeginUpdate limits the area in which QuickDraw draws to areas that are part of the graph port's rectangle and are also part of the update region. After calling WriteIt, the procedure calls EndUpdate to restore QuickDraw's ability to draw anywhere in the window's content region.

Listing 9.7 Windows Program

```
program Windows;
 const
{ToolBox Trap Numbers}
   {Window Manager Traps}
  DrawGrowIcon = $A904;
  TrackGoAway = $A91E;
  FindWindow = $A92C;
  NewWindow = $A913;
  CloseWindow = $A92D;
  ShowWindow = $A915;
  HideWindow = $A916;
  SelectWindow = $A91F;
  FrontWindow = $A924;
  DragWindow = $A925;
  GrowWindow = $A92B;
  SizeWindow = $A91D;
  BeginUpDate = $A922;
  EndUpDate = $A923;
  InvalRect = $A928;
  {ToolBox Utility Traps}
  BitAnd = $A858;
  HiWord = $A86A;
  LoWord = $A86B;
  SysBeep = $A9C8;
  {Menu Manager Traps}
  SetItem = $A947;
  EnableItem = $A939;
  DisableItem = $A93A;
  CheckItem = $A945;
  GetMenuBar = $A93B;
  ClearMenuBar = $A934;
  NewMenu = $A931;
  AppendMenu = $A933;
  InsertMenu = $A935;
  DrawMenuBar = $A937;
  SetMenuBar = $A93C;
  DisposeMenu = $A932;
  MenuSelect = $A93D;
  MenuKey = $A93E;
```

Continued

Listing 9.7 *Continued*

```
  HiLiteMenu = $A938;
  {Quickdraw Traps}
  InitCursor = $A850;
  SetRect = $A8A7;
  TextFace = $A888;
  TextFont = $A887;
  MoveTo = $A893;
  DrawString = $A884;
  ShowPen = $A897;
  SetPort = $A873;
  GetPort = $A874;
  EraseRect = $A8A3;
  {Event Manager Traps}
  GetNextEvent = $A970;
{Event Manager Constants }
  EventMask = $014F;
  MouseDown = 1;
  MouseUp = 2;
  KeyDown = 3;
  KeyUp = 4;
  AutoKey = 5;
  UpdateEvt = 6;
  diskEvt = 7;
  ActivateEvt = 8;
  ActDeact = 1;
  CmdKey = 256;
{Other Constants}
  AppleMenuID = 100;
  Menu1Id = 101;
  Menu2Id = 102;
  Menu3Id = 103;
  Menu4Id = 104;
  M1Item1 = 1;
  M1Item2 = 2;
  M1Item3 = 3;
  M1Item4 = 4;
  M1ExitItem = 5;
  M2Item1 = 1;
  M2Item2 = 2;
```

Continued

Listing 9.7 *Continued*

```
M2Item3 = 3;
M2Item4 = 4;
M3Item1 = 1;
M3Item2 = 2;
M3Item3 = 3;
M3Item4 = 4;
M3Item5 = 5;
M4Item1 = 1;
M4Item2 = 2;
M4Item3 = 3;
M4Item4 = 4;
M4Item5 = 5;
M4Item6 = 6;
Window1ID = 1;
Window2ID = 2;
Window3ID = 3;
Window4ID = 4;
Chicago = 0;
ApplicationFont = 1;
NewYork = 2;
Geneva = 3;
Monaco = 4;
Venice = 5;
Normal = 0;
Bold = 1;
Italic = 2;
Underline = 4;
Outline = 8;
Shadow = 16;
TopY = 1;
LeftX = 2;
BottomY = 3;
RightX = 4;

type
 Ptr = ^LongInt;
 Handle = ^Ptr;
 WindowRecord = array[1..78] of Integer;
 WindowPtr = ^WindowRecord;
```

Continued

Listing 9.7 *Continued*

```
 Rectangle = array[1..4] of Integer;
 EventRecord = record
   What : Integer;
   Message : LongInt;
   When : LongInt;
   Where : LongInt;
   Modifiers : Integer;
  end;
 QDByte = -128..127;
 QDPtr = ^QDByte;
 BitMap = record
   baseAddr : QDPtr;
   rowBytes : integer;
   Bounds : Rectangle;
  end;
 PortRecord = record
   device : integer;
   portBits : BitMap;
   portRect : Rectangle;
  end;
 PortPtr = ^PortRecord;

var
 OldMenuBar, AppleMenu, Menu1, Menu2, Menu3, Menu4 : Handle;
 Window1, Window2, Window3, Window4 : WindowPtr;
 Window1Rec, Window2Rec, Window3Rec, Window4Rec : WindowRecord;
 BoundsRect1, BoundsRect2, BoundsRect3, BoundsRect4 : Rectangle;
 GlobalBounds, MaxWSize : Rectangle;
 Visible1, Visible2, Visible3, Visible4 : Boolean;
 ProcID1, ProcID2, ProcID3, ProcID4 : Integer;
 Behind1, Behind2, Behind3, Behind4 : WindowPtr;
 GoAway1, GoAway2, GoAway3, GoAway4 : Boolean;
 RefCon1, RefCon2, RefCon3, RefCon4 : LongInt;
 EventInfo : EventRecord;
 Abort : Boolean;
 SelectedMenuID, SelectedItemID : Integer;
 SelectedFont, SelectedStyle : Integer;
 M3Item : Integer;
 M1Toggle : array[1..4] of boolean;
```

Continued

Listing 9.7 *Continued*

```
  M2Toggle : array[1..4] of boolean;
  M3Toggle : array[1..5] of boolean;
  M4Toggle : array[1..6] of boolean;
  FontItemMap : array[1..5] of integer;

procedure Initialize;
  var
    index : Integer;
begin
  for Index := 1 to 4 do
    begin
      M1Toggle[Index] := False;
      M2Toggle[Index] := False;
    end;
  for Index := 1 to 5 do
    M3Toggle[Index] := False;
  for index := 1 to 6 do
    M4Toggle[Index] := False;
  Abort := False;
  SelectedFont := Geneva;
  SelectedStyle := Normal;
  M3Item := 3;
  FontItemMap[1] := Chicago;
  FontItemMap[2] := NewYork;
  FontItemMap[3] := Geneva;
  FontItemMap[4] := Monaco;
  FontItemMap[5] := Venice;
  InLineP(InitCursor);
  FlushEvents(EventMask, 0);
{Save pointer to the old menu bar so we can restore it later }
  OldMenuBar := Pointer(LInLineF(GetMenuBar));
{Define New Menus}
  InLineP(ClearMenuBar);
  AppleMenu := pointer(LInLineF(NewMenu, AppleMenuID,
       Stringof(chr(20))));
  InLineP(AppendMenu, AppleMenu, 'Window Demonstration');
  Menu1 := Pointer(LInLineF(NewMenu, Menu1ID, 'Show'));
  InLineP(AppendMenu, Menu1, 'Show Window 1');
  InLineP(AppendMenu, Menu1, 'Show Window 2');
```

Continued

Listing 9.7 *Continued*

```
  InLineP(AppendMenu, Menu1, 'Show Window 3');
  InLineP(AppendMenu, Menu1, 'Show Window 4');
  InLineP(AppendMenu, Menu1, 'Exit');
  Menu2 := Pointer(LInLineF(NewMenu, Menu2ID, 'Draw'));
  InLineP(AppendMenu, Menu2, '(Draw Window 1;(Draw Window 2;(Draw
       Window 3;(Draw Window 4');
  Menu3 := Pointer(LInLineF(NewMenu, Menu3ID, 'Font'));
  InLineP(AppendMenu, Menu3, 'Chicago;NewYork;Geneva;Monaco;Venice');
  Menu4 := Pointer(LInLineF(NewMenu, Menu4ID, 'Style'));
  InLineP(AppendMenu, Menu4,
       'Normal;<BBold;<IItalic;<UUnderline;<OOutline;<SShadow');
  InLineP(InsertMenu, AppleMenu, 0);
  InLineP(InsertMenu, Menu1, 0);
  InLineP(InsertMenu, Menu2, 0);
  InLineP(InsertMenu, Menu3, 0);
  InLineP(InsertMenu, Menu4, 0);
  InLineP(CheckItem, Menu3, M3Item3, True);
  M3Toggle[M3Item3] := True;
  InLineP(CheckItem, Menu4, M4Item1, True);
  M4Toggle[M4Item1] := True;
{Draw the new menu bar}
  InLineP(DrawMenuBar);
{initialize Windows}
  HideAll;
  InLineP(SetRect, @GlobalBounds, 4, 24, 508, 338);
  InLineP(SetRect, @MaxWSize, 0, 0, 508, 338);
  InLineP(SetRect, @BoundsRect1, 8, 40, 200, 189);
  InLineP(SetRect, @BoundsRect2, 8, 240, 200, 330);
  InLineP(SetRect, @BoundsRect3, 208, 40, 400, 189);
  InLineP(SetRect, @BoundsRect4, 208, 240, 400, 330);
  Visible1 := FALSE;
  Visible2 := FALSE;
  Visible3 := FALSE;
  Visible4 := FALSE;
  ProcID1 := 0;
  ProcID2 := 0;
  ProcID3 := 0;
  ProcID4 := 0;
  Behind1 := Pointer(-1);
```

Continued

Listing 9.7 *Continued*

```
 Behind2 := Pointer(-1);
 Behind3 := Pointer(-1);
 Behind4 := Pointer(-1);
 GoAway1 := True;
 GoAway2 := True;
 GoAway3 := False;
 GoAway4 := False;
 Window1 := Pointer(LInLineF(NewWindow, @Window1Rec, BoundsRect1,
      'Window 1', Visible1, ProcId1, Behind1, GoAway1, 0 + 0));
 Window2 := Pointer(LInLineF(NewWindow, @Window2Rec, BoundsRect2,
      'Window 2', Visible2, ProcId2, Behind2, GoAway2, 0 + 0));
 Window3 := Pointer(LInLineF(NewWindow, @Window3Rec, BoundsRect3,
      'Window 3', Visible3, ProcId3, Behind3, GoAway3, 0 + 0));
 Window4 := Pointer(LInLineF(NewWindow, @Window4Rec, BoundsRect4,
      'Window 4', Visible4, ProcId4, Behind4, GoAway4, 0 + 0));
end;

function GetWindowID (theWindow : WindowPtr) : integer;
begin
 if theWindow = Window1 then
  GetWindowID := Window1ID
 else if theWindow = Window2 then
  GetWindowID := Window2ID
 else if theWindow = Window3 then
  GetWindowID := Window3ID
 else if theWindow = Window4 then
  GetWindowID := Window4ID
 else
  GetWindowID := 0;
end;

procedure SelectType (TWindow : WindowPtr);
 var
  OldPort : WindowPtr;
begin
 InLineP(GetPort, OldPort);
 InLineP(SetPort, TWindow);
 InLineP(TextFace, SelectedStyle);
 InLineP(TextFont, SelectedFont);
```

Continued

Listing 9.7 *Continued*

```
 InLineP(SetPort, OldPort);
end;

function GetVertBar (Port : PortPtr) : Rectangle;
 const
  LeftOffset = 16;
  RightOffset = 0;
  TopOffset = 0;
  BottomOffset = 16;
 var
  TheBar, WindowRect : Rectangle;
begin
 WindowRect := Port^.PortRect;
 TheBar[LeftX] := WindowRect[Rightx] - LeftOffset;
 TheBar[RightX] := WindowRect[RightX] - RightOffset;
 TheBar[TopY] := WindowRect[TopY] + TopOffset;
 TheBar[BottomY] := WindowRect[BottomY] - BottomOffset;
 GetVertBar := TheBar;
end;

function GetHorizBar (Port : PortPtr) : Rectangle;
 const
  LeftOffset = 0;
  RightOffset = 0;
  TopOffset = 16;
  BottomOffset = 0;
 var
  TheBar, WindowRect : Rectangle;
begin
 WindowRect := Port^.PortRect;
 TheBar[LeftX] := WindowRect[Leftx] + LeftOffset;
 TheBar[RightX] := WindowRect[RightX] - RightOffset;
 TheBar[TopY] := WindowRect[BottomY] - TopOffset;
 TheBar[BottomY] := WindowRect[BottomY] - BottomOffset;
 GetHorizBar := TheBar;
end;

procedure EraseScrollBars (theWindow : WindowPtr);
 var
```

Continued

Listing 9.7 *Continued*

```
  VertBar, HorizBar : Rectangle;
  OldPort : WindowPtr;
begin
 InLineP(GetPort, OldPort);
 InLineP(SetPort, theWindow);
 VertBar := GetVertBar(theWindow);
 InLineP(EraseRect, VertBar);
 HorizBar := GetHorizBar(theWindow);
 InLineP(EraseRect, HorizBar);
 InLineP(SetPort, OldPort);
end;

procedure WriteIt (WritePort : PortPtr);
 var
  x, y, LineHeigth : integer;
  OldPort : PortPtr;
  theRect : Rectangle;
begin
 InLineP(GetPort, OldPort);
 InLineP(SetPort, WritePort);
 theRect := WritePort^.portRect;
 InLineP(EraseRect, theRect);
 InLineP(ShowPen);
 x := 4;
 y := 16;
 LineHeigth := 16;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
 y := y + LineHeigth;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, 'abcdefghijklmnopqrstuvwxyz');
 y := y + LineHeigth;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, '1234567890');
 y := y + LineHeigth;
 InLineP(MoveTo, x, y);
 InLineP(DrawString, '!@#$%^&*()-_=+[]{}\|;:,./<>?');
 EraseScrollBars(WritePort);
 InLineP(DrawGrowIcon, WritePort);
```

Continued

Listing 9.7 *Continued*

```
 InLineP(SetPort, OldPort);
end;

procedure OnOffM2Items;
 var
  index, ActiveID : integer;
  ActiveWindow : WindowPtr;
begin
 ActiveWindow := Pointer(LInLineF(FrontWindow));
 ActiveID := GetWindowID(ActiveWindow);
 for index := 1 to 4 do
  if index = ActiveID then
   InLineP(DisableItem, Menu2, Index)
  else if M1Toggle[index] = True then
   InLineP(EnableItem, Menu2, Index)
  else
   InLineP(DisableItem, Menu2, Index);
end;

procedure DoMenu1;
begin
 case SelectedItemID of
  M1Item1 :
   if M1Toggle[1] = False then
    begin
     SelectType(Window1);
     InLineP(ShowWindow, Window1);
     InLineP(SetItem, Menu1, M1Item1, 'Hide Window 1');
    end
   else
    begin
     InLineP(HideWindow, Window1);
     InLineP(SetItem, Menu1, M1Item1, 'Show Window 1');
    end;
  M1Item2 :
   if M1Toggle[2] = False then
    begin
     SelectType(Window2);
     InLineP(ShowWindow, Window2);
```

Continued

Listing 9.7 *Continued*

```
      InLineP(SetItem, Menu1, M1Item2, 'Hide Window 2');
     end
    else
     begin
      InLineP(HideWindow, Window2);
      InLineP(SetItem, Menu1, M1Item2, 'Show Window 2');
     end;
  M1Item3 :
   if M1Toggle[3] = False then
    begin
     SelectType(Window3);
     InLineP(ShowWindow, Window3);
     InLineP(SetItem, Menu1, M1Item3, 'Hide Window 3');
    end
   else
    begin
     InLineP(HideWindow, Window3);
     InLineP(SetItem, Menu1, M1Item3, 'Show Window 3');
    end;
  M1Item4 :
   if M1Toggle[4] = False then
    begin
     SelectType(Window4);
     InLineP(ShowWindow, Window4);
     InLineP(SetItem, Menu1, M1Item4, 'Hide Window 4');
    end
   else
    begin
     InLineP(HideWindow, Window4);
     InLineP(SetItem, Menu1, M1Item4, 'Show Window 4');
    end;
  M1ExitItem :
   Abort := True;
  otherwise
 end;
 if SelectedItemID < 5 then
  M1Toggle[SelectedItemID] := not M1Toggle[SelectedItemID];
 OnOffM2Items;
end;
```

Continued

Listing 9.7 *Continued*

```
procedure DoMenu2;
begin
 case SelectedItemID of
  M2Item1 :
   begin
    if Pointer(LInLineF(FrontWindow)) <> Window1 then
     InLineP(SelectWindow, Window1);
    SelectType(Window1);
    WriteIt(Window1);
   end;
  M2Item2 :
   begin
    if Pointer(LInLineF(FrontWindow)) <> Window2 then
     InLineP(SelectWindow, Window2);
    SelectType(Window2);
    WriteIt(Window2);
   end;
  M2Item3 :
   begin
    if Pointer(LInLineF(FrontWindow)) <> Window3 then
     InLineP(SelectWindow, Window3);
    SelectType(Window3);
    WriteIt(Window3);
   end;
  M2Item4 :
   begin
    if Pointer(LInLineF(FrontWindow)) <> Window4 then
     InLineP(SelectWindow, Window4);
    SelectType(Window4);
    WriteIt(Window4);
   end;
  otherwise
 end;
 OnOffM2Items;
end;

procedure DoMenu3;
begin
 InLineP(CheckItem, Menu3, M3Item, False);
```

Continued

Listing 9.7 *Continued*

```
 M3Item := SelectedItemID;
 InLineP(CheckItem, Menu3, M3Item, True);
 SelectedFont := FontItemMap[M3Item];
end;

procedure DoStyleItem;
begin
 M4Toggle[SelectedItemID] := not M4toggle[SelectedItemID];
 InLineP(CheckItem, Menu4, SelectedItemID,
      M4Toggle[SelecteditemID]);
 M4Toggle[M4Item1] := False;
 InLineP(CheckItem, Menu4, M4Item1, False);
end;

procedure DoMenu4;
 var
  index : integer;
begin
 case SelectedItemID of
  M4Item1 :
   begin
    SelectedStyle := Normal;
    for index := 2 to 6 do
     begin
      M4Toggle[index] := False;
      InLineP(CheckItem, Menu4, index, False);
     end;
   end;
  M4Item2 :
   begin
    DoStyleItem;
    if M4Toggle[SelectedItemID] = True then
     SelectedStyle := SelectedStyle + Bold
    else
     SelectedStyle := SelectedStyle - Bold;
   end;
  M4Item3 :
   begin
    DoStyleItem;
```

Continued

Listing 9.7 *Continued*

```
  if M4Toggle[SelectedItemID] = True then
   SelectedStyle := SelectedStyle + Italic
  else
   SelectedStyle := SelectedStyle - Italic;
 end;
M4Item4 :
 begin
  DoStyleItem;
  if M4Toggle[SelectedItemID] = True then
   SelectedStyle := SelectedStyle + Underline
  else
   SelectedStyle := SelectedStyle - Underline;
 end;
M4Item5 :
 begin
  DoStyleItem;
  if M4Toggle[SelectedItemID] = True then
   begin
    if M4Toggle[M4Item6] = True then
     begin
      SelectedStyle := SelectedStyle - Shadow;
      M4Toggle[M4Item6] := False;
      InLineP(CheckItem, Menu4, M4Item6, False);
     end;
    SelectedStyle := SelectedStyle + Outline;
   end
  else
   SelectedStyle := SelectedStyle - Outline;
 end;
M4Item6 :
 begin
  DoStyleItem;
  if M4Toggle[SelectedItemID] = True then
   begin
    if M4Toggle[M4Item5] = True then
     begin
      SelectedStyle := SelectedStyle - Outline;
      M4Toggle[M4Item5] := False;
      InLineP(CheckItem, Menu4, M4Item5, False);
```

Continued

Listing 9.7 *Continued*

```
        end;
       SelectedStyle := SelectedStyle + Shadow;
      end
     else
      SelectedStyle := SelectedStyle - Shadow;
    end;
   otherwise
 end;
 if selectedStyle = Normal then
  begin
   M4Toggle[1] := True;
   InLineP(CheckItem, Menu4, M4Item1, True);
  end;
end;

procedure HandleMenu;
 var
  SelectedMenu : LongInt;
  Ch : Char;
begin
 EventInfo.Message := LInLineF(BitAnd, EventInfo.Message, $FF + 0);
 Ch := Chr(EventInfo.Message);
 if EventInfo.What = MouseDown then
  SelectedMenu := LInLineF(MenuSelect, EventInfo.Where)
 else
  SelectedMenu := LInLineF(MenuKey, Ch);
 SelectedMenuID := WInLineF(HiWord, SelectedMenu);
 SelectedItemID := WInLineF(LoWord, SelectedMenu);
 case SelectedMenuID of
  Menu1ID :
   DoMenu1;
  Menu2ID :
   DoMenu2;
  Menu3ID :
   DoMenu3;
  Menu4ID :
   DoMenu4;
  otherwise
 end;
```

Continued

Listing 9.7 *Continued*

```
 InLineP(HiLiteMenu, 0);
end;

procedure HandleContent (WindowIn : WindowPtr);
begin
 InLineP(SelectWindow, WindowIn);
 OnOffM2Items;
end;

procedure HandleDrag (WindowIn : WindowPtr);
begin
 InLineP(DragWindow, WindowIn, EventInfo.Where, GlobalBounds);
end;

procedure HandleGrow (WindowIn : WindowPtr);
 var
  WSize : LongInt;
  OldPort : WindowPtr;
  VBar, HBar : Rectangle;
begin
 VBar := GetVertBar(WindowIn);
 HBar := GetHorizBar(WindowIn);
 InLineP(GetPort, OldPort);
 InLineP(SetPort, WindowIn);
 WSize := LInLineF(GrowWindow, WindowIn, EventInfo.Where, MaxWSize);
 InLineP(InvalRect, VBar);
 InLineP(InvalRect, HBar);
 InLineP(SizeWindow, WindowIn, WSize, True);
 VBar := GetVertBar(WindowIn);
 HBar := GetHorizBar(WindowIn);
 InLineP(InvalRect, VBar);
 InLineP(InvalRect, HBar);
 InLineP(SetPort, OldPort);
end;

procedure HandleGoAway (WindowIn : WindowPtr);
 var
  WindowInID : integer;
begin
```

Continued

Listing 9.7 *Continued*

```
 if BInLineF(TrackGoAway, WindowIn, EventInfo.Where) = True then
  begin
   WindowInID := GetWindowID(WindowIn);
   InLineP(HideWindow, WindowIn);
   M1Toggle[WindowInID] := False;
   case WindowInID of
    Window1ID :
     InLineP(SetItem, Menu1, M1Item1, 'Show Window 1');
    Window2ID :
     InLineP(SetItem, Menu1, M1Item2, 'Show Window 2');
    Window3ID :
     InLineP(SetItem, Menu1, M1Item3, 'Show Window 3');
    Window4ID :
     InLineP(SetItem, Menu1, M1Item4, 'Show Window 4');
    otherwise
   end
  end;
end;

procedure HandleMouse;
 const
  InDesk = 0;
  InMenuBar = 1;
  InSysWindow = 2;
  InContent = 3;
  InDrag = 4;
  InGrow = 5;
  InGoAway = 6;
 var
  AWindow : WindowPtr;
  WhereItIs : Integer;
begin
 WhereItIs := WInLineF(FindWindow, EventInfo.Where, @AWindow);
 case WhereItIs of
  InMenuBar :
   HandleMenu;
  InContent :
   HandleContent(AWindow);
  InDrag :
```

Continued

Listing 9.7 *Continued*

```
    HandleDrag(AWindow);
   InGrow :
    HandleGrow(AWindow);
   InGoAway :
    HandleGoAway(AWindow);
   otherwise
    InLineP(SysBeep, 5);
 end
end;

procedure HandleUpdate;
 var
  UpWindow : WindowPtr;
begin
 UpWindow := Pointer(EventInfo.Message);
 InLineP(BeginUpDate, UpWindow);
 WriteIt(UpWindow);
 InLineP(EndUpDate, UpWindow);
 InLineP(DrawGrowIcon, Upwindow);
end;

procedure HandleActivate;
 var
  ActWindow : WindowPtr;
begin
 ActWindow := Pointer(EventInfo.Message);
 EraseScrollBars(ActWindow);
 InLineP(DrawGrowIcon, ActWindow);
 OnOffM2Items;
end;

procedure HandleKey;
begin
 if LInLineF(BitAnd, EventInfo.Modifiers + 0, CmdKey + 0) = CmdKey
      then
  HandleMenu
 else
  InLineP(SysBeep, 5)
end;
```

Continued

Listing 9.7 *Continued*

```
{The Outer Loop}
begin
 initialize;
 begin
  repeat
  {wait for an event, then handle it}
   if BInLineF(GetNextEvent, EventMask, @EventInfo) then
    case EventInfo.What of
     MouseDown :
      HandleMouse;
     KeyDown :
      HandleKey;
     UpdateEvt :
      HandleUpdate;
     ActivateEvt :
      HandleActivate;
     otherwise
    end
  until Abort;
  InLineP(SetMenuBar, OldMenuBar);
  InLineP(DrawMenuBar);
  InLineP(DisposeMenu, Menu1);
  InLineP(DisposeMenu, Menu2);
  InLineP(DisposeMenu, Menu3);
  InLineP(DisposeMenu, Menu4);
  InLineP(CloseWindow, Window1);
  InLineP(CloseWindow, Window2);
  InLineP(CloseWindow, Window3);
  InLineP(CloseWindow, Window4);
 end
end.
```

CHAPTER

10 GETTING INTO THE HARDWARE

Nuts and Bolts

Chips and Such

Memory

The Display

Peripherals and Interfaces

Interrupts

The 6522

The Keyboard

The Mouse

The Sound Generator

The Printer

NUTS AND BOLTS

There are no earthshaking innovations in the Macintosh package; it's just a well-thought-out and well-executed design. It has one of the smallest footprints in the desktop computer arena. It would be hard to find a quieter computer since, like the Apple II, it uses convection cooling rather than a fan.

Unlike other Apple computers and most other microcomputers, the Macintosh is designed to be an appliance and, as such, is a closed system. It is not easily expanded. It has no expansion slots like the Apple II or IBM PC. The only way a user can add devices is by attaching them to one of the serial ports. Even though Apple designed the system to discourage add-on devices that reside inside the Macintosh case, at least one company has introduced a Winchester disk drive that sits inside the Macintosh. Others will probably follow.

There are only two circuit boards in the Macintosh—the analog board (containing the power supply, speaker, and display electronics) and the main circuit board (containing the processor chip, memory, and all of the other digital circuits). The analog board is mounted vertically on the left side of the machine under the cooling slots. If you put your hand on top, you can feel the heat coming from the power supply on the analog board. The power supply is a switching supply that produces +5 volts, +12 volts, and −12 volts for the Mac.

The video display produces white images rather than green or amber, and the Macintosh software draws black images on a white background. If you are interested in photographing images on the CRT, you will find some useful information on color balance and other photographic subjects in *The Apple Macintosh Book* by Carry Lu.

CHIPS AND SUCH

The heart of the Macintosh is a Motorola 68000 processor running at a clock speed of 7.78 megahertz (MHz). The 68000 has 32-bit registers and a 16-bit-wide external data bus. For more information about the 68000, see chapter 11.

Since the 68000 reads and writes 16 bits (2 bytes) of data at a time, the memory in the Macintosh is 16 bits wide. The Macintosh currently comes in two memory sizes, 128K bytes and 512K bytes. The 128K version uses sixteen RAM (random access memory) chips that store 64K bits each; the 512K version uses sixteen RAM chips that store 256K bits each, giving it four times the memory of the 128K version. In both cases, a single row of

sixteen memory chips resides on the main circuit board. The memory does not have parity bits.

The Mac also has 64K bytes of ROM (read-only memory) in two chips that store 32K bytes each. The ROM chips contain the toolbox routines used by the operating system and application programs.

Three chips are used as I/O controllers—the 8530 serial communications controller, the 6522 interface chip, and a custom chip containing the disk controller. The 8530 controls two serial communications ports, the printer port and the modem port. The 6522 provides two timers, a shift register and an interface for several interrupts.

The Macintosh makes extensive use of programmable array logic (PAL) to shrink the number of chips on the main circuit board and thus the size and cost of the board. The Macintosh's designers used a PAL chip wherever they could find combinations of digital logic circuits that could be incorporated into one.

The time-of-day clock is derived from an interrupt provided by the 6522 chip. The 6522 is a CMOS chip that uses very little power; it continues to run from the power of a small battery when the Macintosh is turned off. Other timing information in the Macintosh—for processor bus operations and the video display circuits—is derived from the processor clock, which operates at 7.78 MHz.

MEMORY

The Macintosh has either 128K bytes or 512K bytes of RAM, but there's more than RAM in the Mac's address space. There are the 64K bytes of ROM that we already mentioned and all of the I/O devices. The 68000 does not have a separate I/O bus. All devices are on the memory bus; the processor just reads or writes to their memory locations, instead of executing I/O instructions. The 68000 has a 24-bit-wide address bus, which enables the processor to address up to 16 million bytes of memory. The Macintosh uses only about 4 of that 16 megabytes of address space. The RAM, ROM, and I/O devices each have their own assigned portion of the address space (figure 10.1).

The Mac uses RAM for storing programs and data, and for the video display. The display circuits read the portion of memory set aside for the video display every time they update the display, about 60 times a second. Access to the video display section of RAM is actually shared between the 68000 and the display. The display requires about 22K RAM.

Apple II programmers will not be surprised to find two sections of RAM that can be used by the display. A program can store an image

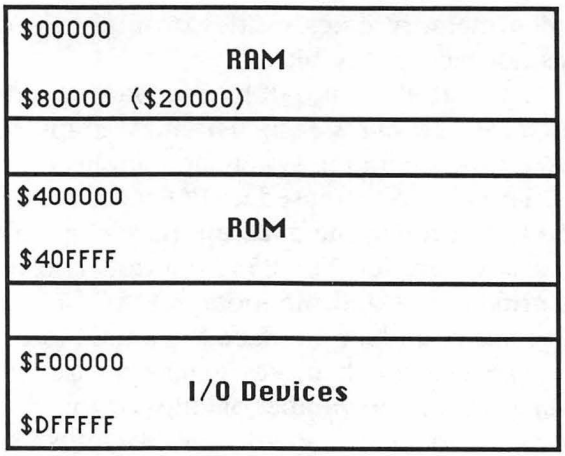


Figure 10.1 Address map

in the primary display area or the secondary display area. It can have images in both areas and switch between them.

Another portion of RAM is dedicated as a sound buffer. The sound generator circuits read data from the sound buffer much the same way that the video circuits read data from the display area. Like the video circuits, the sound generator can get data from two sound buffers (figure 10.2).

The processor and operating-system software use dedicated portions of memory for specific purposes. The starting addresses of various areas of memory are contained in system pointers. The memory boundary constants and their corresponding pointers are listed below:

Name	*Meaning*
SysZone	System heap
ApplZone	Application heap
HeapEnd	End of application heap
CurStackBase	End of stack
BufPtr	End of application parameters
ScrnBase	Start of current screen buffer
MemTop	End of RAM

The first section of memory is the system globals area, where the operating system keeps global variables (the system dispatch table and the exception vectors) that are used for communication among different parts of the operating system.

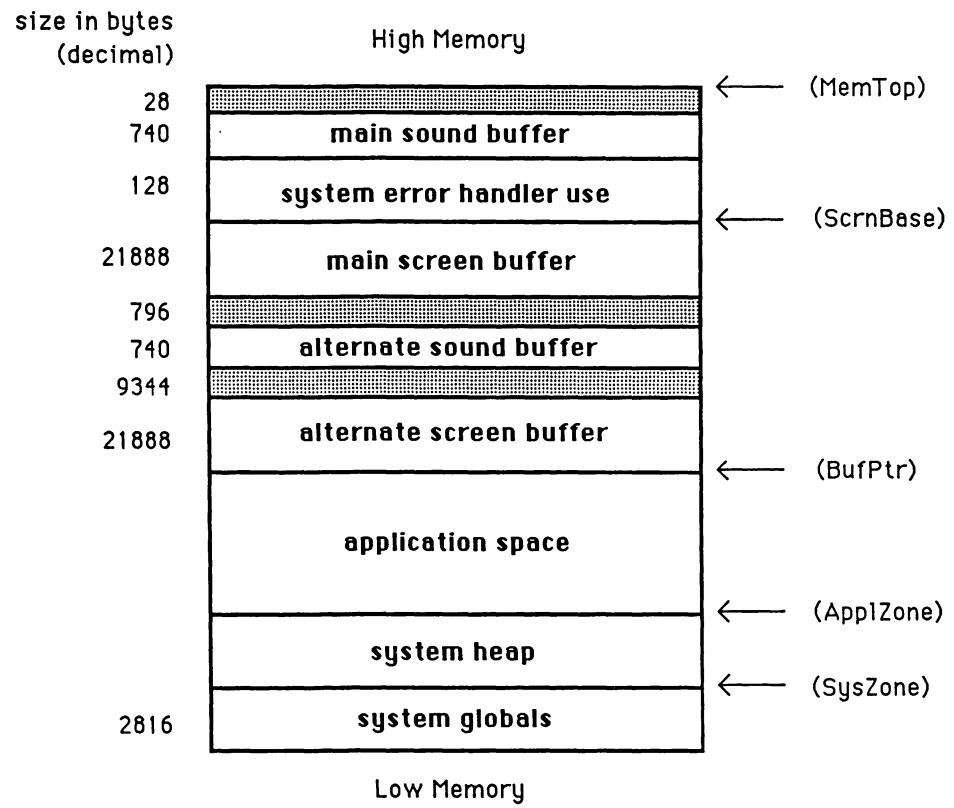


Figure 10.2 RAM map

The 68000 processor requires that the first 512 bytes of memory be dedicated to exception vectors. Exception vectors are pointers to routines that handle interrupts and exceptional events (for more information on exception and interrupt processing, see chapter 11).

The next part of the system globals area is the system dispatch table, used to find the location of toolbox routines. Accessing toolbox routines through a dispatch table allows programs to use the toolbox without knowing which version of the ROM code they are using. It also allows the operating system to replace some toolbox routines with newer routines that are located in RAM.

The dispatch table contains 512 one-word (16-bit) entries. Thus, it can access up to 512 toolbox routines. How can the dispatch table squeeze a 24-bit address into a 16-bit word? By a little magic with a base and displacement.

The one-word dispatch-table entry contains 14 bits of displacement and a 1-bit flag that indicates whether the displacement is relative to the start of the ROM or the start of the area of RAM used for toolbox routines (the system heap). The system gets the base address from a global variable, and adds the displacement to create the memory address of the toolbox routine.

A program calls a toolbox routine through a mechanism called a *trap*. The program codes a trap number as if it were an instruction. A trap number is a 16-bit number that begins with $A (hex A). Since no legal 68000 instructions begin with $A, the processor generates an illegal instruction exception (interrupt). The exception handler uses the remaining 12 bits of the trap number to identify which toolbox or operating-system routine is being called, and accesses the routine via the dispatch table.

The system global variables follow the system dispatch table.

Many system routines require a mechanism for dynamically allocating blocks of memory. The system allocates memory blocks from the system heap. Those blocks of memory can contain executable code or data. Because the routine requesting a memory allocation has no idea where that block will be allocated, it addresses the block through a pointer.

Since some types of blocks can be relocated by the memory manager, the program requesting the block must access it through a pointer. If the memory manager moved the block, every location in the program that used the block would need its pointer updated. The Macintosh circumvents that problem by addressing dynamically allocated blocks of memory through a handle, a pointer to a pointer. There is only one pointer to a relocatable block, and all references to the block use a pointer to that pointer. When a block moves, only the single pointer to the block need be updated.

At the end of memory, several sections of RAM are dedicated to the primary and alternate display buffers and sound buffers. Usually a program needs to use only the primary display and sound buffers, so the program is allowed to use the alternate buffer areas for its own purposes.

The physical addresses of the display and sound buffers differ in the 128K and 512K versions of the Macintosh.

After about 20K bytes of memory are dedicated to the operating system and another 22K bytes are set aside for the display and sound buffers, there are about 68K bytes left for an application program in a 128K Macintosh. A 512K Macintosh has about 450K bytes left for an application program. It has four times the memory of the 128K version, but it can run an application program five times the size of an application for the 128K Mac.

The system also divides up the area of memory used by an application program. The application program has areas set aside for its globals just as the system does. The application also has a heap, similar to the system heap. It is an area used for the dynamic allocation of blocks of memory for the program. The application program's code and data both reside in the application heap (figure 10.3).

The stack starts just below the QuickDraw globals area and grows down. The heap is located at the start of the application area and grows up. The system periodically checks to see if the heap and stack have collided and issues an error message if they have. In the next chapter, we will see that the 68000 maintains two stack pointers, one for use by a supervisor or operating system and the other for use by an application program. The Macintosh does not use that capability. It has only one stack, which is used by both the system and the application program.

The QuickDraw globals area is just above the application stack. It contains global variables used by the QuickDraw routines.

The application globals area is next; it contains global variables used by the application program. Following that is the application parameters

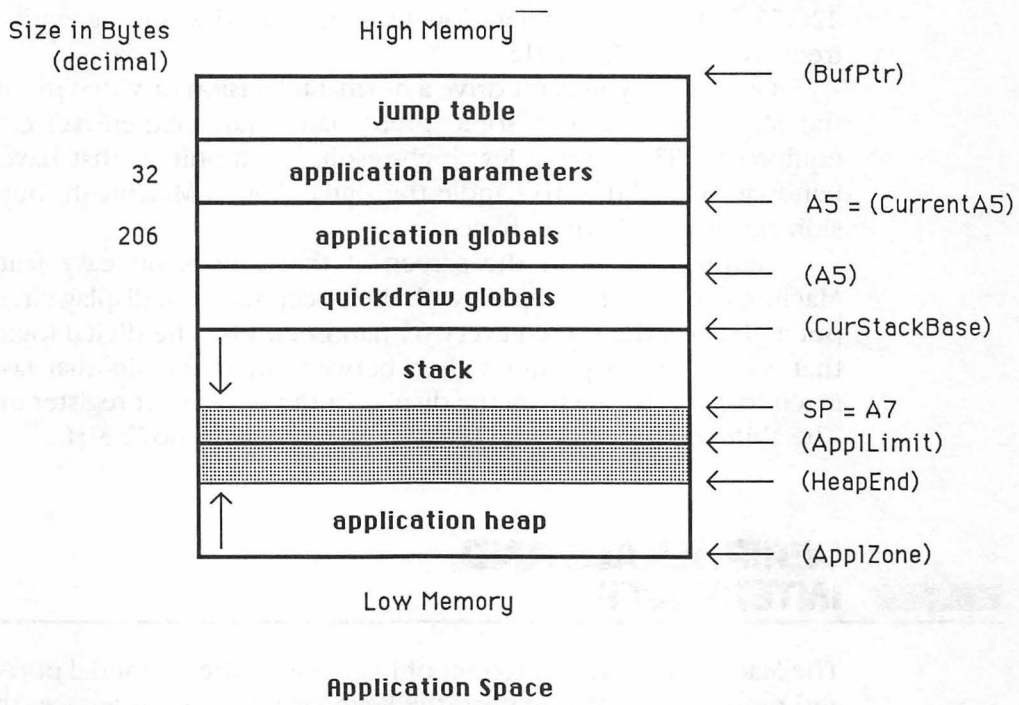


Figure 10.3 Application program memory map

area that contains parameters used for communications between the Finder and the application program.

The last area of the application program's memory contains the jump table. The segment loader creates and maintains the jump table, which contains pointers to code and data segments that the segment loader has loaded into the application heap area. When a portion of a program tries to access something outside of its segment, the access is done via the jump table. The segment loader takes care of loading segments as they are needed and maintaining the jump table. The application program itself doesn't need to know anything about segmentation or the jump table.

THE DISPLAY

The CRT display uses a 9-inch (diagonal) tube and video circuitry that displays graphics with a resolution of 512 vertical pixels by 342 horizontal pixels—about 72 pixels per inch. The frame rate is 60.15 Hz (not the standard 60 Hz used by television and other video equipment). The display uses noninterlaced video with a horizontal sweep frequency of 22.255 KHz; normal television uses interlaced scanning with a sweep frequency of 15.738 KHz.

Obviously, you can't drive a normal television or video monitor with the Macintosh without some pretty fancy (and expensive) conversion equipment. There are a few high-resolution monitors that have enough bandwidth (20 MHz) to handle the signal from a Macintosh, but a television set doesn't have a chance.

Getting pixels on the screen at that rate is no easy feat for the Macintosh either. To keep up with the sweep rate, the display circuits must put a pixel on the screen every 65 nanoseconds. The digital logic circuits that access memory can't switch between memory bits that fast, so the Macintosh loads data from the display buffer into a shift register in parallel. The shift register shifts the bits out in serial at 15.6672 MHz.

PERIPHERALS AND INTERFACES

The Macintosh has a limited set of I/O devices: the two serial ports (printer and modem), the mouse port, the keyboard, the disk interface, the sound generator, and the programmer's switch (a set of two push buttons that attach to the left side of the Macintosh). See figure 10.4.

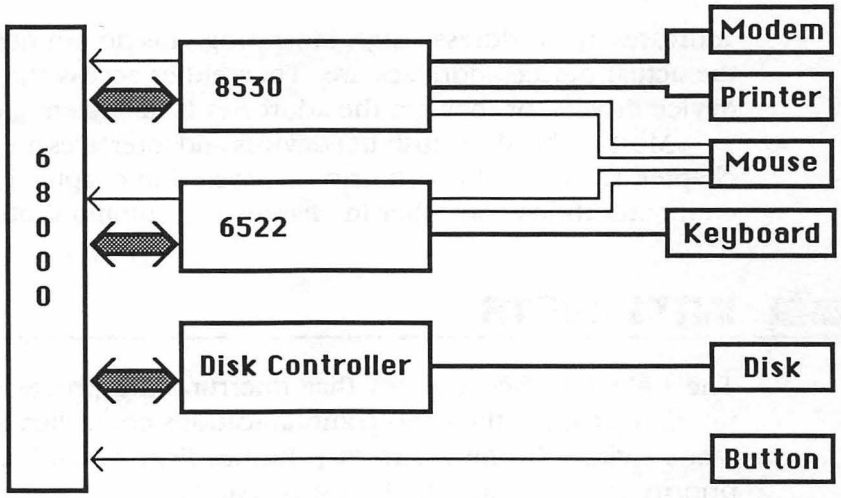


Figure 10.4 Devices and controllers

The mouse button, time-of-day clock, and keyboard use the 6522 interface chip. The mouse-position signals use a portion of the 8530 communications chip. Only four memory regions are assigned to I/O devices, and two of them are for the 8530 communications chip (figure 10.5).

The devices don't actually use all of the memory address space allocated to them. The 8530, for instance, uses only the top four even

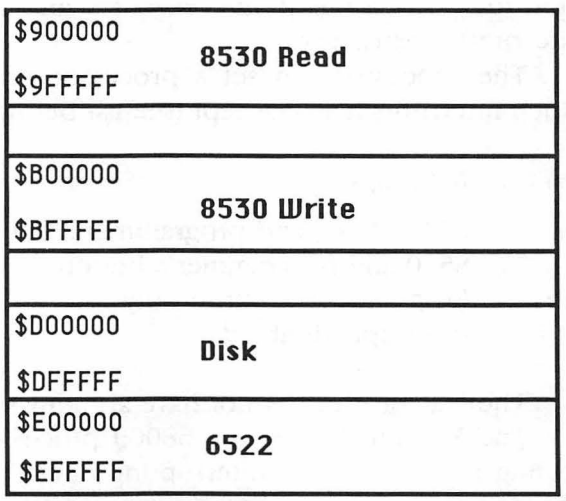


Figure 10.5 Device address map

addresses in its address range. Most programs do not need to know what the actual device addresses are. They either access the devices through device drivers, or they get the addresses from system global variables.

Most of the Macintosh I/O devices and interfaces are described in this chapter. However, the disk drive is covered in chapter 13, Disks, and the communications controller in chapter 14, Communicating.

INTERRUPTS

The only interface devices that interrupt the processor are the 6522 interface adapter, the 8530 communications controller, and the programmer's switch. In the interrupt priorities list (shown below), the highest priority is level 6 and the lowest is level 0.

Priority Level	*Device*
0	None
1	6522
2	8530
3	6522 and 8530
4–6	Programmer's button

A level 3 interrupt is a result of the way the interrupt signals are wired to the 68000 processor. This interrupt occurs if a 6522 and 8530 interrupt both occur at the same time. The level 3 interrupt handler does no interrupt processing; it just executes an RTE—return from exception (interrupt)—instruction.

The processor can set a processor priority level that determines which interrupts it will accept (see list below).

Level	*Interrupts*
0	6522, 8530, and programmer's button
1	8530 and programmer's button
2–6	Programmer's button only
7	Interrupts disabled

The Macintosh does not have an unmaskable (level 7) interrupt.

The Macintosh uses the 68000 processor's autovector process for fielding interrupts. The interrupting device doesn't need to supply an interrupt vector to the processor to identify itself. The interrupt level identifies both the interrupting device and the vector that will take the processor to the proper interrupt handler.

The 8530 and 6522 can generate interrupts for a variety of reasons. The interrupt handler for each of these devices interrogates the device to find the cause of the interrupt, and uses an additional vector to access the routine that will do the interrupt processing.

The 8530 has two communications channels. Each generates interrupts for several reasons, so the interrupt handler uses a secondary interrupt vector table to point to the proper routine to handle the specific type of 8530 interrupt. One of the 8530 interrupts can occur for two different reasons, a status change or an external input. The 8530 interrupt handler uses a third interrupt vector table to call a routine to handle the external/status change interrupt. The secondary and external/status change interrupt tables are shown below:

Secondary Vectors

Channel B transmit buffer empty
Channel B external/status change
Channel B receive character available
Channel B special receive condition
Channel A transmit buffer empty
Channel A external/status change
Channel A receive character available
Channel A special receive condition

External/Status Change Vectors

Channel B status change
Channel B external (mouse vertical motion)
Channel A status change
Channel A external (mouse horizontal motion)

The 6522 interrupt handler reads the 6522's interrupt flag register (IFR) and interrupt enable register (IER) to find out what caused the interrupt, and goes to a secondary interrupt vector table (shown below) to call the routine to handle the interrupt.

6522 Interrupts

One-second timer
Vertical retrace
Shift register (keyboard/mouse)
T1 timer (sound driver timer)
T2 timer (disk drive timer)

The one-second timer supplies the time base for the time-of-day clock. The one-second-timer interrupt handler updates the system time (a system global variable) and flashes selected menu items.

The video display circuitry generates an interrupt at the beginning of the vertical retrace interval, and the system uses it as a high-resolution timer (one interrupt every 1/60 second). When the electron beam that writes on the CRT screen reaches the bottom of the screen, it must travel back to the top to begin scanning the screen again. The time interval occupied by the beam's traveling back to the top of the screen is called the vertical retrace interval. The vertical-retrace interrupt handler updates a global variable that contains the number of time ticks since the system was turned on. The toolbox routine TickCount returns the value of that variable.

The vertical retrace interrupt is also useful to programs that want to update the display during the retrace interval. (When changes to the display are made during the retrace interval, the display does not flicker.) The vertical-retrace interrupt handler maintains a queue of routines to execute when the interrupt occurs. A routine can be added to that queue with a call to the vertical retrace manager.

The vertical-retrace interrupt handler is responsible for:

- Updating the system global TickCount
- Checking for stack/heap collisions
- Moving the cursor
- Checking for changes in the state of the mouse button
- Checking for disk insertions
- Executing routines in the vertical-retrace manager queue

The vertical-retrace interrupt handler is the routine that checks to see if the stack has collided with the application heap (a fatal error). It also handles cursor movement and checks for changes in the state of the mouse button. It checks the mouse button only every other interrupt. If the button has changed and remained at its new state for four interrupts, the handler posts a mouse event.

Every thirty interrupts (every 1/2 second) the vertical-retrace interrupt handler checks to see if a disk has been inserted and, if so, generates a disk insertion event.

THE 6522

The 6522 is an interface chip usually called the Versatile Interface Adapter (VIA) in Apple's documentation. It is a CMOS device that uses very little power and can run on a battery while the Macintosh is turned off.

The 6522 has two bidirectional, 8-bit parallel data ports; an 8-bit shift register for serial data transfers; and two 16-bit timers (T1 and T2).

Both of the 8-bit parallel ports have handshake and interrupt lines. The bits in these ports can be set individually for input or output. The shift register can shift in data under the control of the system clock signal or an external clock signal. The Macintosh uses the shift register for input from the keyboard.

The timers operate by counting down from a preloaded count. When the timer count reaches zero, the timer generates an interrupt. Timers can generate interrupts while operating in free-running mode; they do not need to wait for a new timer value to be loaded after an interrupt. They have the old time count stored in a latch and begin a new time count immediately, using the old count. Interrupt-processing delays do not affect the time interval between 6522 timer interrupts.

A timer may also operate in one-shot mode—that is, it may generate a single interrupt and wait for the processor to load another time value before starting again.

The 6522 has a set of sixteen registers that control its operation and hold data that is being written to or read from external devices (table 10.1).

The T1 timer operates in one-shot or free-running mode. The T2 timer operates in one-shot mode or as a pulse counter, counting the pulses that appear on pin B6. In one-shot mode, the timer starts when you write the high-order half of the count value into the latch or the counter. As the timer runs, it decrements the count in the counter. When the counter reaches zero, the 6522 generates an interrupt. To start the timer again, you must write a new count value to the timer.

When you operate T1 in free-running mode, you do the same thing, except when the count reaches zero, the 6522 automatically loads the counter from the latch again and starts the timer again without a pause (figure 10.6).

The auxiliary control register controls the operation of the timers and other parts of the 6522 (figure 10.7).

The functions of the control bits in the shift register and timers are illustrated in tables 10.2 through 10.4.

The T2 timer in one-shot mode works just like the T1 timer in one-shot mode. The T2 pulse-counter mode is similar to the one-shot mode; the only difference is that, instead of decrementing the count every clock pulse, the 6522 decrements the count every time there is a negative pulse on the PB6 pin.

The shift register can be clocked by an external clock signal on pin B1 (the shift clock pin), by the system clock signal, or by the system clock signal divided by the count value in the low-order byte of T2. When

Table 10.1 6522 Registers

Number	*Name*	*Read/Write*	*Description*
0	ORB	Write	Parallel port B output data
0	IRB	Read	Parallel port B input data
1	ORA	Write	Parallel port A output data
1	ORB	Read	Parallel port A input data
2	DDRB	Both	Parallel port B data direction
3	DDRA	Both	Parallel port A data direction
4	T1C-L	Write	T1 low-order latch
4	T1C-L	Read	T1 low-order counter
5	T1C-H	Both	T1 high-order counter
6	T1L-L	Both	T1 low-order latches
7	T1L-H	Both	T1 high-order latches
8	T2C-L	Write	T2 low-order latches
8	T2C-L	Read	T2 low-order counter
9	T2C-H	Both	T2 high-order counter
10	SR	Both	Shift register
11	ACR	Both	Auxiliary control register
12	PCR	Both	Peripheral control register
13	IFR	Both	Interrupt flag register
14	IER	Both	Interrupt enable register
15	ORA	Write	Parallel port A output data (without handshake)
15	IRA	Read	Parallel port A input data (without handshake)

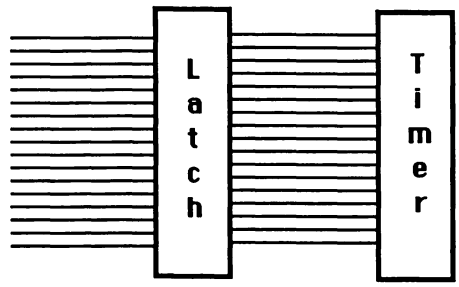


Figure 10.6 T1 latch and counter

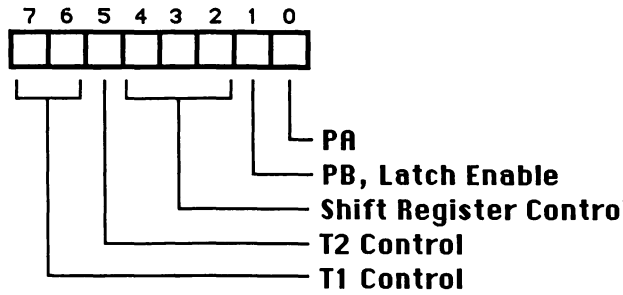


Figure 10.7 Auxiliary control register

Table 10.2 Shift Register Control Bits

Bit 4	*Bit 3*	*Bit 2*	*Function*
0	0	0	Shift register is disabled.
0	0	1	T2 controls shifting (shift in).
0	1	0	System clock controls shifting (shift in).
0	1	1	External clock controls shifting (shift in).
1	0	0	Shift at T2 rate free-running (shift out).
1	0	1	T2 controls shifting (shift out).
1	1	0	System clock controls shifting (shift out).
1	1	1	External clock controls shifting (shift out).

Table 10.3 Timer T1 Control Bits

Bit 7	*Bit 6*	*Function*
0	0	Loading T1 starts the timer. Timer interrupts when it reaches zero. No output on pin B7.
0	1	The timer runs free and interrupts each time it reaches zero. No output on pin B7.
1	0	Loading T1 starts the timer. Timer interrupts when it reaches zero. Pin B7 has a single-pulse output every time the timer reaches zero.
1	1	The timer runs free and interrupts each time it reaches zero. Pin B7 has a square-wave output.

Table 10.4 Timer T2 Control Bits

Bit 5	*Function*
0	T2 is an interval timer.
1	T2 counts pulses on pin B6.

shifting is not controlled by an external clock, the 6522 provides an internally generated shift clock signal to the external device on pin B1. The serial data being shifted appears on pin B2.

The data-direction registers determine whether each individual bit in the parallel ports operates as an input or an output. The parallel data ports may operate latched or unlatched. In latched operation, reading the data register gets you the state of the input pins when the last transition of the handshake signal occurred. In unlatched operation, reading the data register gets you the current state of the input pins.

Parallel data port B operates in a slightly different manner. When you read port A, the values you get for bits programmed as output bits are the values on the data lines. If an external device has enough current-sinking capacity to pull down a data line, that's what you see. When you read port B, the values you get for pins programmed for output are the values in the output register, not the actual data on the data lines.

Another difference between port A and port B is in handshaking with an external device. Port A will do handshaking on a read or write operation. Port B handshakes only when writing to an external device.

When the 6522 is using handshaking to control a read operation, it expects the external device to provide a signal on the A1 or B1 line, indicating that valid data exists on the data lines. The "valid data" signal causes an interrupt cycle in the 6522. The 6522 then reads data from the data lines and asserts the A2 or B2 signal to indicate that it has read the data. The external device is expected to keep the data on the data lines until the 6522 asserts the A2 or B2 signal (figure 10.8).

When writing data, the 6522 asserts a "data ready" signal on the A1 or B1 line; the external device then reads the data lines and replies with the "data accepted" signal on A2 or B2.

The 6522 interrupts the 68000 processor under the control of the interrupt enable register (IER) in the 6522. When an event occurs that generates a 6522 interrupt—for instance, a parallel-data-port handshake—an interrupt flag is set in the 6522 interrupt flag register (IFR). This flag remains set until the 68000 services the interrupt and clears the interrupt flag. If the corresponding bit is on in the 6522's interrupt enable register, it requests a 68000 interrupt. If that bit is off in the 6522's interrupt enable register, the interrupt flag stays on and the interrupt remains pending in the 6522 until it is enabled. The 68000 clears an interrupt flag by writing a 1 to the flag's bit in the 6522 interrupt flag register.

THE KEYBOARD

The keyboard has fifty-eight keys and contains an 8021 microprocessor to scan the keys and send data serially to the 6522 shift register. The keyboard

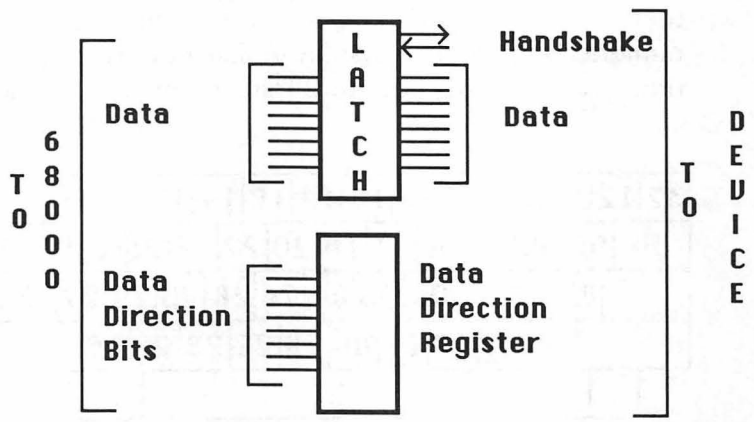


Figure 10.8 Parallel data ports

does a two-key rollover; that is, if you press two keys and hold both of them down at the same time, the keyboard processor remembers which one you pressed first and reports that keystroke first. The processor doesn't lose the second keystroke; it reports that one second. If any additional keys are pressed while the first two are still down, those keystrokes are lost.

The keyboard produces a unique hex code for each key, not an ASCII character. The keyboard handler in the Macintosh converts the hex codes into ASCII character codes. The numeric keypad works the same way; it also produces hex key codes that the keyboard handler maps into ASCII character codes. The numeric keypad produces hex codes different from those produced by the main keyboard. The fact that two keys are labeled the same means nothing to the keyboard and numeric keypad hardware (figure 10.9).

When you press a key on the keyboard, that action is eventually reported to an application program by the event manager. The event record contains the hardware hex key code generated by the keyboard, the ASCII character code supplied by the keyboard handler, and a set of modifier bits that report which modifier keys were down when you pressed the key (figure 10.10). The keyboard handler uses the keyboard configuration to convert hardware hex key codes into ASCII characters. The configuration is a resource file that contains a translation table and an assembly language subroutine to perform the translation. You can change the keyboard configuration by supplying a new resource file. There's also a somewhat tricky way to change the keyboard configuration at runtime. You use an assembly language routine to change the pointer to the key-translation subroutine, and then substitute your own subroutine.

We've been talking about one keyboard configuration routine, but there are actually two—one for the main keyboard and one for the numeric keypad. The system global Key1Trans points to the keyboard-translation routine, and Key2Trans points to the numeric-keypad transla-

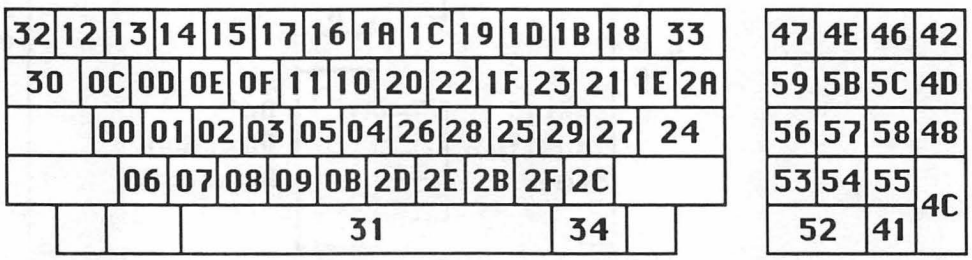


Figure 10.9 Keyboard and keypad codes

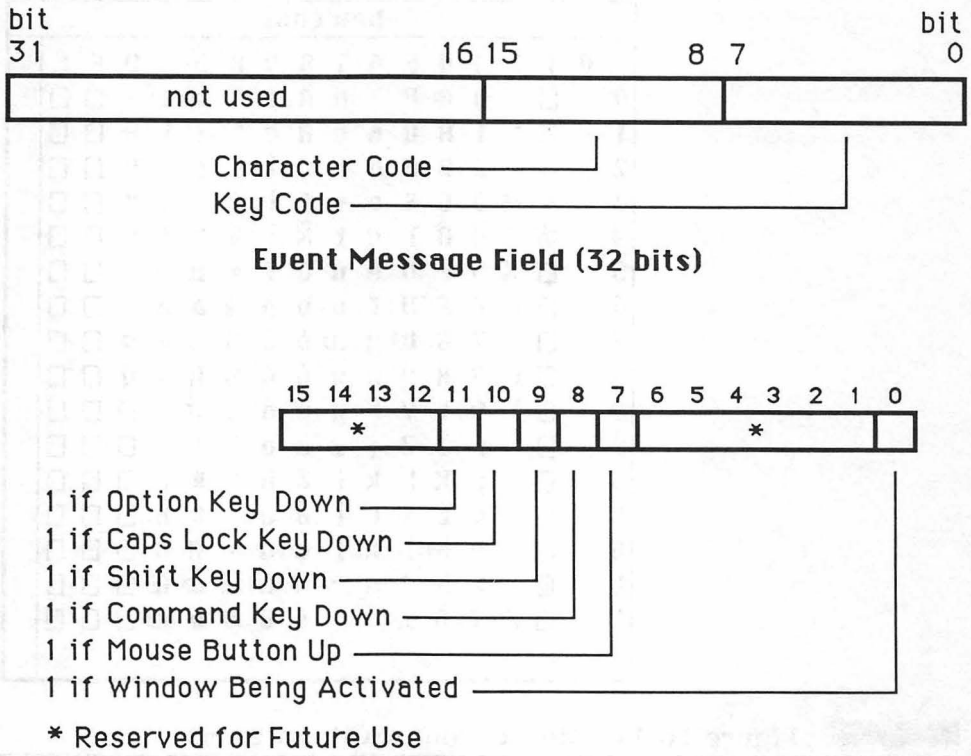


Figure 10.10 Keyboard event data

tion routine. A translation routine gets the modifier bits and hex key code as input and returns the ASCII character code.

The characters displayed for each ASCII code are, for the most part, the same, regardless of the type font you are using. There are a few notable exceptions, though—the most obvious being the Cairo font that doesn't display characters at all. Most of the character fonts produce the same characters for the same ASCII codes, but one or two may vary. Note the characters displayed for an ASCII character value of $D9 in the ASCII character tables (figures 10.11 and 10.12).

Since most fonts support the same character set, we can use one set of keyboard diagrams to show which characters you can produce with the modifier keys—shift, caps lock, and option (see figures 10.13 through 10.16). Note that we did not include a keyboard diagram for the command key. When you use the command key, the key codes are not translated into ASCII characters but are used to execute commands.

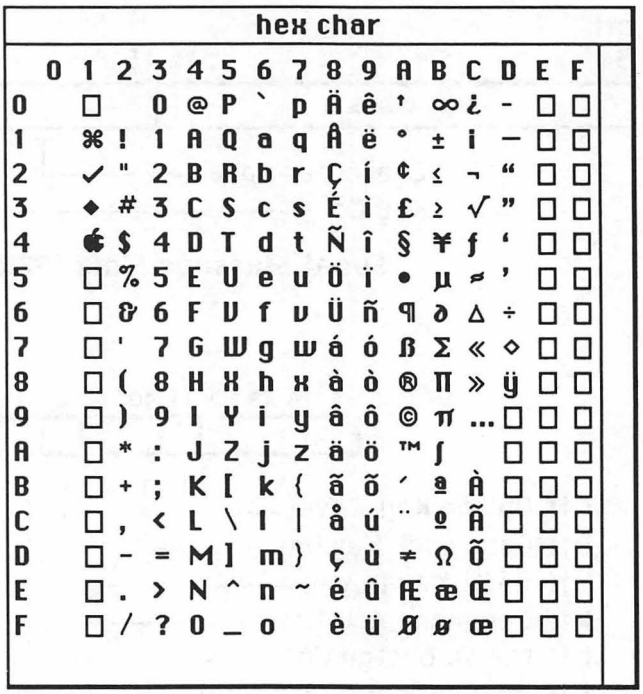


Figure 10.11 Monaco font ASCII characters

THE MOUSE

The mouse contains two optical shaft encoders to sense movement. The mouse ball rests on three wheels: one is an idler wheel just for support, and the other two are attached to shafts. When you move the mouse across a flat surface, the motion of the ball is transferred to the wheels that turn the shafts. The two shafts are at right angles to each other so that one turns with vertical mouse motion and the other turns with horizontal mouse motion (figure 10.17).

As a shaft turns, it turns the shaft encoder. The shaft encoder generates a pulse for every 3.54 millimeters of mouse movement. That gives the mouse the ability to report its movement at a resolution of about 90 positions per inch.

Each of the optical shaft encoders has two outputs. By comparing the signals from both outputs, the Macintosh tells what direction the mouse has moved. Since there are two shaft encoders, one for the x direction and one for the y direction, the Macintosh gets four position signals from

hex char

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		□		0	@	P	`	p	Ä	ê	†	∞	¿	-	□	□
1		□	!	1	A	Q	a	q	Å	ë	°	±	¡	–	□	□
2		□	"	2	B	R	b	r	Ç	í	¢	≤	¬	"	□	□
3		□	#	3	C	S	c	s	É	ì	£	≥	√	"	□	□
4		□	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	'	□	□
5		□	%	5	E	U	e	u	Ö	ï	•	µ	≈	'	□	□
6		□	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷	□	□
7		□	'	7	G	W	g	w	á	ó	ß	Σ	«	◇	□	□
8		□	(	8	H	X	h	x	à	ò	®	Π	»	ÿ	□	□
9		□	)	9	I	Y	i	y	â	ô	©	π	…		□	□
A		□	*	:	J	Z	j	z	ä	ö	™	∫		□	□	□
B		□	+	;	K	[	k	{	ã	õ	´	ª	À	□	□	□
C		□	,	<	L	\	l	|	å	ú	¨	º	Ã	□	□	□
D		□	-	=	M	]	m	}	ç	ù	≠	Ω	Õ	□	□	□
E		□	.	>	N	^	n	~	é	û	Æ	æ	Œ	□	□	□
F		□	/	?	O	_	o		è	ü	Ø	ø	œ	□	□	□

Figure 10.12 Geneva font ASCII characters

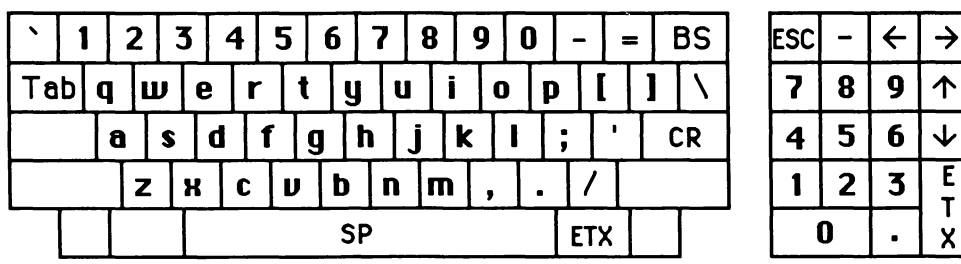


Figure 10.13 Unshifted keyboard layout

the mouse. The two *x* signals are called X-1 and X-2; the *y* signals are Y-1 and Y-2.

The position signals go to different places in the Mac. The X-2 and Y-2 signals go to the 6522, and the X-1 and Y-1 signals go to the Data Carrier Detect (DCD) inputs to the 8530 communications controller.

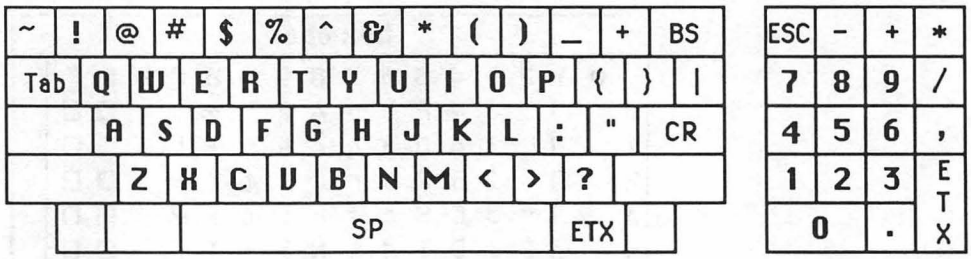


Figure 10.14 Keyboard layout with shift key



Figure 10.15 Keyboard layout with option key

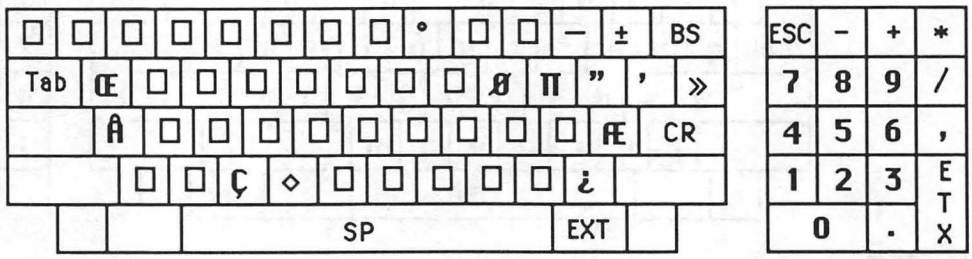


Figure 10.16 Keyboard layout with shift and option keys

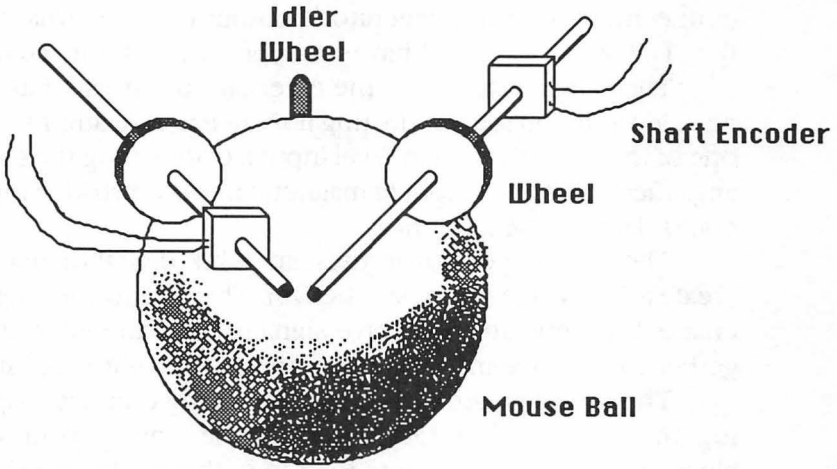


Figure 10.17 Mouse

The list below identifies the signals in the mouse connector at the back of the Macintosh.

Pin	*Signal*
1	Ground
2	+5v
3	Ground
4	X-2 (to 6522 PB4)
5	X-1 (to 8530 *DCDA)
6	N/C
7	Switch to ground (normally open)
8	Y-2 (6522 PB3)
9	Y-1 (8530 *DCDB)

The mouse push button just makes a connection to ground. It's up to the Mac to detect the switch-contact closure.

The Macintosh supplies +5 volts to the mouse to provide power for the optical shaft encoders. The +5 volts is not intended to supply current to a hungry device, just to provide a voltage level.

THE SOUND GENERATOR

The Macintosh has a small internal speaker that the sound generator normally uses. There is also a standard ⅛-inch phone jack on the back for

connecting the sound generator's output to an external speaker or amplifier. The speaker should have an impedance of 4 to 10 ohms.

The signal available at the external output jack has a 1-volt peak-to-peak level. If you are connecting it to an external amplifier, you should use one of the amplifier's high-level inputs. Connecting the sound output to an amplifier's low-level input (a magnetic phono cartridge input, for example) could damage the amplifier.

The sound generator uses an 8-bit digital-to-analog converter to create a four-voice sound synthesizer. That means the sound generator can create four separate sine-wave signals simultaneously and mix them together to produce more complex signals or to produce musical harmony.

The sound generator's digital-to-analog converter operates only during the horizontal retrace interval—the time period when the CRT's electron beam is returning to the top of the display. The beam also sweeps horizontally across the display and must return to the left side of the CRT after each sweep. Horizontal retrace takes a finite amount of time to move the beam back to the left side of the tube. The horizontal retrace interval is much shorter than the vertical retrace interval and occurs more often, every 44.93 microseconds (the horizontal sweep frequency is 22 KHz).

The consequence is that the digital-to-analog converter operates at a rate of 22 KHz, which enables it to produce tones of up to 11 KHz in frequency. All of its frequencies are submultiples of the horizontal sweep frequency. The lower frequency limit is imposed by the analog circuitry, not the digital-to-analog converter, and is below the range of human hearing.

THE PRINTER

The Imagewriter printer does amazing things with nine little wire pins that sweep across the page, making lots of little dots. How does it manage to produce such good-quality characters in different fonts and such incredible graphics? It's not magic, just a lot of carefully designed hardware and software.

The printer can print in three modes—draft, standard (proportional-space), and high-quality. Each has a different resolution. In draft mode, the printer operates like any normal dot matrix printer, putting about 120 characters on the page every second. The characters look like normal dot matrix characters, readable but not high-quality. In draft mode, the Imagewriter cannot do proportional spacing, so the Macintosh tries to put each word (on the paper) in the same place it would appear in standard or high-quality mode. The result looks a little strange, with a lot of space between some words.

The standard and high-quality modes print graphics. Even when printing text, the printer treats the characters in various fonts as if they were graphics (collections of tiny pictures). In the standard mode, the Imagewriter prints with a resolution of 80 dots per inch horizontally and 72 dots per inch vertically—that's close to the resolution of the Macintosh screen.

In high-quality mode, the Imagewriter prints 160 dots per inch horizontally and 144 dots per inch vertically. This resolution puts the dots closer together than the pins in the print head, so that the dots placed on the page overlap, producing a higher-quality image than you get with visibly separate dots.

You pay a price for standard and high-quality modes—that price is time. Compare the Imagewriter's draft-mode speed of 120 characters per second with its standard-mode speed of 90 characters per second. In high-quality mode, the Imagewriter prints at about half the standard-mode rate.

The Imagewriter file in your system folder is a resource file that contains a printer data record, the print driver program, and the print manager program. The print manager program converts a document into a QuickDraw picture file, reads the file (32 scan lines at a time), and passes the data to the printer via the print driver program. In draft mode, the data is just ASCII characters and goes directly to the print driver, bypassing the print manager.

The printer interfaces to the Macintosh through one of the two serial communications ports.

CHAPTER

11 THE 68000

Architecture
Addressing
Input/Output
Exceptions and Interrupts
Condition Codes
Instruction Set

ARCHITECTURE

No one could hope to teach assembly language programming in one or two chapters; entire books have been written about programming the 68000 processor in assembler. This chapter should, however, give the experienced assembly language programmer a feel for the capabilities of the 68000.

Anyone who has been writing really tight assembly code for the 6502 to do realtime animated graphics will appreciate the 68000. It's closer to a good-sized minicomputer in capabilities than it is to an 8-bit processor.

The 68000—the computing engine for the Macintosh—is a powerful and versatile microprocessor manufactured by Motorola. It is a true 16-bit microprocessor. It has 16-bit-wide internal and external data paths and 32-bit internal registers, and it can perform some 32-bit arithmetic operations. The address bus is 24 bits wide, allowing the 68000 to directly address up to 16 million bytes of memory.

All input and output devices connected to the 68000 are memory-mapped. To do input or output operations, you just read or write memory locations assigned to the device you are using. Having memory-mapped I/O means that some of the memory address space must be dedicated to I/O devices, but with 16 million addresses, there's plenty of space.

The 68000 has sixteen registers: eight address registers and eight data registers. The address registers may be used for both addressing and data operations. In addition, the processor has a 2-byte status register and a program counter that holds the address of the next instruction to be executed (figure 11.1).

The standard Motorola notation for registers is summarized below.

A*n*	Address register *n*
D*n*	Data register *n*
R*n*	Register *n* (0–7 = data register; 8–15 = address register)
PC	Program counter
SR	Status register
CCR	Condition code byte (user byte) of status register
SP	Stack pointer (A7)
USP	User stack pointer
SSP	Supervisor stack pointer

The 32-bit data registers can hold 32-bit, 16-bit, or 8-bit numbers. The 16- and 8-bit numbers align at the right in data registers (in the low-order bits). During 16- or 8-bit arithmetic operations, the processor

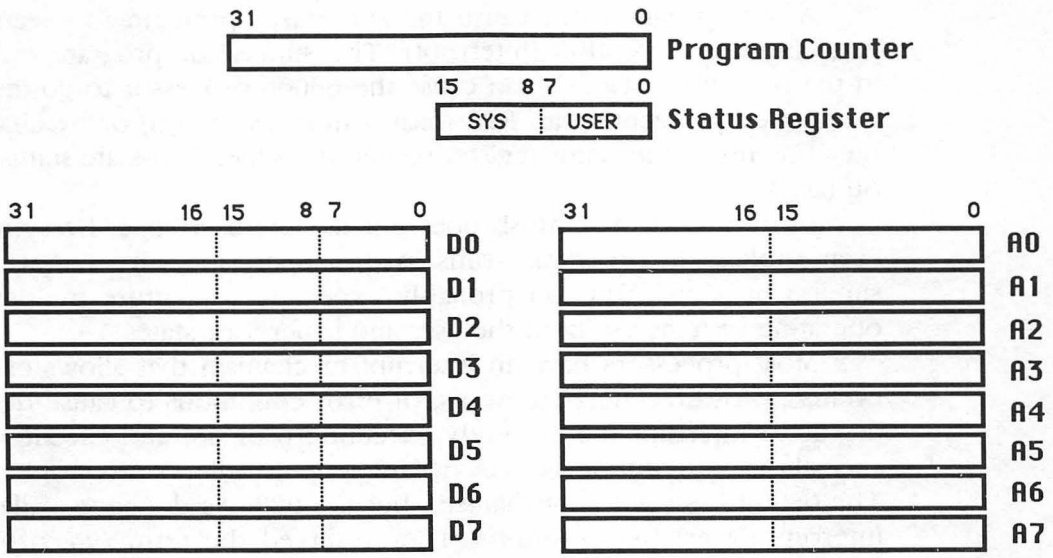


Figure 11.1 68000 registers

ignores the unused high-order bits in a data register. The address registers can be used only for 32-bit operations. In the 68000, bits are numbered right to left, with the least significant bit numbered 0 and the most significant bit numbered 31.

The 68000 operates in two states, the supervisor and user states. The Macintosh always runs the 68000 in the supervisor state. The high-order byte of the status register holds the status bits for the supervisor state, and the low-order byte holds the status bits for the user state. The two states of operation provide a mechanism for protecting operating-system or other supervisor code from errant user programs. Some 68000 operations are privileged operations that can be done only in the supervisor state. Most operating systems use the supervisor state for controlling input/output operations and for task control in multitasking systems. The supervisor-state status bit in the supervisor status byte determines whether the 68000 processor is in the supervisor or user state.

The 68000 maintains two separate stacks, a user stack and a supervisor stack. The stack pointer in both states is addressed as register A7. In the user state, A7 contains the user stack pointer; in the supervisor state, it has the supervisor stack pointer.

When operating in the user state, a program may not access the supervisor stack nor may it change the supervisor status bits. User programs are also prohibited from executing STOP or RESET instructions.

A user program can cause the supervisor program to execute by generating an exception (interrupt). The supervisor program executes in the supervisor state. It can cause the 68000 processor to go into the user state by executing an RTE (return from exception) or by changing the contents of the status register so that the supervisor-state status bit is off (zero).

Currently the Macintosh does not use the user state. Everything—even application programs—runs in the supervisor state and uses the supervisor stack. You can probably expect to see future multitasking operating systems use both the user and supervisor states.

Most processors have an interrupt mechanism that allows external events, software-generated events, or error conditions to cause the processor to interrupt the currently executing program and execute some special-purpose routine like a device interrupt handler or an error handler. The 68000 has such a mechanism, but the 68000's designers called the interrupting events *exceptions*. They reserved the term *interrupts* for exceptions caused by external devices.

Other than the instructions that can cause exceptions, the only instructions that affect the operation of the processor are the STOP and RESET instructions. RESET sends a reset signal to external devices, resetting them, one hopes. It has no other effect. The STOP instruction stops the processor from executing instructions. After "STOP," an exception caused by an interrupt or external reset will cause the processor to start executing instructions again.

The 68000 has a set of branching instructions that control program execution. Unconditional branch instructions cause a branch (jump) to another program location. Conditional branch instructions cause the program to branch to another location if a specified test of the condition code is true (the condition code is in the user portion of the status register). Among the set of branch instructions are subroutine call instructions and return-from-subroutine instructions.

Most microprocessors fetch instructions one byte at a time as needed by the instruction-decoding logic in the processor. The 68000 gives itself a leg up by fetching instructions (one word at a time) before they are needed by the instruction decoder—this process is called *instruction prefetch*. When the instruction decoder needs the next word of a multiple-word instruction, it doesn't have to stop and wait for a memory fetch cycle.

One feature of the 68000 is something you rarely see implemented in hardware, a trace capability. When tracing is turned on, the processor generates a trace exception after the execution of every instruction (except when executing an exception handler). The trace exception processor can

then keep a record of program execution. Setting the trace bit in the supervisor status byte to 1 enables tracing; setting it to 0 disables tracing.

The 68000 performs integer or decimal arithmetic calculations. A decimal number is stored as a binary-coded decimal (BCD). Integer arithmetic can be performed on 8-, 16-, or 32-bit operands. Operands may reside in memory or data registers; 16- or 32- bit operands may also reside in address registers.

ADDRESSING

The 68000 can address 8-, 16-, or 32-bit operands, but all memory references are byte addresses. In the vernacular of the 68000, 16-bit operands are called *words*, and 32-bit operands are called *long words*. Since the processor addresses bytes, byte addresses can have any value. Words are located on even byte boundaries only, so the first byte of a word always has an even address value, and the second byte of a word has an odd address value. Long words may be located on any word boundary (even address).

The high-order byte of a word is the first byte; it is on an even address boundary. The low-order byte is the second byte, lying on an odd address boundary. The high-order word of a long word is located on an even address, and the low-order word is located on the next even address. The most significant part of an operand always has the lower address value. See figure 11.2.

Instructions must always be stored on a word boundary. The operands of some instructions must also lie on word boundaries. Attempting to fetch an instruction from an odd address or addressing a word or long word at an odd address causes an address exception. A decimal number is composed of digits, each digit occupying 4 bits. The length of a decimal number is, therefore, a multiple of 2 digits (1 byte). The most significant digit is in the high-order bits of the first byte (lowest address). The least significant digit is in the last byte. Decimal instructions operate on a single byte but have provisions for doing multiple-precision (multiple-byte) arithmetic.

The simplest form of addressing in the 68000 is immediate data. It doesn't use an address for an operand at all; the operand is contained in the instruction. The immediate operand can be a byte, word, or long word. If it is a byte, it occupies an entire word, but only the lower-order byte is used. The limitation of this address mode is that the operand value is fixed; it must be a constant.

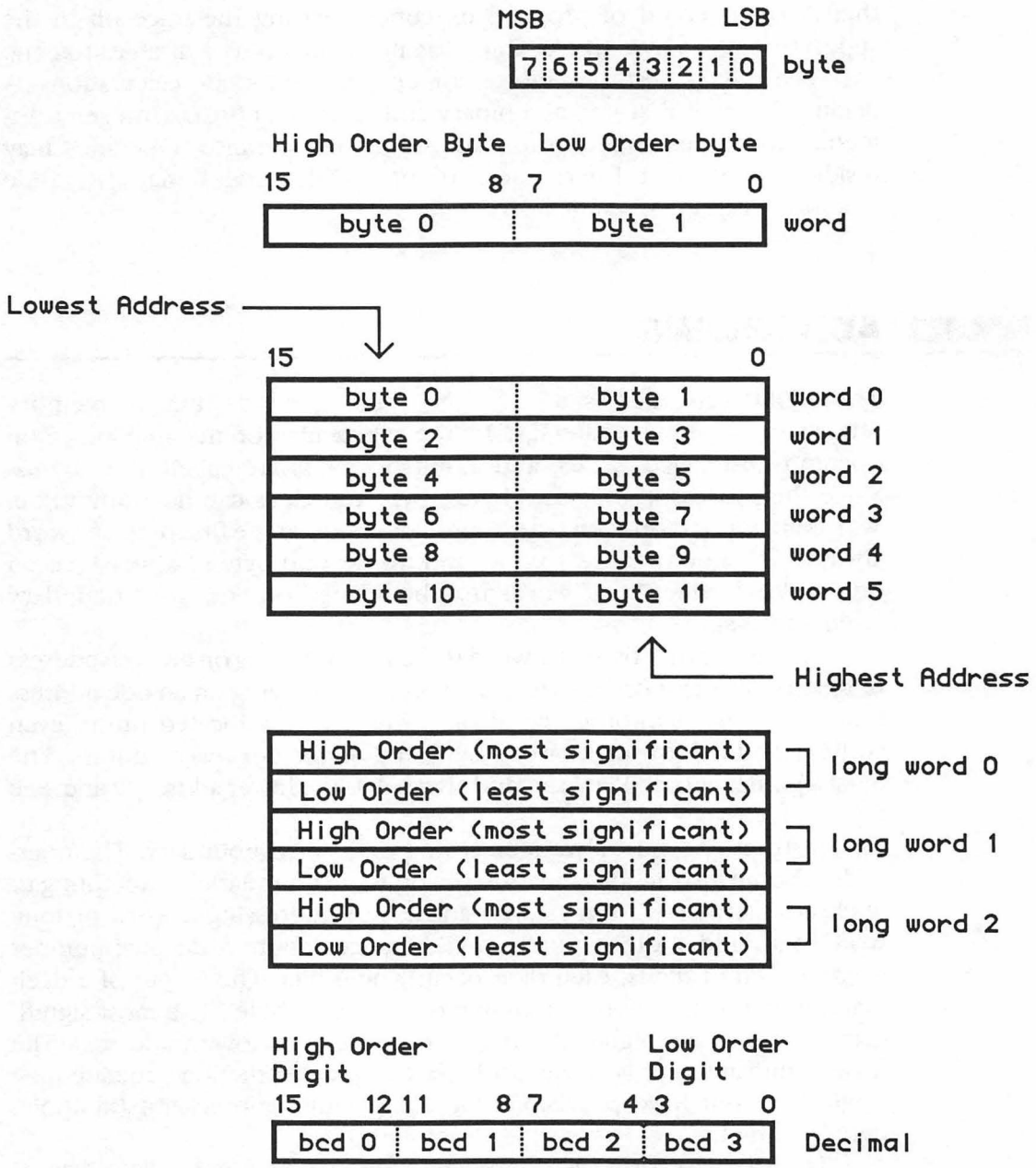


Figure 11.2 Address boundaries

The 68000 accesses operands in memory or in registers, and uses several different address modes to address operands. In register direct address mode, an instruction specifies the register containing the operand. Operands that reside in memory can be addressed with absolute memory addresses or with several register indirect or program counter address modes.

Each address mode uses a different method to calculate the address that the instruction will use to address memory. The result of that calculation is called the *effective address*.

An instruction using register indirect mode specifies a register that contains the memory address of the operand and supplies a displacement value as part of the instruction (figure 11.3). The effective address is the sum of the register contents and the displacement.

Register indirect with index uses a register containing a memory address, which is added to a displacement value contained in the instruction. The effective address equals the contents of the address register plus the contents of the index register plus the displacement. The index register may be either an address register or a data register. See figures 11.4 and 11.5.

There are two additional register indirect address modes. Register indirect with postincrement works just like register indirect, except the processor increments the value in the address register after getting the address from it. When using register indirect with predecrement, the processor decrements the value in the address register and then uses it as the effective address.

The 68000 also has program-counter relative addressing, both with and without an index register. See figures 11.6 and 11.7.

The predecrement and postincrement address modes are usually used to implement user stacks. Stacks grow down in memory from a high starting address to a low ending address. An address register is used as the

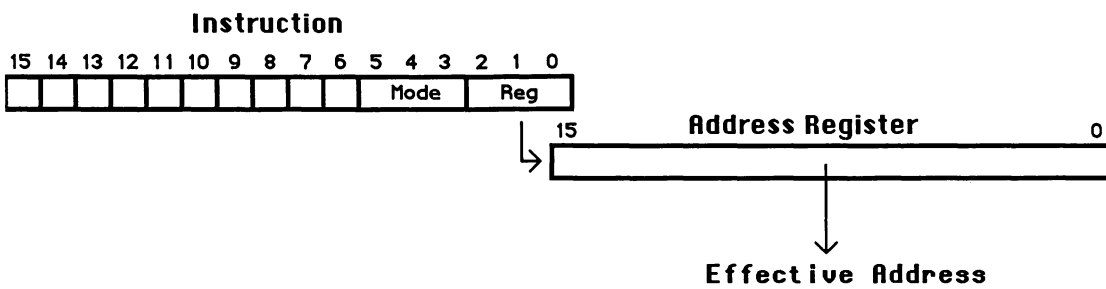


Figure 11.3 Register indirect addressing

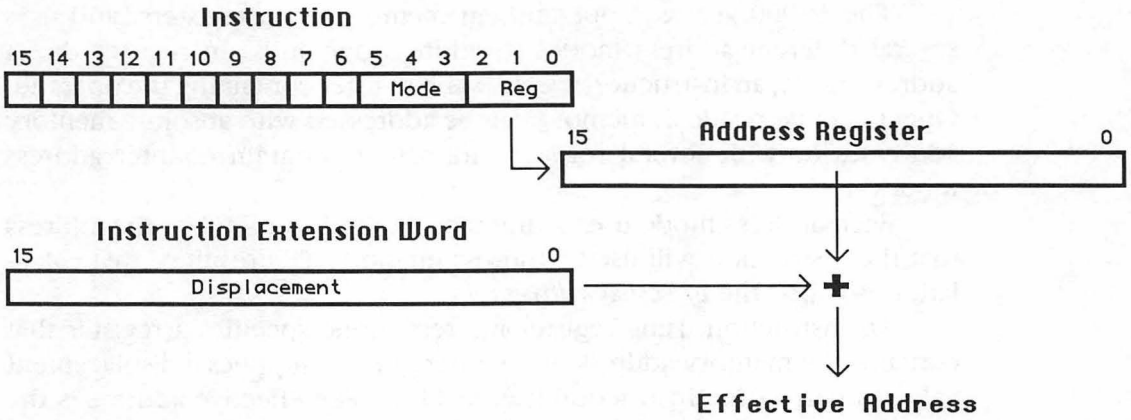


Figure 11.4 Register indirect with displacement

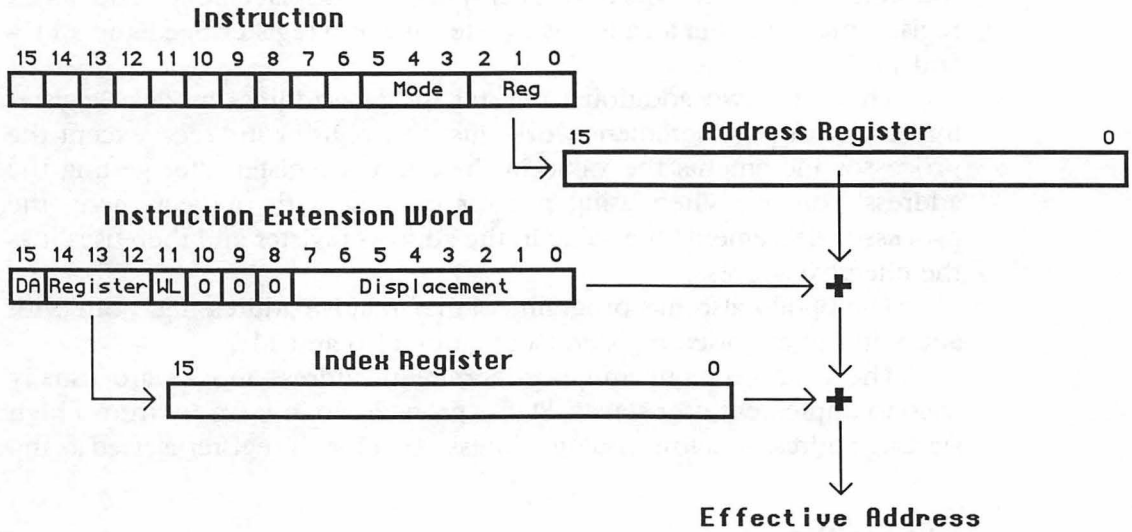


Figure 11.5 Register indirect with index

stack pointer and is initialized with the starting address of the stack. Only words or multiple words are put on the stack; the stack pointer always points to a word boundary.

To push a word onto the stack, use the register indirect with predecrement address mode and move the word to the effective address. The

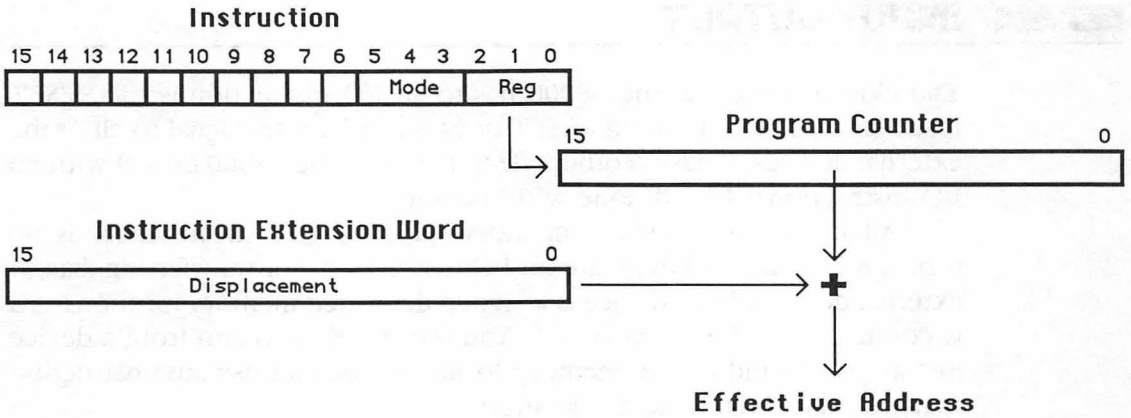


Figure 11.6 Program counter with displacement

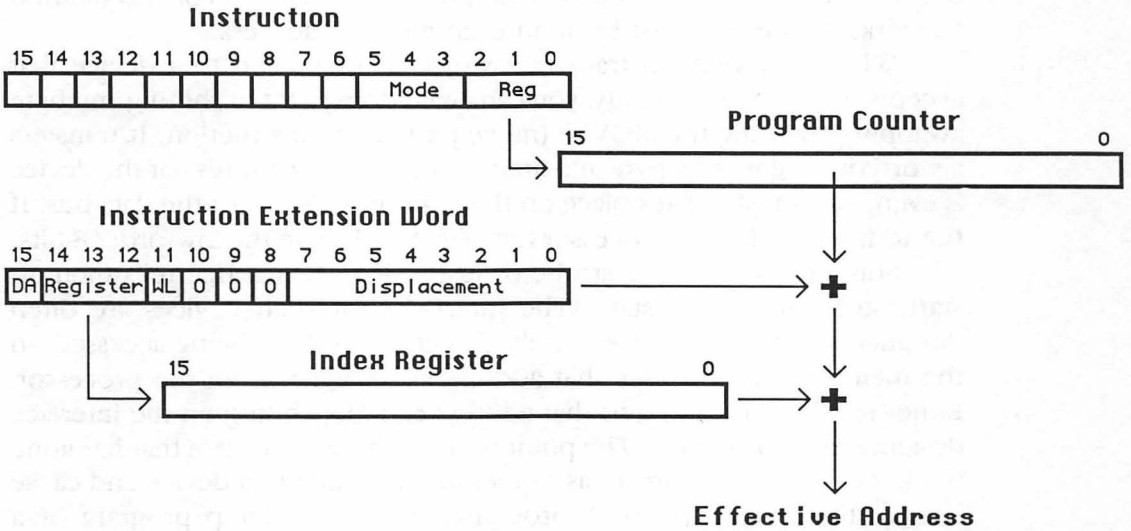


Figure 11.7 Program counter with index

processor will decrement the address register (stack pointer) and then move your data word to the address it specifies. To pop a word from the stack, do just the reverse; do a move from memory, using register indirect mode with postincrement. The processor will get a word from the location in the address register and then increment the register.

INPUT/OUTPUT

The closest thing that the 68000 has to an I/O instruction is the RESET instruction. The only thing RESET does is send a reset signal to all of the external devices; it has no other effect. How can the 68000 do I/O with no I/O instructions? It's all done with memory.

All I/O in the 68000 is memory-mapped. That means there is no separate I/O bus, and there are no I/O instructions for transferring data to external devices. Each device is assigned dedicated memory locations and is connected to the memory bus. You transfer data to and from a device just as you would to any memory location. You can use any instruction that would ordinarily access memory.

Most memory access instructions read and write memory one word at a time, but many devices are designed to transfer data a byte at a time. If you are dealing with a device that accepts data a byte at a time, you can use memory reference instructions that transfer a byte and have the device use the low-order 8 bits of data from the 16-bit data bus. For this method to work, the device must be at an even memory address.

When you want to transfer a word or long word to a device that accepts data by the byte only, you can avoid doing a lot of shifting and byte swapping by using the MOVEP (move peripheral) instruction. It transfers a word or long word a byte at a time. If the memory address of the device is even, the transfer takes place on the high-order 8 bits of the data bus. If the address is odd, the processor transfers the data on the low-order 8 bits.

Some devices need a strobe or just a single signal that tells them to start, stop, or change state. The interfaces for such devices are often designed so that the device merely detects its address being accessed on the memory bus and uses that address as its signal from the processor. Either reading or writing to that address can, depending on the interface design, signal the device. The point is, if you have a program that has gone wild, it can read or write areas of memory assigned to a device and cause strange things to happen. A programmer using a dump program or a debugger to read from memory addresses assigned to a device can have the same thing happen.

You now know how to transfer data to and from a device, but you still need to be able to find out what the device is doing. You can read a device status register just by reading its memory address, but you still need one more thing—interrupts.

EXCEPTIONS AND INTERRUPTS

Most processors allow external events to alter the execution of a program by means of interrupts. Some also allow interrupts to be generated by program statements or by error conditions that the processor detects. The 68000 has one of the most elaborate interrupt schemes that you will find on a microprocessor. Actually, it looks more like the interrupt structure of a mainframe or a superminicomputer.

The 68000's designers reserved the term *interrupt* for interrupts caused by external devices. They used the term *exception* for the general class of interrupting events. A device interrupt is just one type of exception. If you are accustomed to other microprocessors, just think "interrupt" whenever you see the word *exception*.

Exceptions in the 68000 are vectored. That is, each type of exception has a vector—the address of the program that will be executed when the exception occurs. When the exception occurs, the processor finds out what type of exception occurred, uses the vector number associated with that type of exception to calculate the address of the vector, reads the vector, and uses it to address the program that handles the exception.

The 68000 keeps the vector for each type of exception in a dedicated location in memory. If, for instance, the processor detects an illegal instruction, it knows that the vector number for an illegal-instruction exception is 4. The processor shifts the vector number left by 2 bits to generate the address of the vector (the result of the shift is hex 010) and then reads the vector from address hex 010 (figure 11.8). Table 11.1 lists the vector numbers and addresses for different types of exceptions.

For processor-generated exceptions, the processor identifies the kind of exception and the proper vector number to use. When the exception is a device interrupt, the device identifies the vector number.

A device has a choice of how to identify its vector number. The device can supply the vector number to the processor on the data bus during the interrupt-acknowledge cycle; the processor then uses that vector number to locate the exception vector. Or the device can request an autovector instead of supplying the vector number. An autovector is a vector number assigned to each interrupt priority level. All of the interrupts of the same priority share the same autovector number. The only advantage to using an autovector is that it requires less hardware in the device interface. The Macintosh uses autovectors for all interrupts.

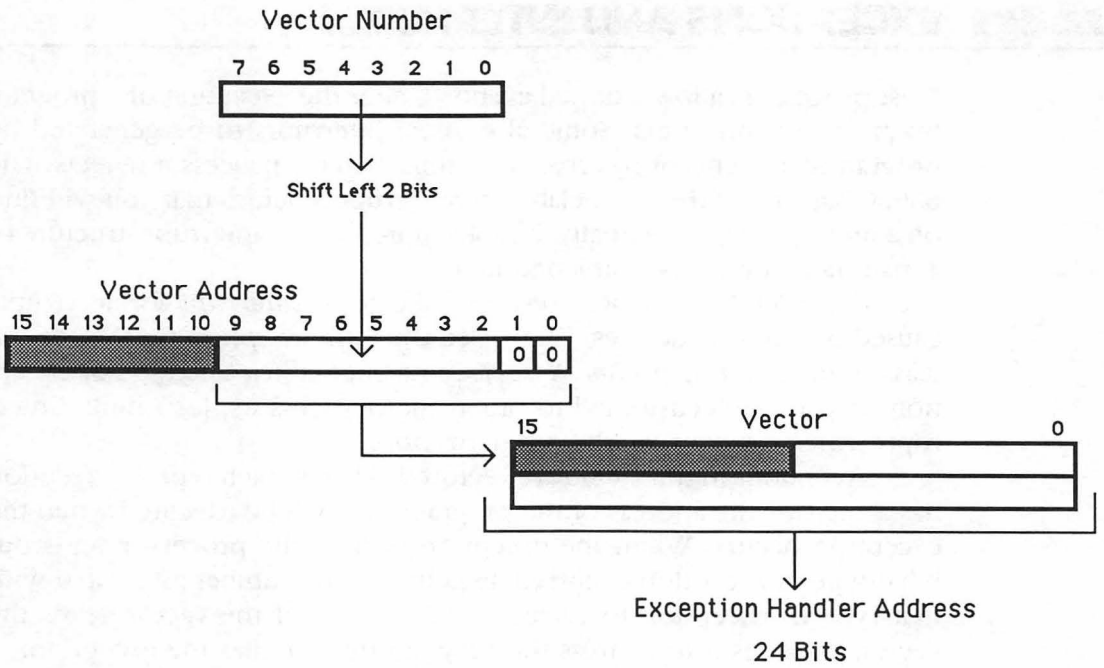


Figure 11.8 Finding the exception vector

In either case (the autovector or the device-supplied vector), the vector identification process is completely transparent to the program running the 68000. Communications with the device during the interrupt process are handled entirely by hardware.

The reset vector in the 68000 is 4 words long. It contains both the address of the first program instruction to execute and the initial stack pointer. All other vectors are 2 words and contain a 24-bit address.

Identifying and reading the vector are only a small part of processing an exception. The first step that the 68000 takes is to save the status register inside the 68000, set the supervisor-state status bit, and clear the trace status bit. The next step is to identify the exception, get the exception vector number, and shift the number left 2 bits to generate the vector address. In its third step, the processor pushes the old status-register contents and the program counter onto the supervisor stack. The last step is to read the vector, load its contents into the program counter, and resume instruction execution at the address specified by the vector.

If more than one exception occurs at the same time, the exceptions are handled in priority order. Priority numbering for exceptions is the opposite of priority numbering for interrupts. With interrupts, the highest

Table 11.1 Exception Vectors

Vector Number	*Hex Address*	*Vector*
0	000	Reset
2	008	Bus error
3	00C	Address error
4	010	Illegal instruction
5	014	Divide by zero
6	018	CHK instruction
7	01C	TRAPV instruction
8	020	Privilege violation
9	024	Trace
10–11	028–02C	Emulator (for 68000 in circuit emulator)
12–14	030–038	Reserved
15	03C	Uninitialized interrupt
16–23	040–05F	Reserved
24	060	Bus error during interrupt acknowledge
25	064	Priority 1 interrupt autovector
26	068	Priority 2 interrupt autovector
27	06C	Priority 3 interrupt autovector
28	070	Priority 4 interrupt autovector
29	074	Priority 5 interrupt autovector
30	078	Priority 6 interrupt autovector
31	07C	Priority 7 interrupt autovector
32–47	080–0BF	TRAP instruction vectors
48–63	0C0	Reserved
64–255	100–3FF	User interrupts

number is the highest priority. With exceptions, priority 0 is the highest, and 2 is the lowest priority.

Priority	*Exception*
0	Reset, bus error, address error
1	Trace, device interrupt, illegal instruction, privilege violation
2	TRAP, TRAPV, CHK, divide by zero

Device interrupts have a separate priority scheme, with seven priority levels. Each priority level may be used by multiple devices, which are

daisy-chained together with the interrupt acknowledge signal. The current Macintosh design does not use interrupt-acknowledge daisy chains.

There are two classes of priority for device interrupts: the device's interrupt priority level (1 to 7) and the exception priority (always 1 for device interrupts). You cannot alter the priority of the device interrupt exception, nor can you suspend one exception group in favor of another. The program can, however, determine which device-interrupt priority levels it will handle. The supervisor status byte contains 3 bits that determine the processor priority. Only interrupts with a priority level higher than the processor priority level will be accepted by the processor.

The way that interrupt and exception priorities are numbered can be confusing, so their priority schemes bear repeating. Exception priorities are numbered 0 to 2, with 0 the highest priority. Device interrupt priorities are numbered 1 to 7, with 7 the highest priority—the opposite order of the exception numbering scheme. It makes you wonder if people on the 68000 development project talked to each other.

CONDITION CODES

Condition codes are stored in the user-status-byte portion of the status register after certain types of instructions are executed. Supervisor-related status information is stored in the supervisor status byte (figure 11.9).

Let's look at the supervisor status byte first. The low-order 3 bits contain the interrupt mask. The "official" 68000 documentation from Motorola calls it a mask, but it is actually used to indicate the processor's

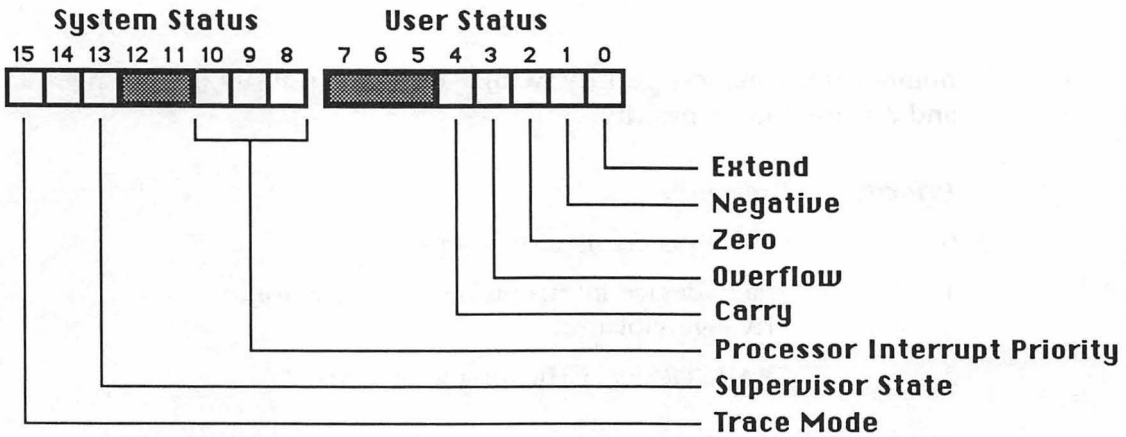


Figure 11.9 Status register

current interrupt priority level. Only interrupts with a priority level higher than the current processor priority level can interrupt the processor, with the exception of interrupts of priority 7 (these cannot be disabled).

When set, the supervisor-state bit indicates that the processor is in the supervisor state. Stack references use the supervisor stack, and the processor executes privileged instructions.

The trace bit's being set causes the processor to operate in trace mode. The processor generates a trace exception after executing each instruction. The trace exception handler then records the trace information in any way it deems appropriate.

All of the bits in the user status byte are part of the condition code. After the processor executes some types of instructions, the condition code bits are set to indicate the results of the instructions' execution. Individual instructions vary in which bits they set, but generally Extend, Overflow, and Carry are set by arithmetic instructions, and Negative and Zero are set by arithmetic, logical, and some other instruction types (for example, MOVE). (The tables at the end of this chapter list the condition code bits that are affected by each instruction.)

There is an entire set of instructions called *condition code test instructions* that test the status bits and allow the processor to alter its instruction execution based on the results of the test. The mnemonics for these instructions contain the letters *cc*. For example, the mnemonic for the conditional branch instruction is Bcc. When you are actually writing a program, you replace the letters *cc* with a one- or two-letter code that defines the condition you want the instruction to test (see table 11.2).

INSTRUCTION SET

In order to get a feel for the capabilities of the 68000 instruction set, we will look at a summary of the instruction set. The instructions are divided into seven classes (see the tables at the end of this chapter). Before looking at the instructions, though, you need to understand the instruction formats and address modes used in the summary.

Instructions are one to five words in length. An instruction cannot be an odd number of bytes in length, nor can it be stored at an odd byte address. The first word of the instruction contains the operation code and other information—such as the operand size, the address mode, and the register (if any)—used in calculating the effective address. The format and information contained in the instruction varies considerably from instruction to instruction, so let's look at several representative examples (figure 11.10).

Table 11.2 Condition Code Test

cc	*Condition*
T	Always evaluates to True
F	Always evaluates to False
HI	High; Z and C both = 0
LS	Low or same; Z or C = 1
CC	Carry clear; C = 0
CS	Carry set; C = 1
NE	Not equal; Z = 0
EQ	Equal; Z = 1
VC	Overflow clear; V = 0
VS	Overflow set; V = 1
PL	Positive; N = 0
MI	Negative; N = 1
GE	Greater than or equal; N and V are the same
LT	Less than; N and V are different
GT	Greater than; N and V are the same, and Z = 0
LE	Less than or equal; N and V are different, and Z = 1

In the STOP instruction, the operation code occupies the entire first word. The other extreme is the MOVE instruction, in which the operation code takes up only bits 15 and 14. Note that other information is coded in the middle of some instructions. For instance, the OR instruction has register select and operation mode fields in the middle.

Some instructions, like CMPI and Bcc, occupy more than one word in memory, but the operation code never extends beyond the first word. The additional word or words contain immediate data (as in the case of CMPI) or additional addressing information. The Bcc instruction has the displacement portion of an address in the additional word.

Many instructions use a 6-bit field to specify an address mode and an address register; the entire 6-bit field is called the *effective address*. In table 11.3, you see the address modes—each with the 3-bit address mode field, the 3-bit register field, and the address mode code that we use in the instruction summary. Note that the last five address modes have the same value in the 3-bit mode field of the instruction. In those address modes, no register is used in calculating the address, so the instruction format uses the contents of the address register field to distinguish among the last five address modes.

Some instructions can use operands of several different sizes, so they have a size field to indicate the operand size. The instruction summary

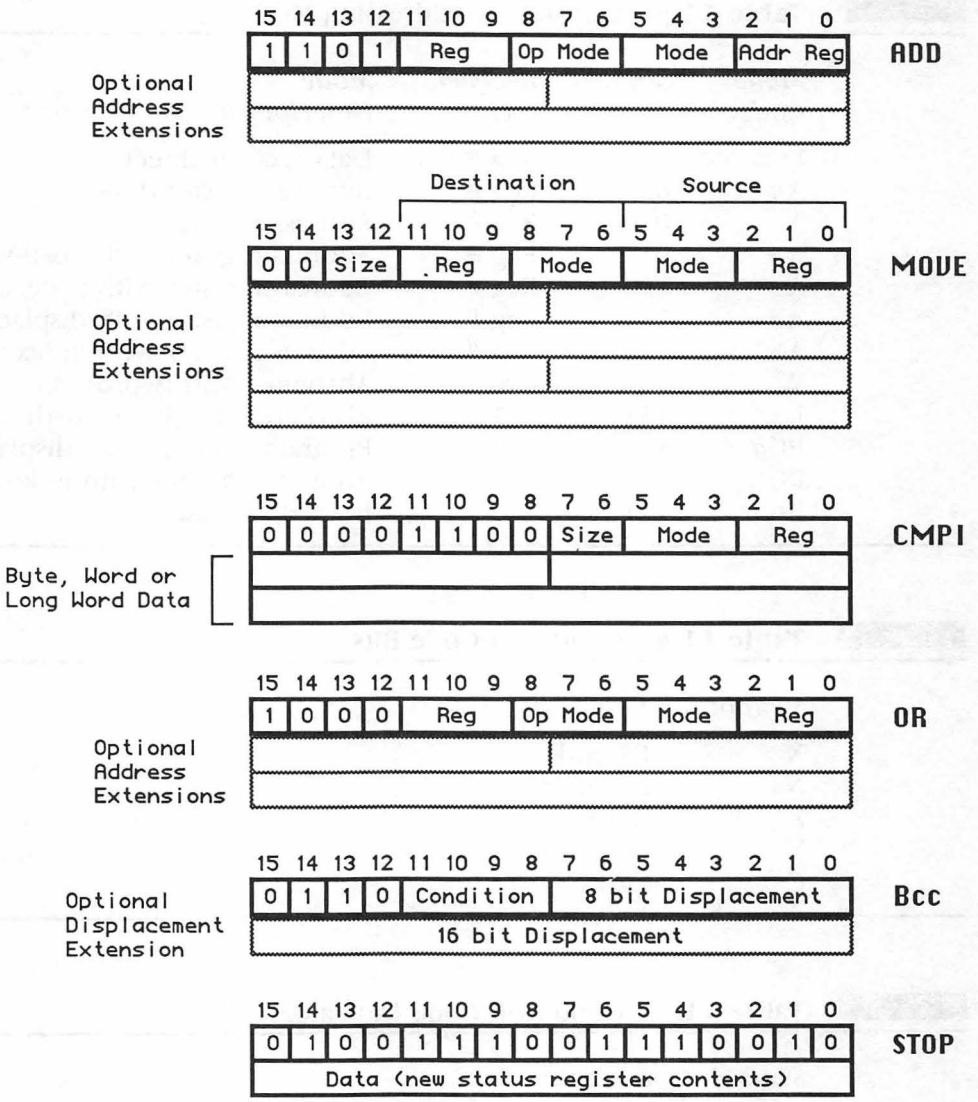


Figure 11.10 Instruction format examples

tables have a size column that shows which of the three operand sizes (8, 16, and 32 bits) an instruction uses.

The condition code column in the summary tables indicates which condition code bits an instruction affects. Table 11.4, Condition Code Bits, lists the names of the condition code bits identified at the head of the column. Table 11.5, Condition Code Indicators, defines the symbols found in the condition code column.

Table 11.3 Instruction Addressing Modes

Mode Code	*Mode Field*	*Register Field*	*Mode Description*
D*n*	000	Reg #	Data register direct
A*n*	001	Reg #	Address register direct
A	010	Reg #	Address register
A+	011	Reg #	Address register with postincrement
A−	100	Reg #	Address register with predecrement
A*d*	101	Reg #	Address register with displacement
AX	110	Reg #	Address register with index
W	111	000	Absolute short (word)
L	111	001	Absolute long (long word)
PC*d*	111	010	Program counter with displacement
PCX	111	011	Program counter with index
Imm	111	100	Immediate data

Table 11.4 Condition Code Bits

Symbol	*Bit*
X	Extend
N	Negative
Z	Zero
V	Overflow
C	Carry

Table 11.5 Condition Code Indicators

Symbol	*Meaning*
S	Bit is set (to either 0 or 1) by instruction
?	Bit value unpredictable after instruction execution
0	Bit always set to 0 by instruction
1	Bit always set to 1 by instruction
.	Bit not affected by instruction

Tables 11.6 through 11.12 summarize the instruction set, grouping the instructions in seven classes: Data Movement, Arithmetic, Shift/Rotate, Logical, Bit Manipulation, Branching, and Processor Control.

Table 11.6 Data Movement Instructions

Instruction	Condition Code Byte X N Z V C	Size(s) 8 16 32	Address Mode Dn	An	A	A+	A–	Ad	AX	W	L	PCd	PCX	Imm	Description
EXG	. . . . .	. . X	.	.	.	.	.	.	.	.	.	.	.	.	Exchange registers
LEA	. . . . .	. . X	.	.	X	.	.	X	X	X	X	X	X	.	Load effective address
LINK	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Link and allocate
MOVE	. S S O O	X X X	X	.	X	X	X	X	X	X	X	.	.	.	Move data
MOVEA	. . . . .	. X X	X	X	X	X	X	X	X	X	X	X	X	X	Move address
MOVEM	. . . . .	. X X	.	.	X	X	X	X	X	X	X	X	X	.	Move multiple registers
MOVEP	. . . . .	. X X	.	.	.	.	.	.	.	.	.	.	.	.	Move to/from peripheral
MOVEQ	. S S O O	. . X	.	.	.	.	.	.	.	.	.	.	.	.	Move quick
PEA	. . . . .	. . X	.	.	X	.	.	X	X	X	X	X	X	.	Push effective address
SWAP	. S S O O	. . X	.	.	.	.	.	.	.	.	.	.	.	.	Swap register words
UNLK	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Unlink

Table 11.7 Arithmetic Instructions

	Condition Code Byte					Size(s)			Address Mode											
Instruction	*X*	*N*	*Z*	*V*	*C*	*8*	*16*	*32*	*Dn*	*An*	*A*	*A+*	*A–*	*Ad*	*AX*	*W*	*L*	*PCd*	*PCX*	*Imm*	*Description*
ABCD	S	?	S	?	S	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Add decimal with extend
ADD	S	S	S	S	S	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Add binary
ADDA	.	.	.	.	.	.	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Add address
ADDI	S	S	S	S	S	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Add immediate
ADDQ	S	S	S	S	S	X	X	X	X	X	X	X	X	X	X	X	X	.	.	.	Add quick
ADDX	S	S	S	S	S	X	X	X	.	.	.	.	.	.	.	.	.	.	.	.	Add extended
CLR	.	0	1	0	0	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Clear operand (set to zero)
CMP	.	S	S	S	S	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Compare
CMPA	.	S	S	S	S	.	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Compare address
CMPI	.	S	S	S	S	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Compare immediate
CMPM	.	S	S	S	S	X	X	X	.	.	.	.	.	.	.	.	.	.	.	.	Compare memory
DIVS	.	S	S	S	O	.	X	.	X	.	X	X	X	X	X	X	X	X	X	X	Signed divide
DIVU	.	S	S	S	O	.	X	.	X	.	X	X	X	X	X	X	X	X	X	X	Unsigned divide
EXT	.	S	S	O	O	.	X	X	.	.	.	.	.	.	.	.	.	.	.	.	Sign extend
MULS	.	S	S	O	O	.	X	.	X	.	X	X	X	X	X	X	X	X	X	X	Signed multiply
MULU	.	S	S	O	O	.	X	.	X	.	X	X	X	X	X	X	X	X	X	X	Usigned multiply
NBCD	S	?	S	?	S	X	.	.	X	.	X	X	X	X	X	X	X	.	.	.	Negate decimal with extend
NEG	S	S	S	S	S	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Negate
NEGX	S	S	S	S	S	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Negate with extend
SBCD	S	?	S	?	S	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Subtract decimal with extend
SUB	S	S	S	S	S	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Subtract
SUBA	.	.	.	.	.	.	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Subtract address
SUBI	S	S	S	S	S	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Subtract immediate
SUBQ	S	S	S	S	S	X	X	X	X	X	X	X	X	X	X	X	X	.	.	.	Subtract quick
SUBX	S	S	S	S	S	X	X	X	.	.	.	.	.	.	.	.	.	.	.	.	Subtract with extend
TAS	.	S	S	O	O	X	.	.	X	.	X	X	X	X	X	X	X	.	.	.	Test and set
TST	.	S	S	O	O	X	X	X	X	.	X	X	X	X	X	X	X	.	.	.	Test

Table 11.8 Shift/Rotate Instructions

Instruction	Condition Code Byte X N Z V C	Size(s) 8 16 32	Address Mode Dn	An	A	A+	A–	Ad	AX	W	L	PCd	PCX	Imm	Description
ASL	S S S S S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Arithmetic shift left
ASR	S S S S S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Arithmetic shift right
LSL	S S S . S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Logical shift left
LSR	S S S . S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Logical shift right
ROL	. S S O S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Rotate left
ROR	. S S O S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Rotate right
ROXL	S S S O S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Rotate left with extend
ROXR	S S S O S	X X X	.	.	X	X	X	X	X	X	X	.	.	.	Rotate right with extend

Table 11.9 Logical Instructions

Instruction	Condition Code Byte X N Z V C	Size(s) 8 16 32	Address Mode Dn	An	A	A+	A–	Ad	AX	W	L	PCd	PCX	Imm	Description
AND	. S S O O	X X X	X	.	X	X	X	X	X	X	X	X	X	X	Logical AND
ANDI	. S S O O	X X X	X	.	X	X	X	X	X	X	X	.	.	.	Logical AND immediate
EOR	. S S O O	X X X	X	.	X	X	X	X	X	X	X	.	.	.	Logical exclusive OR
EORI	. S S O O	X X X	X	.	X	X	X	X	X	X	X	.	.	.	Exclusive OR immediate
NOT	. S S O O	X X X	X	.	X	X	X	X	X	X	X	.	.	.	Logical complement
OR	. S S O O	X X X	X	.	X	X	X	X	X	X	X	X	X	X	Logical inclusive OR
ORI	. S S O O	X X X	X	.	X	X	X	X	X	X	X	.	.	.	Inclusive OR immediate
Scc	. . . . .	X . .	X	.	X	X	X	X	X	X	X	.	.	.	Set byte conditionally

Table 11.10 Bit Manipulation Instructions

Instruction	Condition Code Byte X N Z V C	Size(s) 8 16 32	Address Mode Dn	An	A	A+	A–	Ad	AX	W	L	PCd	PCX	Imm	Description
BCHG	. . S . .	X . X	X	.	X	X	X	X	X	X	X	.	.	.	Test and change bit
BCLR	. . S . .	X . X	X	.	X	X	X	X	X	X	X	.	.	.	Test and clear bit
BSET	. . S . .	X . X	X	.	X	X	X	X	X	X	X	.	.	.	Test and set bit
BTST	. . S . .	X . X	X	.	X	X	X	X	X	X	X	X	X	.	Test bit

Table 11.11 Branching Instructions

Instruction	Condition Code Byte X N Z V C	Size(s) 8 16 32	Address Mode Dn	An	A	A+	A–	Ad	AX	W	L	PCd	PCX	Imm	Description
Bcc	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Branch on condition
BRA	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Branch unconditionally
BSR	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Branch to subroutine
DBcc	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Branch on condition/decrement
JMP	. . . . .	. . .	.	.	X	.	.	X	X	X	X	X	X	.	Jump (branch relative)
JSR	. . . . .	. . .	.	.	X	.	.	X	X	X	X	X	X	.	Jump to subroutine
RTR	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Return and restore condition codes
RTS	. . . . .	. . .	.	.	.	.	.	.	.	.	.	.	.	.	Return from subroutine

Table 11.12 Processor Control Instructions

Instruction	*Condition Code Byte* X	N	Z	V	C	*Size(s)* 8	16	32	*Address Mode* Dn	An	A	A+	A–	Ad	AX	W	L	PCd	PCX	Imm	*Description*
ANDI to CCR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Trap on overflow
ANDI to SR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	AND to status register
CHK	.	S	?	?	?	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Range check register
EORI to CCR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	EOR to condition code register
EORI to SR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	EOR to status register
ILLEGAL	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Cause illegal instruction exception
MOVE to CCR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Move to condition code register
MOVE to SR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Move to status register
MOVE from SR	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Move from status register
MOVE USP	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Move user stack pointer
NOP	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	No operation
ORI to CCR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	OR immediate to condition code register
ORI to SR	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	OR immediate to status register
RESET	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Reset external devices
RTE	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Return from exception
STOP	S	S	S	S	S	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Stop processor
TRAP	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Trap (generate trap exception)
TRAPV	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	Trap on overflow

CHAPTER

12 ASSEMBLER

The Assemblers

Developing Applications

MacASM

The Macintosh 68000 Development System

Exec

Edit

Asm

Link

PackSyms—the Symbol Packer

RMaker—the Resource Compiler

The Debuggers

THE ASSEMBLERS

Until now, we have been dealing with interpretive languages that handle a lot of the details of integrating an application into the Macintosh environment. Most of the interpretive languages themselves have been well integrated into the Mac and have been as easy to use as any other application. We are about to leave some of that behind.

Assemblers are usually designed by the kind of people who use them, very technically oriented programmers who are so accustomed to putting up with poor user interfaces that they consider them normal. The assemblers for the Macintosh are not that bad, but they do require a more technical orientation and a lot more attention to detail than the BASIC and Pascal interpreters.

Programming in assembly language on any machine is not something that you should attempt unless you are a serious programmer. That doesn't mean you need twenty years of programming experience (I know fifteen-year-olds who are quite competent in assembler). You should, however, be prepared for a very technical and demanding task. On the Macintosh, assembler programming is even more involved. Not only must you be a good assembly language programmer, but you must also be familiar with the Macintosh operating system and toolbox. This book and the other three in this series will help, but you should have a copy of the *Inside Macintosh* documentation from Apple Computer and the documentation that comes with your assembler.

Many assembler programmers who are working on the Macintosh have the Macintosh 68000 Development System (MacDS) from Apple Computer and MacASM from Mainstay. They both have their strong points.

MacASM is good for creating small- to medium-sized programs. The assembler, editor, source code, and object code are all in memory. The editor and assembler work from memory and hence are very fast. The MacASM tools lack the range of capabilities of the Apple assembler, but they can still produce complete, stand-alone applications and do it quickly.

MacASM's only shortcomings are its lack of capabilities equal to the Apple assembler (relocatable object modules, a linker, and full-feature resource compiler) and its less user-friendly editor and user interface. MacASM was designed to be compact and fast, so it's hard to fault it for having fewer bells and whistles. And if you are an experienced assembly language programmer, you will probably feel right at home with its user interface.

The Macintosh Development System from Apple Computer is more than an assembler; it's a full-feature development system for developing

complete, stand-alone Macintosh applications. Even though it isn't as easy to use as some Macintosh applications, it's that way partially because of features that assembly language programmers demand.

MacDS has a number of applications that are linked together in a way not common to other Mac applications. You can transfer from one to another without having to go back to the Finder and wait for it to rebuild the desk top. That capability—coupled with an ability to execute command files that run the entire assembly/link process quickly and automatically—makes MacDS fast and relatively easy to use. Unlike MacASM, it is a disk-based system (source and object code, for example, are in disk files); thus it doesn't suffer from the limitations in program size that memory-based systems do.

Most assembly language programmers will find both MacDS and MacASM useful—MacASM for quickly trying out ideas, and MacDS for developing full-blown applications. With these assemblers you can build applications on a stand-alone Macintosh; you don't need the Lisa or another Macintosh computer. Both assemblers can be used from a floppy disk, but for serious development work, you will probably want a Winchester disk drive and a 512K Macintosh.

DEVELOPING APPLICATIONS

When we discussed interpretive languages, we were able to ignore some of the details of how an application fits into the Macintosh operating system; those specifics were handled for us by the interpreter. Now that we are going to develop our own stand-alone applications, we need to know more about such things as segmentation, resources, files, building an application, and some assembler coding conventions.

A Macintosh file carries two identification codes that are important to the Finder, the file signature and the file type. The signature identifies the program that should be started if the user opens the file. Application programs have the file type of APPL and a file signature unique to the application.

A file created by MacWrite has a signature that identifies MacWrite as the program to run when the file is opened. Its file type might be MacWrite or text-only. A MacPaint file has a file signature that identifies MacPaint as the program that created it and a file type that identifies it as a MacPaint image file. The MacPaint program itself has the MacPaint file signature and a file type of APPL, indicating that it is an application program. The file type indicates the format of the data in the file, and the file signature indicates the program to start if the file is opened.

The file signature and file type are each a four-letter code. New file type and signature codes must be assigned by Apple in order to avoid conflict among programs for the Macintosh.

A Macintosh file has two parts, called *forks*—a resource fork and a data fork. The data fork can contain any type of data; the resource fork contains resources. The executable code of a program is a resource, as are icons, fonts, and templates for windows, dialog boxes, menus, and alert boxes.

A file doesn't necessarily have to use both the resource fork and the data fork. One of them may be empty. A data file may have an empty resource fork. Or an application program may not contain any data.

All Macintosh executable programs are broken into segments. The maximum size of a segment is 32K bytes. The program is actually loaded into memory by the segment loader, which dynamically allocates segments in the heap and loads the program segments into them from the disk. You must have one main segment (sometimes called the *base*, or *root*, segment) that stays in RAM as long as the program executes. The other segments may be called in from a disk as they are needed.

The program doesn't have to worry about getting segments into memory as they are needed; the segment manager takes care of that. Any time a program makes a memory access outside of the current segment, the segment manager handles that access and loads the segment if necessary.

Programmers can specify where programs are to be broken into segments, or they may let the linker break them into segments. If a program is short enough, it may remain in one segment.

Properly used, the segment loader can make it possible to run a program that would not otherwise fit in memory. When a program is executing, it can cause the segment loader to unload any segment except the main segment, making room for additional program segments.

You can keep local variables in the segment where they are used, but because segments can come and go as a program executes, putting widely used variables (globals) in a segment doesn't pay. These variables go in a separate area of memory that is not part of the heap and stay in memory as long as the program executes.

Sometimes the term *resource* is used in a confusing manner; you hear about resource forks and resource files. Every file has a resource fork, and an application program file is no exception. An application may, in fact, consist of resources only (its data fork could be empty). So what's a resource file? It's a separate file from which an application program can retrieve resources—usually templates for windows, menus, and the like. The separate resource file can also contain code segments.

Usually when you are developing a program, it's more convenient to keep the code in the application file, and most of the other resources in a separate resource file. Once development is completed, you can integrate all of the resources in the application file. If you expect to later convert the program to use foreign languages in the menus, you might want to keep a lot of the resources in a separate file.

The linker creates application program files; everything that is in an application—code and other types of resources—is put there by the linker. Separate resource files can be created by either the linker or the resource compiler (called RMaker in MacDS).

The Macintosh system software expects subroutines to preserve the contents of the A1 through A6 and the D3 through D7 registers. Any subroutine that may be called by the system must save and restore these registers. On the other hand, you can expect any toolbox or system routines that you call to return those registers intact. For consistency in your own programs, a good design practice is to follow the same register-saving conventions in subroutines.

A5 is a special case. The system expects A5 to point to the application global area. If you alter A5, you must restore it *before* calling any system routine or executing a trap. The best thing to do is leave A5 alone.

MACASM

MacASM is an assembler that resides in memory. It can work from files when necessary, but its real value is its speed when working from memory. It's a truly great assembler for prototyping and experimenting.

MacASM uses a command-driven editor; it does not use the Macintosh Edit menu or the mouse. If you are really sold on editing with the mouse, you can use MacWrite to edit your MacASM source files, but you pay a price; you lose the speed that you gained by having the editor, assembler, source code, and object code all in memory at the same time.

MacASM directly produces executable-code modules. Unlike MacDS, however, it does not have a separate linker to segment your program. Thus, with MacASM, you specify segmentation at assembly time, using macro instructions.

While MacASM has macro-instruction capabilities, the format for macro definitions is not compatible with the format for MacDS or Lisa macro definitions (both supported by MacDS). The differences are not that great, though, so conversion to MacDS macros, when you have finished experimenting in MacASM, should be easy.

The comment designations in MacASM and MacDS are also different. In MacDS, a comment at the end of a program line begins with a semicolon. In MacASM, the first nonblank character after the statement's operand fields starts a comment, which begins with a semicolon or an asterisk. Since the semicolon works in both assemblers, if you always start a comment with a semicolon and separate the comment by at least one blank from the operands, both assemblers will like it.

The debugger included with MacASM is similar to the MacsBug debugger that comes with MacDS.

MacASM uses the standard 68000 assembler notation for instruction mnemonics, operands, expressions, and address modes. For assembler syntax and notation, see the section in this chapter entitled Asm. One exception to the notation in that section is an extra address notation for PC relative addressing with index. The MacDS assembler assembles the notation *expr(Dn)* as if it were *expr(PC,Dn)*. MacASM does not support that address notation.

MacASM includes a program that performs some of the functions of a resource compiler. The resource definitions are in a designated area of your program's source file. You assemble the program but produce an object file instead of putting the object code in memory. If you double-click the object file's icon on the desk top, the resource compiler runs and creates a stand-alone application file.

THE MACINTOSH 68000 DEVELOPMENT SYSTEM

The Macintosh 68000 Development System is more than an assembler; it's a complete system for creating Macintosh applications in assembler language. It contains two different types of debugger and six separate programs for producing an application (including an executive).

Let's take a look at the programs that are part of the Macintosh 68000 Development System (figure 12.1).

Program	*Function*
Exec	An executive that executes command files to do the complete compilation, link, and resource compile steps. It executes alternate programs (usually the editor) if a program encounters errors.

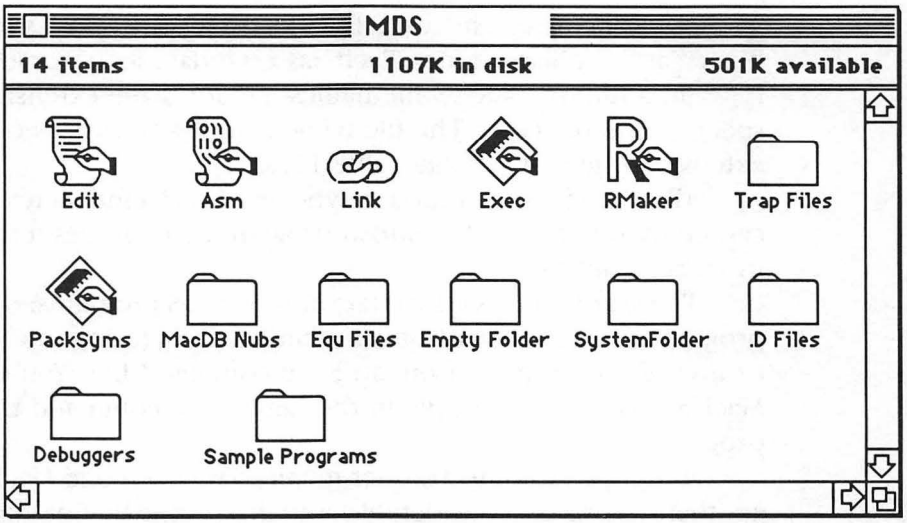


Figure 12.1 MacDS directory

Program	*Function*
Edit	A text-only editor for creating assembler source files, Exec control files, linker control files, and resource compiler source files. It uses the Edit menu and the mouse.
Asm	A 68000 macro assembler that uses standard Motorola mnemonics and syntax.
Link	Links relocatable object files produced by the assembler to produce applications or resource files.
PackSyms	Reads a symbol file produced by the assembler and converts it to a packed format that can be reused by the assembler to speed the assembly process.
RMaker	A resource compiler that uses a resource definition file to control the generation of a resource file.
Debuggers	A collection of debuggers that can run an application, set breakpoints, and examine and change memory or registers.

MacDS creates and uses files in three different formats: text-only, binary, and application files. Each has a standard icon that identifies the file type. In addition, MacDS file names include a file extension that further specifies the file type. The file name is followed by a period and the file extension (figure 12.2). See table 12.1.

All of the files identified as type *text* are Macintosh text-only files and can be used with any Macintosh program that handles text-only files (for instance, MacWrite).

There are three ways to start most MacDS programs: double-click the program's icon, transfer from another program (using the Transfer menu), or invoke the program from an Exec command file. You can specify the MacDS program's input file in the same Exec command that invokes the program.

When you use the Transfer menu to go from one MacDS program to another, you save a considerable amount of time by not going back to the Finder. The Finder must rebuild the desktop display every time you exit a Macintosh program, and that takes time. The use of the Transfer menu makes the MacDS system work like a separate mini–operating system. You can do endless edit-assembly-link-edit cycles and never go back to the Finder (figure 12.3).

Note that you cannot start most MacDS programs by double-clicking the file that you will use as input for the program. Most MacDS program input files contain commands that control the execution of the program, and you create those files with the editor. If you double-click the linker

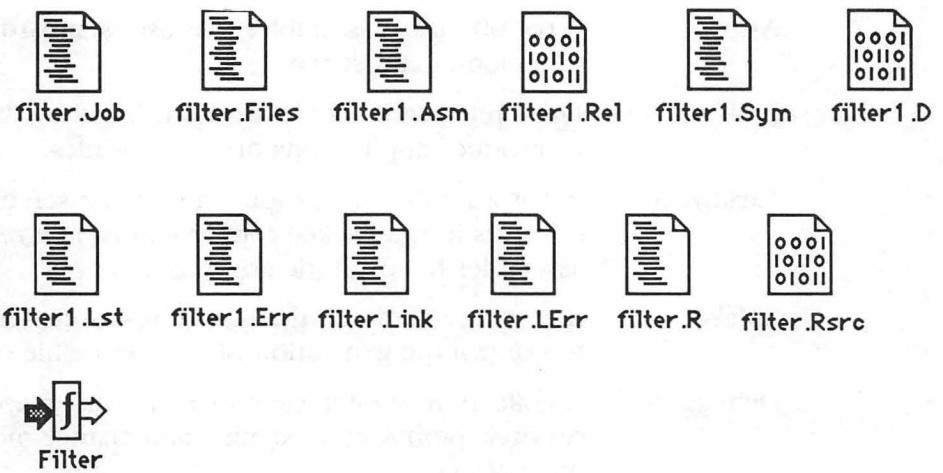


Figure 12.2 MacDS file icons

Table 12.1 MacDS File Types

Extension	*Type*	*Creator*	*Use*
.Link	Text	Edit	Linker control file
.Asm	Text	Edit	Assembler program source
.R	Text	Edit	RMaker control file
.Job	Text	Edit	Exec control file
.Files	Text	Edit	List of assembler source files
.Rel	Binary	Asm	Relocatable object
.Lst	Text	Asm	Assembler listing
.Err	Text	Asm	Assembler errors
.Sym	Text	Asm	Symbol table
.Map	Text	Link	Link map
.LErr	Text	Link	Linker errors
.D	Binary	PackSyms	Packed symbols
.Rsrc	Binary	RMaker	Resource file



Figure 12.3 A Transfer menu

control file, filter.Link, you invoke the program that created it, the editor, not the linker.

If you start the program by double-clicking its icon or transferring from another program, the program either presents you with a Minifinder dialog box or has an item in the File menu that invokes the dialog box. You specify the input file by choosing it from the dialog box. Most MacDS programs limit the choices in the dialog box to files that have the file name extension that they expect. For instance, the Exec program only displays files with the Job extension (figure 12.4).

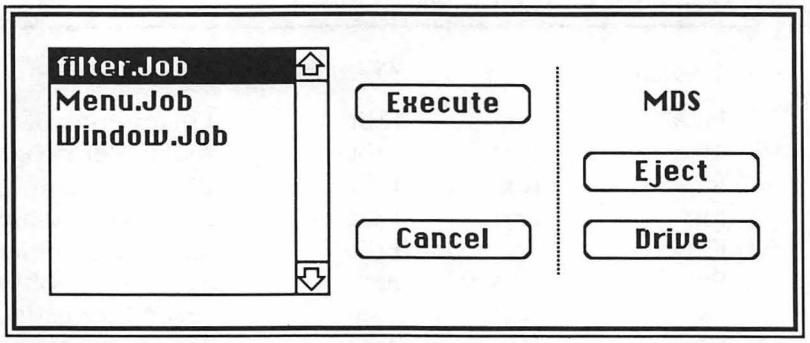


Figure 12.4 Exec Minifinder dialog box

A lot of MacDS programs have an Edit menu, but most don't use it. It's really there so you can use the desk accessories that let you do cutting and pasting (the Note Pad and Scrapbook). In most of the MacDS programs that have an Edit menu (except the Edit program), the menu stays dimmed unless you have a desk accessory on the screen.

Let's follow an assembly process and see how MacDS uses its various file types. We will assume that we are creating an application called *filter*.

The process starts when we use the editor to create the filter.Files, filter1.Asm and filter2.Asm. Filter.Files contains the names of the source files for the assembler to assemble—filter1.Asm and filter2.Asm. The filter1.Asm file contains the assembler source code for our program. The filter2.Asm file has assembler macros to define some of the resources that the filter program will need.

While we are in the editor, we also create the filter.Link file, which contains linker control commands and the filter.R file. Filter.R contains the source for the resource file that we will create with RMaker.

In figure 12.5, we show some of the files used during the program development process and the programs that use them. Figure 12.5 is a type of diagram that we will use to illustrate the behavior of several of the programs in the Macintosh Development System. It's called a data flow diagram. The lines with arrows represent data (in this case, files). The boxes represent programs that process the data. Data that is used primarily for controlling the operation of the program is shown with an arrow going into the top of the program's box. Other data goes in the side of the box. After we exit the editor, we run the assembler. It creates the relocatable object files, filter1.Rel and filter2.Rel. The assembler may also produce several text files that are not shown in the diagram, depending on which

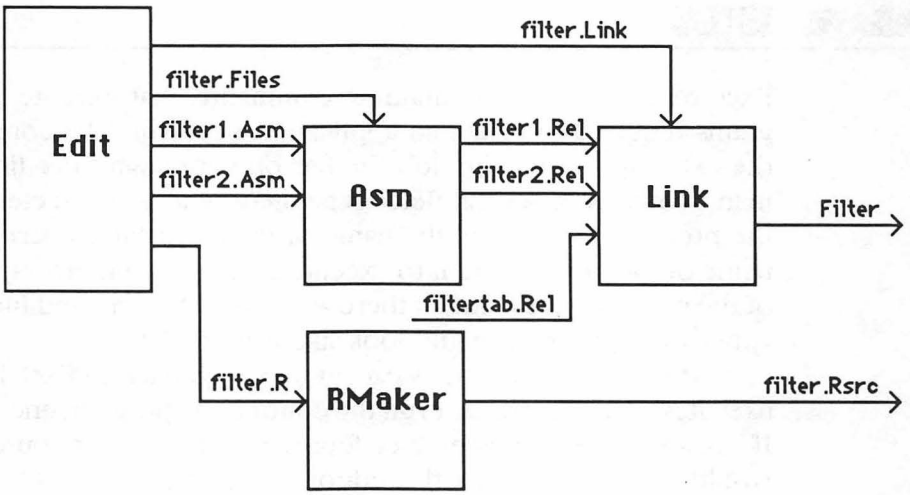


Figure 12.5 MacDS programs and files

assembler options we set (filter1.List, filter2.List, filter1.Sym, or filter-2.Sym). If the assembler detects errors in our source files, it will produce error listing files, filter1.Err and filter2.Err.

Next, we run Link. It reads link control commands from the filter.Link file. It reads the filter1.Rel and filtertab.Rel files, links them (resolving external references), segments the filter program's executable code, and adds the resources compiled by the assembler in the filter2.Rel file. The filtertab.Rel file was created by a previous assembly.

The linker created the application file for our filter program. If we look on the desk top now, we find our program with its icon. We still are not ready to run the program, though. We need to generate the resource file.

We run RMaker and give it the filter.R file as input. RMaker reads the resource definition statements in filter.R and creates the resource file, filter.Rsrc.

Now that we've completed the process of taking a program from source code to an executable stand-alone application, let's take a closer look at each of the programs that we used.

But, first, a digression to look at a program we didn't use, Exec. We did the whole assembly process manually just to see how it works, but if we had used a command file, Exec could have done the entire process for us.

EXEC

Exec reads a text file containing commands that execute the other programs required to create an application program. The command file has the extension .Job. The .Job file has one command per line. Each command consists of several fields separated by a tab character: the name of the program to execute, the name of the input file for that program, the name of the next program to execute if there are no errors, and the name of the program to execute if there are errors. A command file to create our application program would look like listing 12.1.

After each command is executed, we go back to Exec if the program executes without errors or run the editor if the program encounters errors. If, for example, the assembler found an error in our source code, Exec would take us back to the editor, open the source file that had the problem, and open the error file listing produced by the assembler.

When Exec executes, we get a set of menu titles in the menu bar. Looking at each menu should give us a good idea of how to run Exec (figure 12.6).

The File menu looks reasonably familiar. Your choices are to open a job file, open an application, or quit. If you choose the Open Job File item, you get a Minifinder window that displays all of the .Job files. Once you

Listing 12.1 Exec Command File

```
Asm       filter.files        Exec      Edit
Link      filter.link         Exec      Edit
RMaker    filter.R            Exec      Edit
```

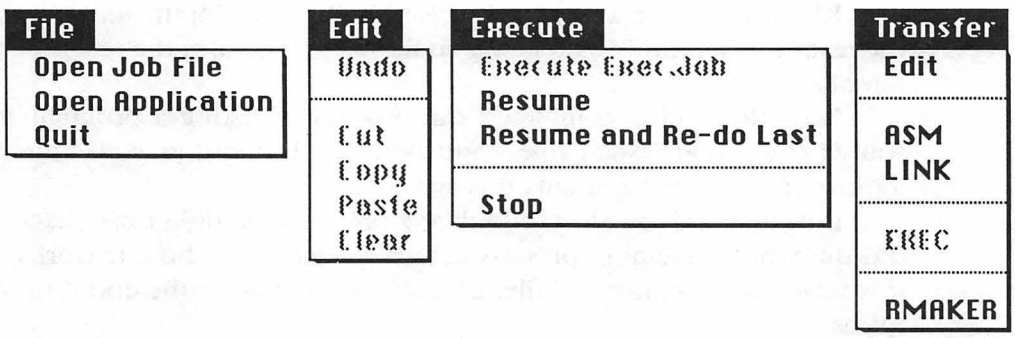


Figure 12.6 Exec menus

have selected a .Job file, you click the Execute button and you're off and running.

If you choose Open Application instead, the Minifinder window displays all of the application programs on the disk, including all of the MacDS programs, other Macintosh applications (Macwrite, Fontmover, and so on), and any applications you have created. The Open Application item in the File menu lets you execute an application program without going back to the Finder.

Let's jump over to the Transfer menu. It lists the other MacDS programs. If you choose one of the items from this menu, MacDS executes that program.

The Edit menu becomes active only when we open a desk accessory.

The Execute menu is a little more interesting. It lets us resume a .Job file that stopped because of an error. We can resume at the next step in the .Job file (by choosing Resume), or we can execute the failed step again and proceed from there (by choosing Resume and Re-do Last).

EDIT

The editor can have up to four files open simultaneously, each in its own window—this feature can cut down considerably on the amount of time it takes to go through the program assembly cycle. Usually it takes a number of passes through the assembler to get a clean assembly of a new program. By allowing you to look at the assembler error file and the assembler listing while you are editing the source file, the editor eliminates the necessity of your making listings. Just think, you may never have to sit and wait for that printer to crank out listings again!

Multiple Edit windows are also handy for copying sections of code from other listings without leaving the editor (figure 12.7).

The active Edit window has a horizontal scroll bar, a convenient feature when those long comments start to run into the right margin.

Looking at the menu bar, you would almost think you were in MacWrite—the only visible differences being a Size menu instead of a Style menu and the addition of the Transfer menu. There are some major differences in the way the two editors operate, though. Edit displays your text in whatever style and font you select, but it stores the file as "text only" with no font information. Edit has a number of features designed specifically for programmers. You can see some of those features by going through the menus.

The only surprise in the File menu is the file name in the Close item (figure 12.8). The Close item lists the name of the file in the active window. If you have several windows open and close one, the next one on the desk

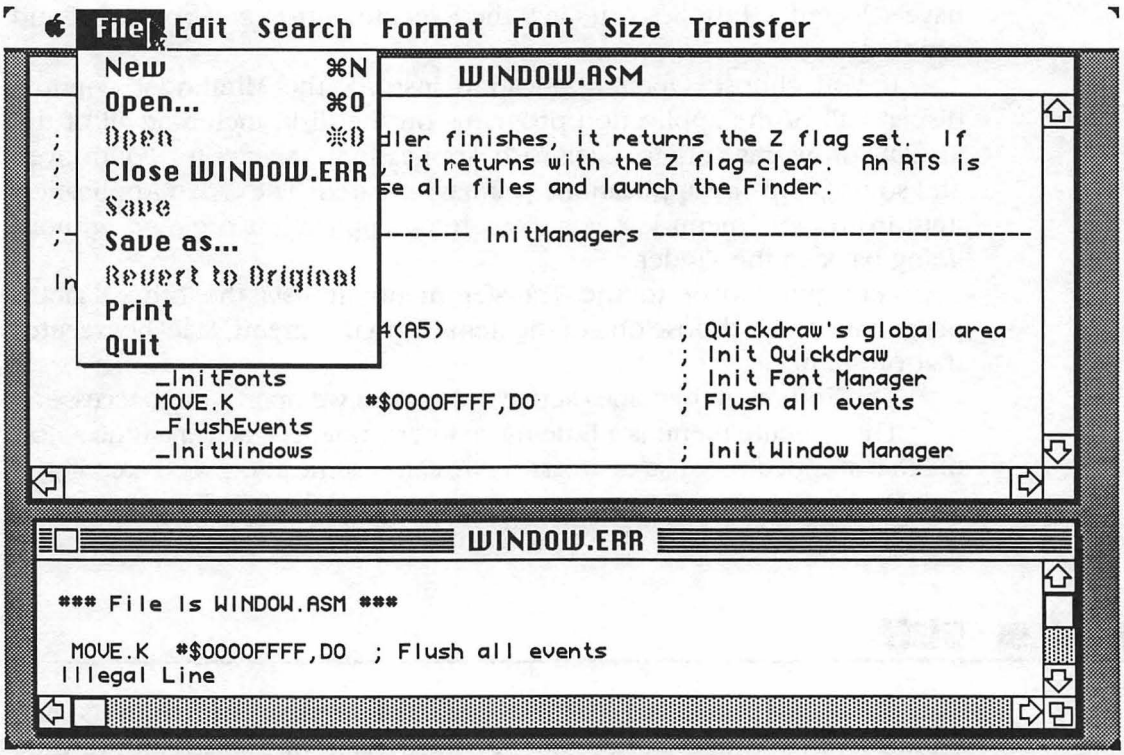


Figure 12.7 Edit windows

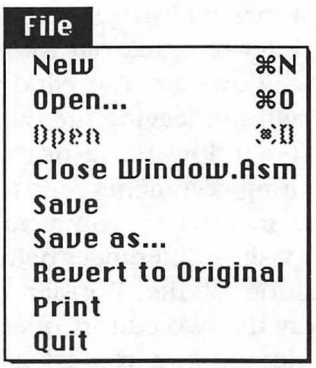


Figure 12.8 File menu

top becomes the active window—and its file name appears in the menu's Close item.

The Edit menu has the usual items and three new ones—Align, Move Left, and Move Right (figure 12.9). The new items are used to line up blocks of code. If you select a block of program statements and then choose Align from the menu, all of the statements are aligned on the left. Then choosing Move Right or Move Left moves the entire block of code.

The Search menu works just like the Search menu in MacWrite. The only difference is the Hide Find item in the menu; it gets rid of the Find dialog box (figure 12.10).

The Search dialog box looks almost the same as the one in MacWrite.

The Format menu has several items of interest to programmers (figure 12.11). Edit has tabs set at regular intervals, and you can change that tab interval by choosing Set Tabs from the Format menu. The Auto Indent item

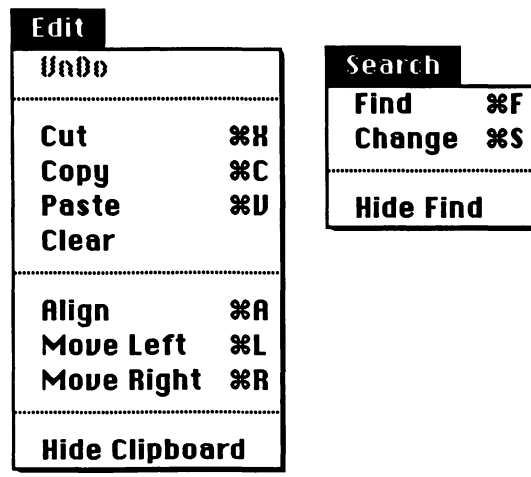


Figure 12.9 Edit and Search menus

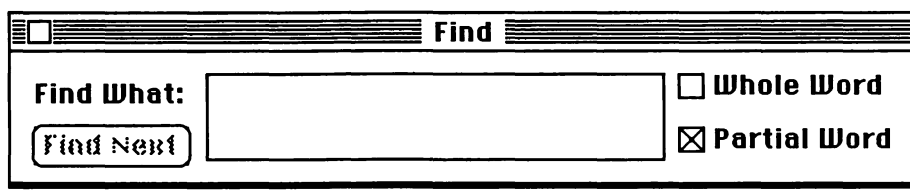


Figure 12.10 Find dialog box

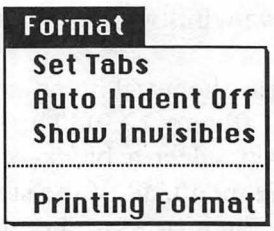


Figure 12.11 Format menu

toggles the auto indent feature. When Auto Indent is on, the cursor doesn't go all the way to the left margin when you type return; it goes to the last tab stop that you used.

The Show Invisibles item makes tabs, carriage returns, and other invisible characters visible on the screen. The Printing Format item brings up the Printing Format dialog box.

You use the Font and Size menus to select the typeface in which Edit will display your text (figure 12.12). These menus affect the display only; no font information is stored in the file. For programming, the Monaco font is probably best; it is the only monospaced font in the Macintosh. If you use one of the proportional-spaced fonts, a lot of things won't line up the way they do with a monospaced font, and the program will be hard to read.

Also, if you are using that plain, monospaced Monaco font, there's no reason not to print in draft mode instead of standard or high-resolution mode. Draft-mode printing is much faster, and the characters look very much like Monaco characters.

The Transfer menu is the same in all of the MacDS programs except the editor; in the editor, the name of the file you are editing shows up in

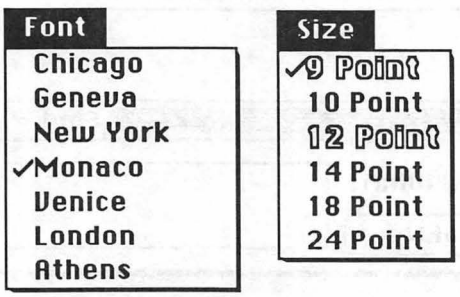


Figure 12.12 Font and Size menus

the Transfer menu's ASM item (figure 12.13). If you choose to transfer to the assembler, the Transfer menu assembles the file you have been editing.

ASM

This section provides an introduction to the assembler and briefly outlines its capabilities and syntax. You will find the detailed information about the assembler in the tables, not in the text. This section should give you an idea of what to expect from the MacDS assembler.

The assembler reads a file containing assembler programming instructions and produces a relocatable object file. It also can produce a listing file, an error file, and a symbol file (figure 12.14).

You may speed up the assembly process by supplying a packed symbol file from a previous assembly of the same program.

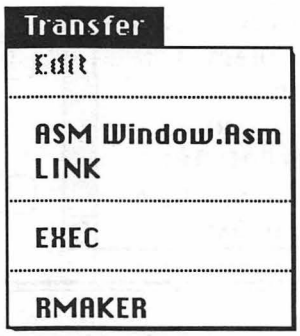


Figure 12.13 Transfer menu

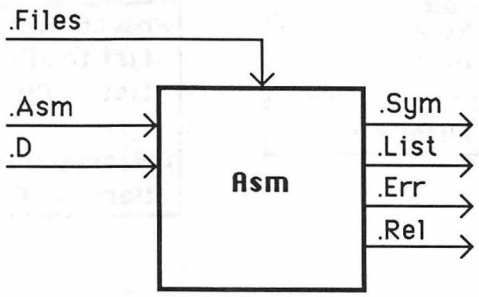


Figure 12.14 The assembler and files

Instead of an assembler source file, you can supply Asm with a file that contains a list of assembler source files to be assembled.

If you start the assembler by double-clicking its icon, you use the assembler's File menu to select the input file. Choosing Select File from the menu causes the assembler to put a Minifinder window on the screen so that you can select a file to assemble (figure 12.15).

The Filter by Time item in the File menu tells the assembler to restrict your input file selection to files that you have modified since they were last assembled (figure 12.16).

The Options menu selects a listing option and an output option. The output option determines whether or not the assembler puts information in the object file that the linker can use to generate a link map. Assembler directives in the source file can override both the listing and output options.

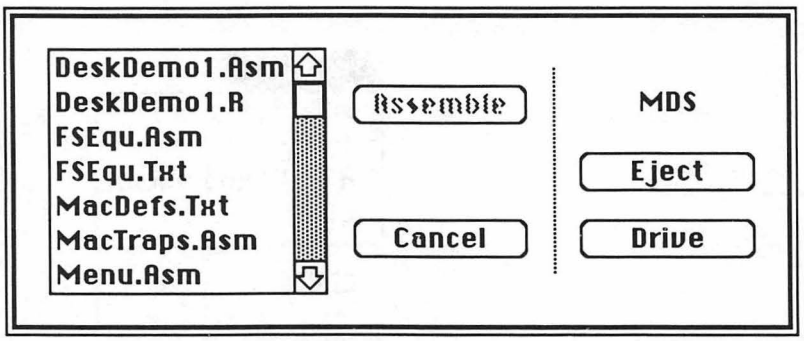


Figure 12.15 Assembler Minifinder

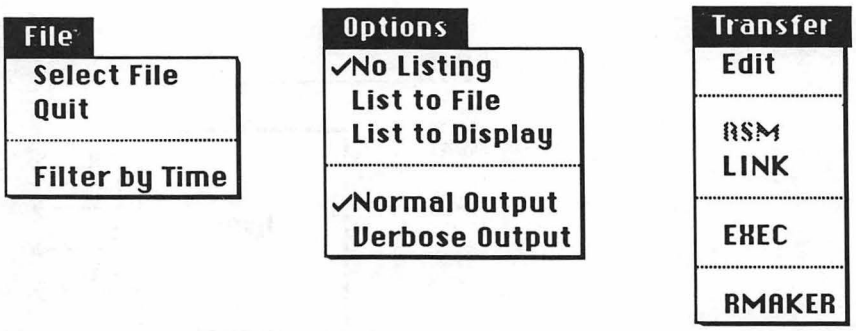


Figure 12.16 Assembler menus

The MacDS assembler is a full-feature macro assembler that supports the complete 68000 instruction set and uses Motorola standard mnemonics. The assembler also supports the Macintosh segmentation scheme by checking jump and branch instructions for references outside of the current segment. Some branch instructions that use absolute addressing cannot transfer control outside of the segment and thus produce an error message. The assembler converts other jumps and branches using absolute addressing to PC relative addressing.

Illegal Absolute Branch Instructions

BRA.S Address
BSR.S Address
Bcc Address
Bcc.S Address

Comment lines begin with an asterisk in column 1. A semicolon at any point in a program line indicates that the rest of the line contains a comment (listing 12.2).

Symbols may contain uppercase or lowercase characters, numbers, underscores, periods, or dollar signs, but neither a dollar sign nor a number may be the first character. That placement would conflict with the use of the dollar sign to designate hexadecimal numbers. A label is a symbol whose value is an address, usually a location within the program's executable code. A label either starts in column 1 or is identified by a trailing colon.

An asterisk can be used where you would expect an expression that evaluates to an address. The asterisk represents the address of the current location.

Listing 12.2 Label, Comment, and Expression Example

```
*
* This is a comment line, no program statements appear here
*
  Label1: MOVE.L    A0,OldAddr      ;this is a comment too
          MOVE.L    NewAddr,A0      ;get a new address
Label2    EQU       *               ;label 2 is right here
MOVE.L    NewAddr+15*(offset-1),A1 ;address expression
MOVE.L    *,A4            ;address of previous instr in A4
```

Expressions are combinations of symbols, strings, logical operations, and arithmetic operations. Arithmetic operators are +, −, *, /, >> (shift right), and << (shift left). The logical operators are ! (or) and & (and).

The assembler uses assembler directives in the source program to control the assembly process, control the listing, specify external references, allocate storage, and define symbols. The assembler directives are:

INCLUDE	Include another source file
STRING__FORMAT	Select assembler or Pascal-style strings
IF THEN ELSE	Control conditional assembly
MACRO	Define a macro instruction
END	Define the end of the source file
.DUMP	Write a symbol table listing in a .Sym file
EQU	Assign a value to a symbol
SET	Assign a temporary value to a symbol
REG	Define a register list
DC	Define a constant
DS	Allocate storage
DCB	Allocate a storage block
.ALIGN	Align next item on a word or long-word boundary
XDEF	Define a label as an external reference
XREF	Allow external reference to a label
RESOURCE	Identify start of resource definitions

The assembler directives that override the Options menu selection are:

- .NoList
- .ListToFile
- .ListToDisp
- .Verbose
- .NoVerbose

When the assembler encounters an INCLUDE directive, it reads assembler language statements from the included file until it reaches the end; then it continues reading statements from the current source file. INCLUDE can also be used to specify the packed symbol files that the assembler will use. In listing 12.3, we include a packed symbol file that

Listing 12.3 Include Example

```
INCLUDE   ToolEqu.D
INCLUDE   FilterEqu.Asm
```

contains toolbox equates and an assembler source file that contains our own equates.

The standard Motorola notation for registers that was presented in chapter 11 still holds:

A*n*	Address register *n*
D*n*	Data register *n*
SP	Stack pointer (A7)
SR	Status register
CCR	Condition code byte of status register
PC	Program counter

The MacDS assembler uses the Motorola mnemonics and addressing syntax as outlined below. Instructions have a base mnemonic with a possible extension indicating operand size. The base mnemonics are in the 68000 instruction summary tables at the end of chapter 11.

Instruction Mnemonic Examples

```
MOVE.B    index,D0  ;move byte to D0
MOVE.W    count,D1  ;move word to D1
MOVE.L    oldptr,A0 ;move long word to A0
```

In the following examples of address modes, *expr* means an assembler expression.

D*n*	Data register direct
A*n*	Address register direct
(A*n*)	Register indirect
(A*n*)+	Register indirect with postincrement
−(A*n*)	Register indirect with predecrement
expr(A*n*)	Register indirect with displacement
expr(A*n*,A*n*)	Register indirect with index
expr(A*n*,D*n*)	Register indirect with index
expr	Absolute or relative
expr(PC)	Relative with displacement
expr(PC,A*n*)	Relative with index
expr(PC,D*n*)	Relative with index

LINK

The linker creates an application or resource file under the control of the commands in the .Link file. The linker can put program code segments and resources in the application file's resource fork and data in its data fork. It can also create a resource file (figure 12.17).

The linker normally reads relocatable object files (.Rel files), resolves external references, and creates an application program file. The linker also adds resources that will be contained in the application program. Those resources can be created by the assembler or the resource compiler and are in their own .Rel files. Note that resources read by the linker must be in .Rel files, even if they were created by the resource compiler.

In figure 12.18, the filter1.Rel and filtertab.Rel files are program object files, and filter2.Rel is a resource object file. All three were created by the assembler.

You control the linker with three menus—the File, Options, and Transfer menus (figure 12.19). The File menu gives you two choices, Select File (the link command, or .Link, file) or Quit. The Transfer menu is the same as that in other MacDS programs.

The Options menu lets you set two options, but both can also be controlled by commands in the .Link file. The Normal/Verbose Map option determines how much information is included in the link map. The Undefined Illegal/OK option determines whether or not the linker treats undefined symbols (unresolved external references) as errors.

Commands in the .Link file tell the linker the names of your relocatable object files (code, resource, and data .Rel files) and the name of your output file; the commands also supply parameters to control the linking process. You can include comment lines in the link control file; a comment line begins with a semicolon.

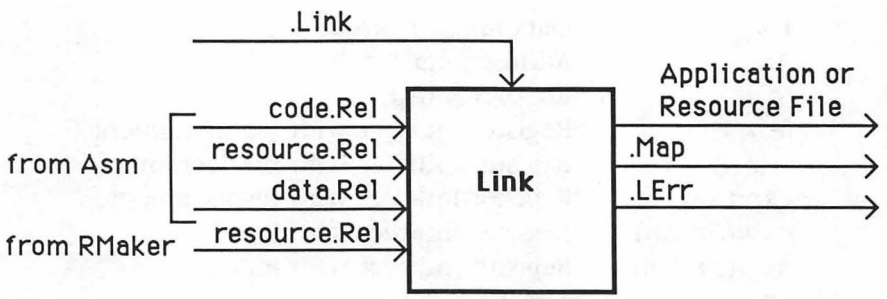


Figure 12.17 Linker and files

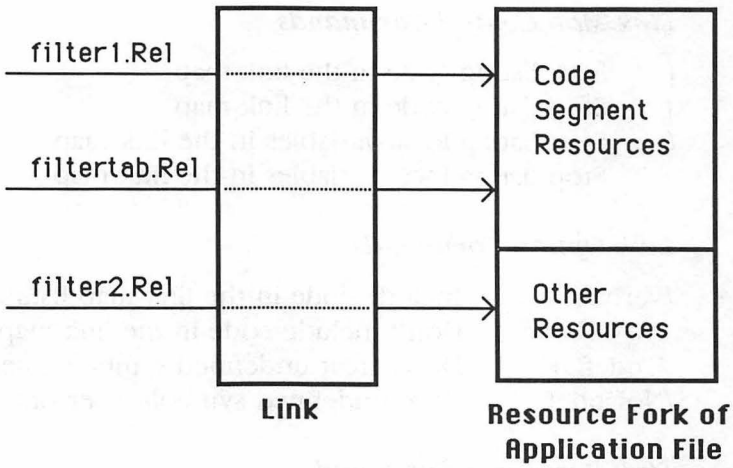


Figure 12.18 Linking an application

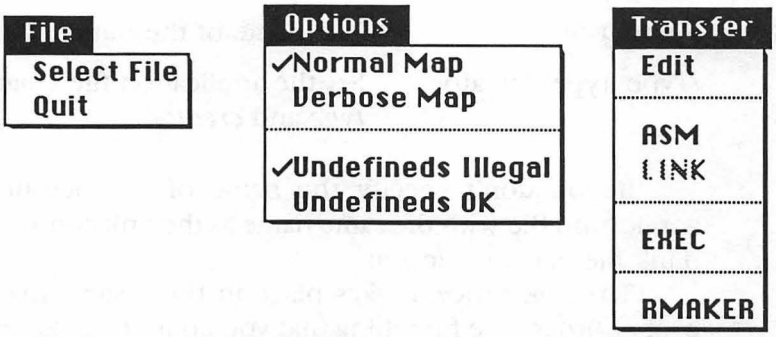


Figure 12.19 Link menus

You specify the names of your .Rel files simply by putting the file names in the .Link file; each file name goes on a separate line. The file name may or may not have the .Rel extension; if you leave it off, the linker supplies it. All of your input files must be .Rel files.

You also list linker commands in the .Link file, one command on each line. Link commands can set link options, control what goes into the link map, and control what goes into the output file.

Link Map Control Commands

[	Start listing code in the link map
]	Stop listing code in the link map
(	Start listing local variables in the link map
)	Stop listing local variables in the link map

Link Option Commands

/Verbose	Include code in the link map listing
/NoVerbose	Don't include code in the link map
/UndefOK	Don't treat undefined symbols as errors
/NoUndef	Treat undefined symbols as errors

Link Parameter Commands

!symbol	The program execution starting point is the location defined by *symbol*
/Globals address	Start the program's global area at *address*
/Output file	The name of the output file is *file*
/Type 'type' 'creator'	Set the application file's signature and type to *type* and *creator*

If you don't specify the name of the output file, it will be an application file with the same name as the link control file but without the .Link file name extension.

The link process takes place in three steps that must occur in the proper order. The first thing that you do in the linker control file is tell the linker to read all of the program code files so that it can produce the code segments for the resource fork of the output file. Next, you specify the .Rel files that contain resources to be included in the resource fork of the output file. Finally, you tell the linker to read the .Rel files containing the application's data and build the data fork of the ouput file.

The link control commands are as follows:

<	Start a new segment
/Bundle	Set the Bundle bit in the application file's directory entry
/Resources	Begin adding resources to the output file
/Data	Begin adding data to the data fork of the output file
$	Mark the end of the .Link file

The resource and data steps are optional; you include them only if you plan to have resources (other than code segments) or data in the output file.

In listing 12.4 we can see what the link control file for our application program would look like.

We didn't specify an ouput file name, so our output file will be called *filter*. Our program uses files created by the editor, so we set its signature to 'EDIT'; the file type is 'APPL' (an application file). We did not include any data in the application file.

We wanted the code listed in the link map, so we set the Verbose Map option. First we read a code file, filter1.Rel; then we started another segment and put the code from filtertab.Rel in it. After producing the code segments, we started creating the resource portion of the application and read the resources from filter2.Rel.

While the linker is executing, it displays what it is doing in three windows (figure 12.20).

PACKSYMS—THE SYMBOL PACKER

Building the symbol table takes a lot of time for the assembler, so we can speed the assembly process by reusing the symbol table. If we are going to recompile a program and have not added any new symbols, we can use the symbol table generated by the last assembly. A packed symbol file is also commonly used for equate files that rarely change.

The symbol table that the assembler creates is a text file that can be read by people, not by the assembler; the assembler stores its symbols in

Listing 12.4 Link Control Commands Example

```
/Verbose
/NoUndef
/Type 'APPL'   'EDIT'
[
filter1.Rel
<
filtertab.Rel
/Resources
filter2.Rel
$
```

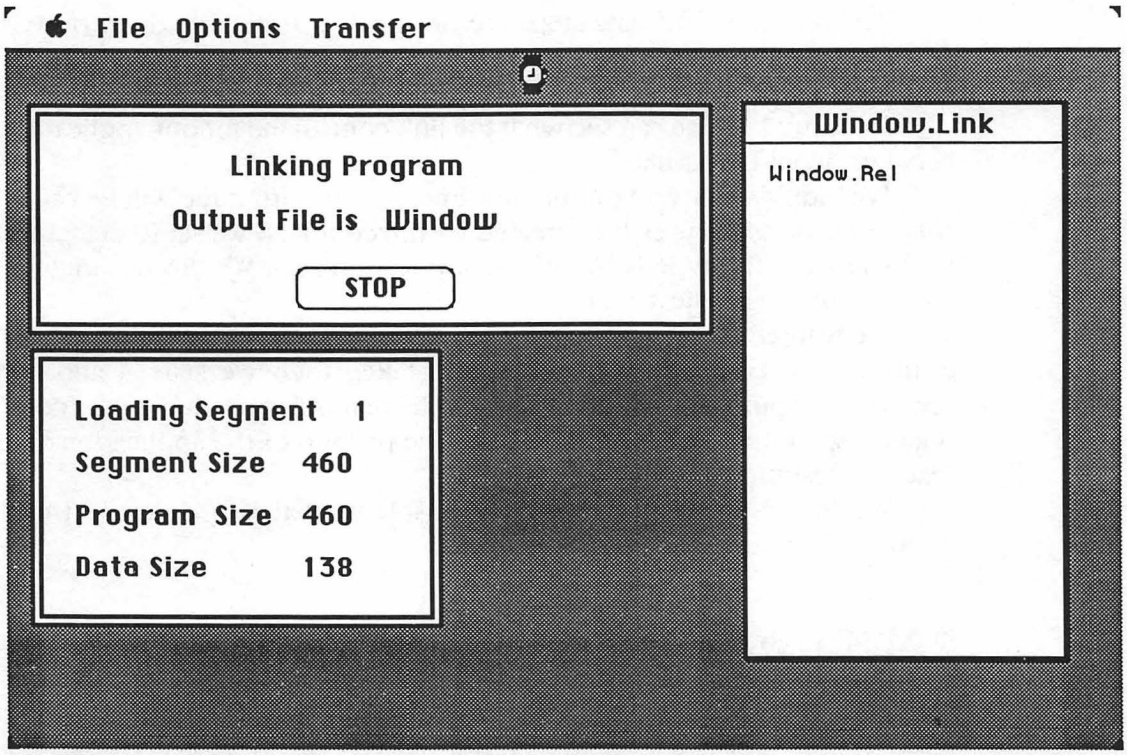


Figure 12.20 Link windows

a more compact format. You can convert the text-format symbol file (a .Sym file) to the packed format used by the assembler (a .D file) with the PackSyms program (figure 12.21).

You start PackSyms by double-clicking its icon. PackSyms doesn't need a control file; you control it with menu selections (figure 12.22).

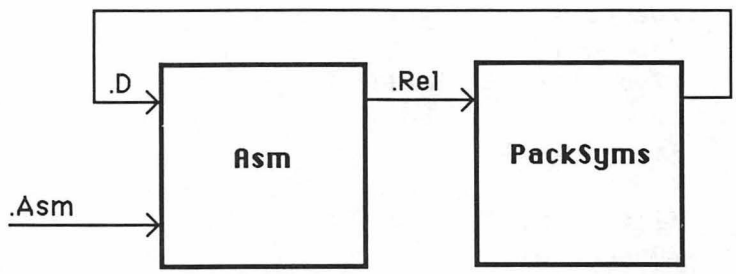


Figure 12.21 PackSyms and files

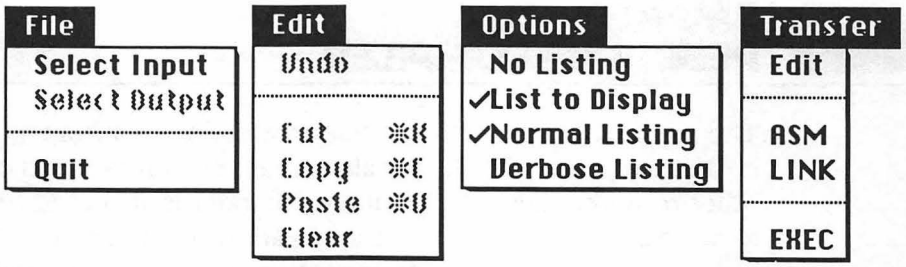


Figure 12.22 PackSyms menus

The Options menu controls the listing output of PackSyms. There's no option to list to a file or the printer, just to the display.

To specify the input file name, you choose Select Input from the File menu. PackSyms puts a Minifinder window on the screen, displaying all of the .Sym files. You select the file name from the Minifinder display and click the Pack button. PackSyms reads the file and creates a packed symbol table.

PackSyms stores the packed symbol table in memory and waits for you to choose Select Output from the File menu to tell the program where to put the table. When you make that menu selection, PackSyms puts up a dialog box with the default output file name already in it (figure 12.23). You can type over that name and specify a name of your own choosing. Clicking the Write button causes PackSyms to write the packed symbol table to the output file.

If you quit PackSyms without writing the table to the output file, the program discards the packed symbol table.

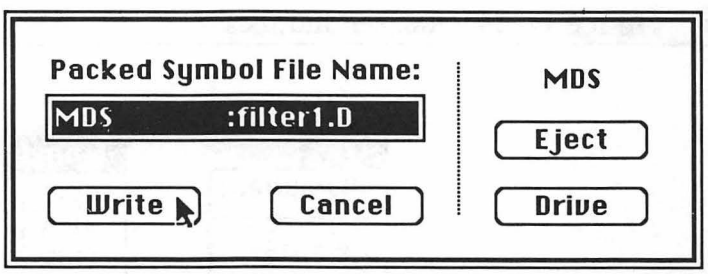


Figure 12.23 PackSyms Output dialog box

RMAKER—THE RESOURCE COMPILER

The resource compiler reads resource definitions from a control file and creates a resource file. It may also obtain resources for its output file from other resource files. The resource compiler is also used to add resources to an existing resource file or application file (figure 12.24).

The resource compiler creates resources from definitions in its control file (.R file) or obtains resources from other resource files. It can produce files in the application format (no file name extension), resource format (.Rsrc), or relocatable object format (.Rel). It produces an application-format file only when adding resources to an existing application.

RMaker is easy to operate; it has only two menus (figure 12.25). The File menu has items to compile a resource definition file, to limit your choice of resource definition files to those with the .R file name extension, and to quit the program.

Selecting the Compile item from the File menu gets you a Minifinder window so you can select the resource definition file. Once you select the

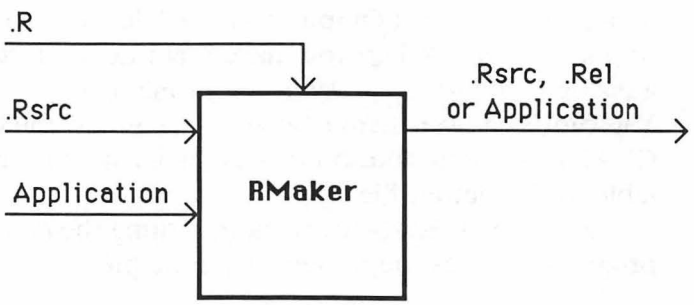


Figure 12.24 RMaker and files

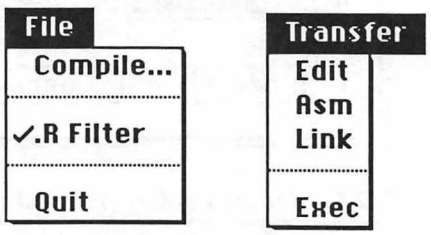


Figure 12.25 RMaker menus

file, you click the Open button in the Minifinder window; RMaker then compiles the resource definition file.

A resource definition file contains resource definition and RMaker control statements. RMaker interprets any line beginning with an asterisk as a comment line. You can place a comment after a resource definition statement on any line if you precede the comment with two semicolons.

The first statement in the resource definition file specifies the output file. It can be a file that RMaker creates or an existing file to which RMaker will add resources. If the output file is to be an application, it should have no file name extension. Otherwise, you may specify .Rsrc or .Rel as the file name extension. You must precede the name of an existing file with an exclamation point so RMaker will know that it must add to that file, not create it. If your output file is an application file, you must also supply the file signature and file type.

You follow the output file specification with resource definition statements. They can be TYPE statements to define resources or INCLUDE statements to include resources from .Rsrc files or application files.

RMaker has twelve predefined resource types, as shown below. One of them is a general resource type that allows you to create new resource types.

ALRT	Alert box template
BNDL	Application bundle
CNTL	Control template
DITL	Dialog or alert item list
DLOG	Dialog box template
FREF	File reference
GNRL	General (custom) resource
MENU	Menu template
PROC	Procedure
STR	String
STR#	A group of strings
WIND	Window template

You define a resource by coding its name in a TYPE statement and putting the resource parameters on the following lines. To fully explain all of the resource types used by RMaker is beyond the scope of this book, but you can find that information in the MacDS RMaker manual.

While RMaker executes, it displays the input file it is compiling and information about the output file in a window (figure 12.26).

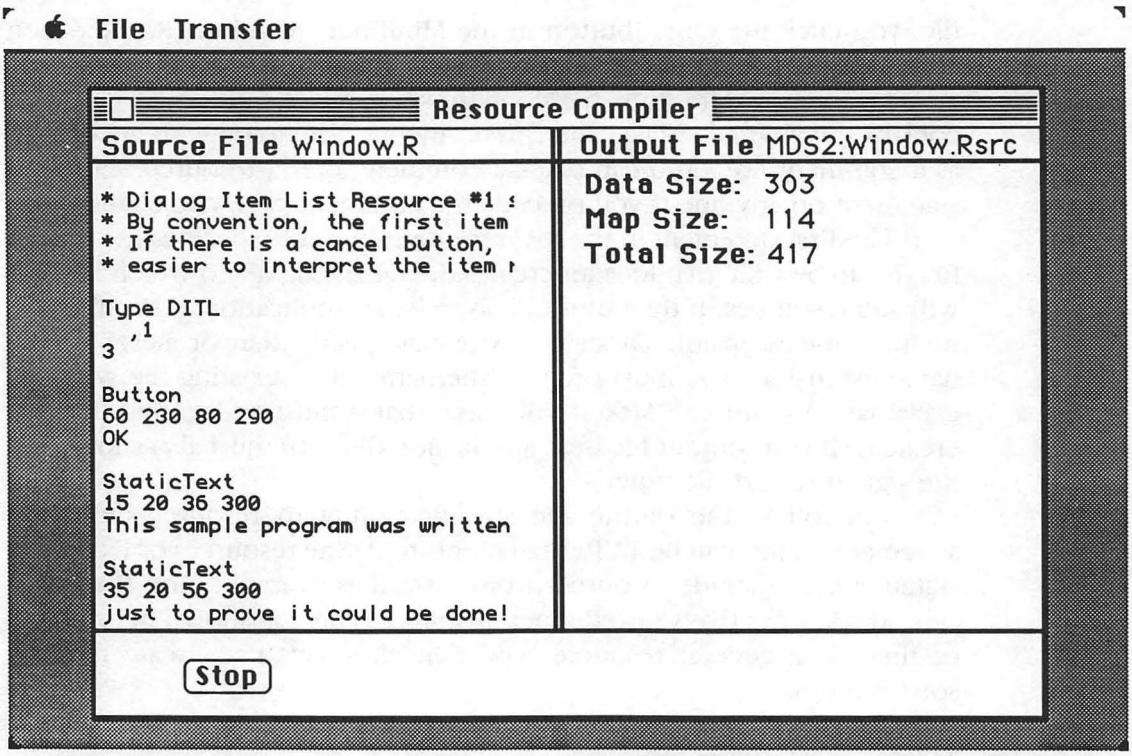


Figure 12.26 RMaker window

THE DEBUGGERS

A debugger is a software developer's tool that runs your application program and allows you to stop it at specific points of your own choosing, examine or change the 68000 registers, and examine or change memory locations. The Macintosh 68000 Development System debuggers do that much and more.

How do the debuggers manage to do that? They replace a lot of the 68000 exception vectors with vectors to their own code. Thus, when an exception occurs, the debugger gets control, not the application program's exception handler. Of course, after you tell the debugger to execute your program again, it picks up where it left off, with the exception handler.

You can generate an exception at any time by pressing the interrupt button. It's the programmer's button toward the rear of the machine on the left side. The front programmer's button doesn't generate an interrupt;

it resets the Macintosh, restarting the machine as if you had turned the power off and back on again.

MacDS has two types of debuggers, MacDB and MacsBug. MacsBug runs on a 128K or 512K Macintosh or on a Lisa with MacWorks. To control MacsBug and view its displays, you can use either the Macintosh or a terminal connected to the modem port or the printer port. MacsBug comes in several versions, depending on which model of Macintosh you are using and how you want to control the debugger.

MacDB is much easier to use than MacsBug and has some features that MacsBug lacks, but it requires two Macintosh computers to run it. A small program (called a *nub*) runs in one Mac with your application program. MacDB runs in the other machine, where it generates the displays and accepts commands. MacDB's major advantages over MacsBug are its use of windows and menus for display and control, and its symbolic debugging capabilities. By referencing a symbol file produced by the assembler, MacDB can accept symbols instead of hex addresses in commands and can display addresses as program labels and displacements.

The windows that MacDB uses include:

PC	Trace program counter, disassembled instruction display
Register	Display and alter registers
Examine	Display and alter memory
Breakpoints	Display breakpoints

Using the menus, you can run the program, trace (single-step through) a program in RAM, trace execution from ROM, run the program until it gets to a designated address, or start execution at a designated address. MacDB has other menu items that either search memory for a pattern or monitor the execution of traps. Menu items also allow you to select the format in which MacDB displays memory in the Examine windows. Memory can be displayed by MacDB in the following formats:

- Disassembled instructions
- Characters—a hex byte and the corresponding ASCII character
- Hex words
- Hex long words
- Pascal strings
- Linked lists
- Heap memory blocks

The MacsBug debugger has many of the same capabilities as MacDB, but for assembly programs developed with MacDS, it cannot read a symbol file and do symbolic debugging. MacsBug doesn't have windows, and you control it through line-oriented commands, not menus. Its major advantages are that it is small (as little as 12K for the version that works with a terminal) and that it doesn't require a second Macintosh.

The MacDS disk has different versions of MacsBug for the 128K Macintosh, the 512K Macintosh, the Lisa running MacWorks, and a Macintosh using a terminal to control MacsBug.

MacsBug can set breakpoints, run the program, stop the program, trace the program, trace to the end of a routine, start execution at a specified location, or execute the program until it hits a particular breakpoint *n* times. It can display and alter the 68000 registers, alter memory locations, and display memory in a number of formats. Its memory display formats include:

- Hexadecimal
- I/O parameter block
- Window record
- Text edit record
- Disassembled instructions

MacsBug has a powerful set of commands for monitoring trap execution. In each trap command you specify a range of trap numbers, a range of addresses (in the program you are debugging), and a range of data values for register D0. A command specifies what action the debugger will take when the application program executes a trap and all of the conditions in the trap command are met—that is, the trap number is in the range of specified trap numbers, the trap was executed at an address in the application program within the specified address range, and the value in D0 is within the specified range.

The trap commands in MacsBug comprise:

AB	Break
AT	Trace and display the trap
AH	Check the heap
HS	Scramble the heap
AS	Check a memory zone for change
AX	Eliminate existing trap commands

When you enter one of these trap commands, nothing happens right away, and nothing will happen when the system executes traps unless all

of the conditions in the command are met. When all of the conditions are met, MacsBug will take the action in the trap command description.

The AS command does something that you usually see only in $20,000 "in-circuit emulation" machines; it checks to see if the specified memory area has changed and generates a break if it has. Every time the 68000 executes a trap, MacsBug does a checksum calculation on the specified memory range. If any data in that memory range has changed, MacsBug generates a break.

MacsBug's heap commands provide you with ways to examine and check the consistency of the heap. The heap commands include:

HX	Switch between system heap and application heap
HC	Check the consistency of the heap
HD MASK	Dump (display) the heap contents
HP MASK	Print the heap
HT MASK	Just display the heap dump summary

The MASK parameter selects which blocks MacsBug will include in the heap display. You can select relocatable blocks, nonrelocatable blocks, free blocks, all resource blocks, or resource blocks of a particular type. The heap print command works only with the versions of MacsBug that are controlled from a terminal.

CHAPTER

13 DISKS

The Disk Drives

The Disk Format

THE DISK DRIVES

The disk drives in the Macintosh are 3½-inch floppy disk drives manufactured by Sony. Currently they are available only in the single-sided 400-KB version, but when Apple Computer can get sufficient quantities of double-sided drives, the 800-KB version will be available for the Macintosh.

Sony makes other versions of the 3½-inch drive that are not compatible with the drive that they make specially for Apple. The Apple version has a variable-speed motor and holds more data than the standard Sony drive.

Floppy disks record data in tracks that are concentric circles on the disk surface. The recording head is attached to an arm that moves back and forth over the disk so that the head can access different tracks (figure 13.1). Because the outside tracks have a larger radius, they are longer than the inside tracks (figure 13.2).

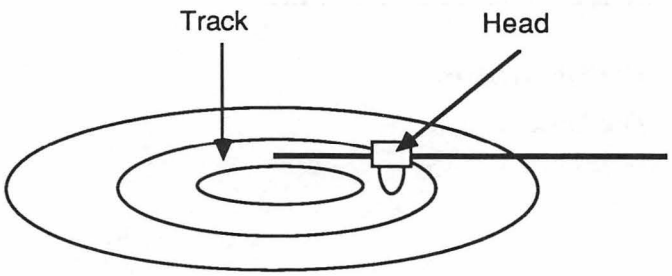


Figure 13.1 Disk drive

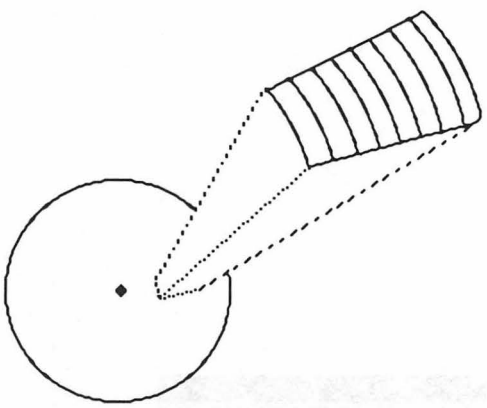


Figure 13.2 Disk tracks

The disk drives can transfer information at a rate of 62.5 KB per second. The drive has a variable-speed motor that rotates the disk at speeds of 390 to 605 RPM (the high speed is about twice the rotational speed of 5¼-inch drives). The disk controller in the Macintosh tells the drive how fast to rotate the disk. The controller sets the drive for slower speeds on the outside tracks of the disk and higher speeds on the inside tracks. By turning the disk more slowly for reading or writing the outside tracks, the controller enables the disk to store more information on the outside tracks than would be possible if the drive had just one speed.

Most floppy disk drives must make a compromise between the best speed for the inside tracks and the best speed for the outside tracks. Ideally, you want the disk to always have the same linear speed under the read/write head. If the disk rotates at one speed, the outer tracks pass under the head at a higher speed than the inside tracks. They both make one revolution in the same time period, but the outside tracks are longer and therefore move at a higher linear speed.

By slowing down the drive when accessing outside tracks, the Macintosh drive can write more data on them. The amount of data that you can get on a track is limited by the number of flux transitions that you can reliably record on the disk. In plain English, that means that the disk can record up to some maximum number of bits per inch on a track. You want to take advantage of the maximum allowable bits per inch to get the most data on the disk.

The number of bits per inch that you are actually recording depends on the rate at which the head writes data on the disk (bits per second) and how fast the disk moves past the head (inches per second). Divide the first by the second, and you get the data density in bits per inch. It's difficult to vary the rate at which you write data, so you are usually stuck with a fixed data rate (bits-per-second recording speed). If you choose the disk speed that puts bits on the inside tracks at the maximum density (bits per inch), the outside tracks are recorded at a much lower density because the disk moves under the head faster. If you slow down the disk drive on the outer tracks, you can write more bits per inch.

Each track on the disk is divided into sectors (figure 13.3). A sector contains a fixed amount of data. In the Macintosh disk, a sector contains 524 bytes. Of that 524 bytes, 12 are used for system information about the sector (the sector tag), so 512 bytes of each sector are available for holding data. The Macintosh disk has 800 sectors. At 512 data bytes per sector, that comes out to 400 KB on a single-sided drive.

The outside track has twelve sectors, and the inside track has just eight. Disk tracks are grouped into five areas, sixteen tracks to an area. All of the tracks of an area have the same number of sectors. Area 1 contains

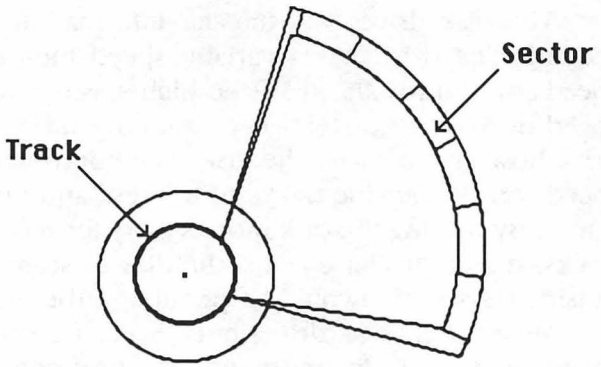


Figure 13.3 Disk sectors

the outermost tracks, and area 5 has the innermost tracks (figure 13.4). See table 13.1.

When you first put a new disk in the Macintosh, the Mac asks if you want to initialize the disk. A new disk lacks some of the information that the Macintosh disk controller needs in order to find the sectors on the tracks. Initializing the disk consists of writing information on the disk that identifies the start of sectors and the sector numbers.

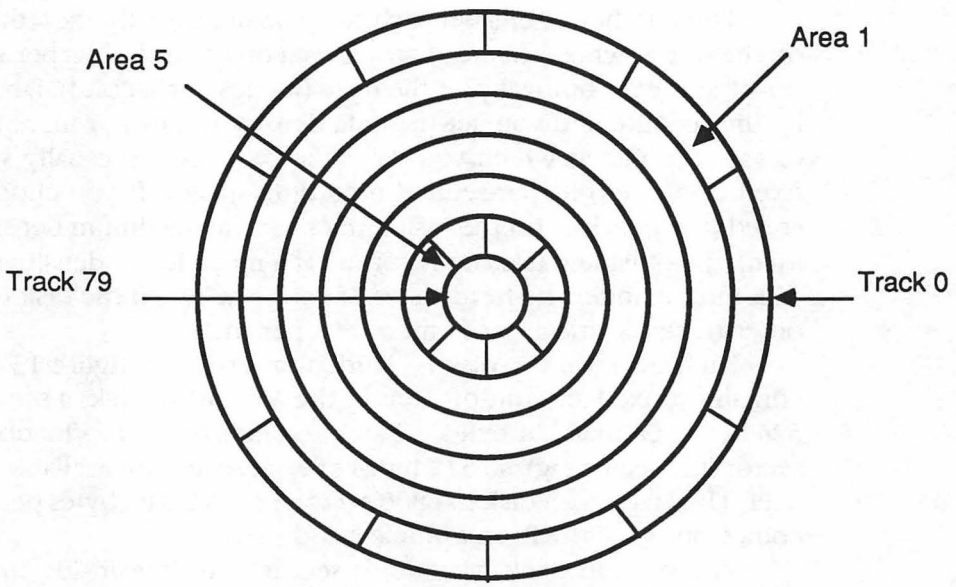


Figure 13.4 Areas

Table 13.1 Areas, Tracks, and Sectors

Area	*Tracks*	*Sectors per Track*	*Bytes per Track*
1	0–15	12	98304
2	16–31	11	90112
3	32–47	10	81920
4	48–63	9	73728
5	64–79	8	65536

THE DISK FORMAT

Not all of the disk sectors can be used to store data or programs. The first two sectors contain system start-up information, and some other sectors store information about file locations on the rest of the disk. These sectors belong to a group of sectors devoted to disk volume, block allocation, and file directory information. Let's see how that information and the rest of the disk are organized.

The Apple documentation uses specialized terms when describing disk organization, so we will use the same terms. A *sector* is the smallest portion of a track that the disk controller can read or write. It contains a 12-byte tag field and 512 bytes for data. A *logical block* is the data storage portion of one sector (512 bytes).

An *allocation block* is an integral number of logical blocks. For instance, a disk could be organized into allocation blocks that each contain two logical blocks, for a total data storage of 1024 bytes (1 KB) per allocation block.

The first two logical blocks on the disk contain the system start-up information. The second two logical blocks make up the master directory block. The master directory block contains volume information and an allocation block map.

Logical Block	*Use*
0 through 1	System start-up information
2 through 3	Master directory block (volume information and block map)
4 through *n*	File directory
n + 1 through 799	Files

A *volume* is a single floppy disk. The volume information consists of the volume name and other information about the volume:

Bytes	*Field*
2	Date and time the volume was initialized
4	Date and time of last backup
4	Volume attributes
2	Number of files in directory
2	File directory's first logical block
2	Number of logical blocks in file directory
2	Number of allocation blocks on disk
4	Size of allocation blocks
4	Number of bytes to allocate
2	Logical block number of first allocation block
4	Next unused file number
2	Number of free allocation blocks
1	Length of volume name
1–255	Volume name

The volume attributes consist of two flags—one indicating that the volume is locked by hardware, the other indicating that the volume is locked by software.

The rest of the master directory block contains the allocation block map. The allocation block map has one 12-bit entry for each allocation block on the disk. That 12-bit entry is an unsigned integer with a value of 0 through 4095; the integer contains the block number of the next block in the file. If the block number is 0, the block is unused; if the block number is 1, the block is the last block of a file.

The file directory follows the master directory block and has one entry for each file on the disk. A file has a resource fork and a data fork, so the file-directory entry contains information about each:

Bytes	*Field*
1	Flags (entry used, and file locked flags)
1	Version number
16	Finder information
4	File number
2	Data fork's first allocation block
4	Data fork's logical end of file
4	Data fork's physical end of file
2	Resource fork's first allocation block
4	Resource fork's logical end of file

Bytes	*Field*
4	Resource fork's physical end of file
4	Creation time and date
4	Modification time and date
1	File name length
1–256	File name

Although the file-directory structure allows a file name to be 256 bytes long, the Finder cannot handle file names that are over 64 bytes long.

The Finder information field contains data that the Finder puts in the directory entry and uses for its own purposes:

Bytes	*Field*
4	File type
4	File creator (signature)
2	Flags (bundle flag and icon visible flag)
4	Icon location (in local coordinates of window)
2	Folder number

The folder number indicates which window or folder the file's icon is in. If the folder number is positive, it is the number of a folder. If the folder number is 0, the file's icon is the disk directory window; if −2, the icon is on the desk top; and if −3, the icon is in the trash window.

We need to look at the structure of a Macintosh file in order to understand what the logical and physical end-of-file fields mean. A Macintosh file really looks like two files because it has two forks: the data fork, accessed by the file manager, and the resource fork, accessed by the resource manager.

A file fork occupies a list of disk allocation blocks. A file-directory entry points to the first allocation block of the fork, and the allocation-block map entry for each block points to the next block in the file. The allocation block entry of the last block in the file has a value of 1. (There is no allocation block 1; allocation block numbering starts with 2.)

The allocation block map forms a chain of allocation blocks for each file fork. The first allocation block entry in the map has the number of the next allocation block (in effect, it points to the next block), and so on, to the end of the file. The allocation blocks of a file need not be in any order; they can be scattered all over the disk.

In figure 13.5, the first block of the data fork of our file is block 3. Each block points to the next until we come to the last allocation-block map entry, which contains a 1.

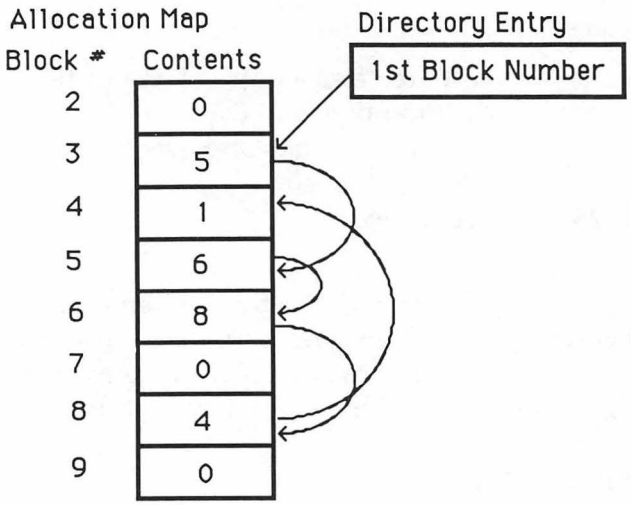


Figure 13.5 Allocation block chaining

A user's program may read or write at any point in the file, at an arbitrary byte. Numbering of bytes in the file starts with byte 0. Physical I/O is done in units of one logical block (512 bytes). The file manager buffers each logical block and takes care of deciding when to write an updated block or read the next one. As far as the application is concerned, the file is just a string of bytes.

The *logical end of file* is the byte number of the last data byte in the file. It won't be the last byte of the last logical block in the file unless the file just happens to be a multiple of 512 bytes long. The *physical end of file* is 1 plus the byte number of the last byte in the last allocation block of the file.

It seems that we have all the information we need to trace the chain of allocation blocks that make up a file and find its logical and physical end of file. So why do we need the tag field that precedes the data in each disk sector?

The logical blocks (disk sectors) of a file are chained together with one set of pointers, forward pointers. The pointers are not kept in the logical blocks themselves but in the allocation block map. If, through a hardware or software error, one of the pointers or the entire map is damaged, we have no way to repair the damage without the tag field. The tag field in each sector has information that enables the system to reconstruct a file with a broken sector pointer chain or a damaged allocation block map. The tag field contains the following information:

Bytes	*Field*
4	File number
1	Fork type (resource or data)
1	Attribute (open or locked)
2	Logical block sequence number
4	Modification time and date

Once the file manager locates a file in the directory by its file name, the file manager finds the file's number and after that refers to the file by number. If the file's logical block chain is broken, a program can read the tag fields of all of the sectors on the disk, select those with the right file number, and get enough information to reconstruct the file.

CHAPTER

14 COMMUNICATING

Communicating with the Mac

Communications Basics

The Communications Ports

The Serial Communications Driver

AppleTalk

COMMUNICATING WITH THE MAC

The Macintosh does the kinds of communications you would expect from a personal computer (like dialing into time-sharing services and exchanging files with other computers) and more. It runs a variety of terminal emulation programs and, with extra hardware, can emulate IBM 3270 terminals. Most individuals use the communications capabilities of the Mac to call time-sharing services like The Source, Compuserve, and the Dow Jones News/Retrieval Service. Businesses are more interested in networking with other personal computers, file servers, and the Apple LaserWriter (a laser printer).

The communications hardware on the Macintosh is also used to interface to the Imagewriter printer and local hard disks.

COMMUNICATIONS BASICS

We usually use the term *communications* to talk about any interfacing scheme in which the data is transferred in serial fashion, one bit at a time. Serial data transfer is used for communications services over telephone lines, interfacing to devices that use serial communications (printers), and most local area networks. When we communicate with a nearby device and do not use telephone lines, we can connect the serial communications port of one device directly to another's with a cable. When we communicate over the telephone system, we must use a modem to convert our data into signals that the telephone system can handle.

Data comes out of a serial communications port one bit at a time—a bit is represented by two voltage levels, one for a bit value of 1, the other for a bit value of 0. There are various industry standards that specify what those voltage levels must be.

The telephone system does not communicate using voltage levels; it transfers only tones (AC signals). If we are going to use the telephone system to transfer data, we need to convert the voltage levels coming from the Mac's communications ports to tones, and back again at the other end of the connection. That's what modems do. Most modems that you can buy for the Macintosh can also dial the telephone or automatically answer incoming calls.

The two most-often-used standards that describe the voltage levels and signals in a serial communications interface are RS232 and RS422. Both of these standards define the interface between a computer and a

modem. The interfaces that they define can be wired to allow communications between two computers without a modem, but that interface is really not defined in the standard. The standards call the modem the Data Communications Equipment (DCE). They call the computer, terminal, printer, or anything else that connects to the modem the Data Terminal Equipment (DTE). Some telephone-company documentation calls the modem a Data Set.

When you connect your Macintosh to a modem, the Mac is the DTE and the modem is the DCE. When you connect the Mac directly to another Mac or to another device like a printer, one must appear to be a DTE and the other must appear to be a DCE. Making a device that is really a DTE look like a DCE is really quite easy. It's done by switching a few wires around in the RS232 connector—either the connector on the cable or the connector on the device. No one likes to take a computer or printer apart to rewire its connector, so people almost always make that change in the connector on the cable.

When the Macintosh starts spitting bits out of a communications port, the device on the other end of the cable sees just a stream of bits. It needs to be able to identify the starting bit of each character. It also must know what the data rate is so it can identify individual bits in a stream of constant ones or zeros. There are two standard methods used to identify the data rate and bit times. Most low-speed devices use *asynchronous* communications methods to identify bit timing and the start of characters. Higher-speed communications devices and IBM 3270 terminals use a more sophisticated method called *synchronous* communications.

In asynchronous communications, the two devices that are communicating must agree on what speed they are using. They identify the start of a character with a start bit (more about that later).

In synchronous communications, the communicating devices have an extra signal—a clock signal—in the interface to identify bit times. Usually the DCE supplies the clock signal, but in some systems the DTE supplies it. Synchronous communications systems identify the start of a character by establishing byte synchronization. One station repeatedly sends a special unique character called a *sync character* to the other station. The other station can identify the start of the sync character and can then identify the start of subsequent characters by counting clock cycles (8 bits, or clock cycles, per character).

The Mac can do synchronous or asynchronous communications. When connected to hard disk drives or other devices that require a high-speed interface, the Mac uses synchronous communications. When connected to a modem, directly to another Mac, or to the printer, the Macintosh uses asynchronous communications.

Let's take a look at what comes out of the communications port when the Mac is doing asynchronous communications (figure 14.1).

Before the character starts, the interface data line is at the voltage level for a 0 bit. The character begins with a start bit that has the voltage level for a 1. The device receiving the characters sees the start bit and, since it knows the data rate, it knows when to look at the data line for each succeeding bit in the character. After the last bit in the character, the Mac sends a stop bit with a voltage level for a 0. The 0-level stop bit ensures that the data line is at 0 before the start of the next character. If it had stayed at the 1 level, the receiving device would not be able to recognize the start bit of the next character. It needs to see the transition from 0 to 1 to identify the start bit.

There is widespread disagreement in the computer industry over the rest of the format of an asynchronous character. Some devices send 2 stop bits instead of 1; some send 1½ stop bits. There's no standard character size either. Different devices use character lengths of 7, 8, 9, or 10 bits; most, however, use either 7 or 8 bits per character.

Many computers and terminals expect to see the data bits of a character followed by a parity bit as a check for communications errors. The device sending a character sets the parity bit to either a 1 or a 0, depending on how many 1 bits are in the data character. There are two different criteria used for setting the parity bit: even parity and odd parity. *Even parity* means that the sending device sets the parity bit to a 1 or a 0 so that there are always an even number of 1 bits in the character (including the parity bit). *Odd parity* means that there are always an odd number of 1 bits. The receiving device can count the 1 bits in each character, and if they are not what it expected (even or odd), the device knows that there was a transmission error.

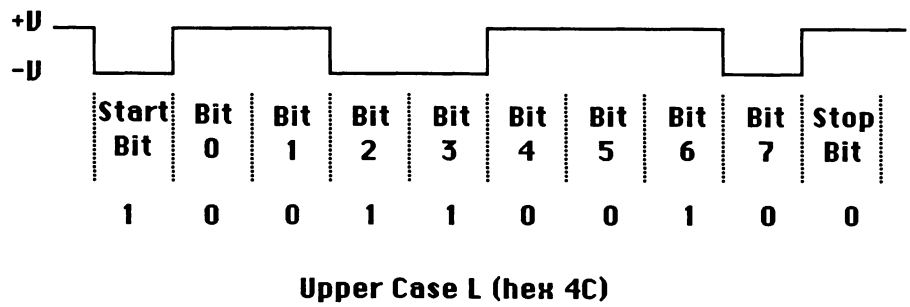


Figure 14.1 Asynchronous data character

A little thought about parity checking will reveal that it can detect only an odd number of bit errors. If a character arrives with 2 bits in error, it will pass the parity check.

We have identified four things on which the communicating devices must agree: the data rate (sometimes called the *baud* rate), the number of bits per character, the number of stop bits, and what kind of parity checking is used. Many modern communications controller chips can live with any number of stop bits, so that choice becomes less important.

It's the choice of the other three communications options that makes data communications so complicated for the user. The lack of standards means that the user must figure out how many data bits, how many stop bits, and what kind of parity to use. Fortunately, there is a growing movement to standardize on 8 data bits, 1 stop bit, and no parity. Though this trend is not official, many systems are now using those options. When in doubt, that's the first combination you should try.

The program that runs in the Mac must let you specify those options if it is to be able to communicate with a wide variety of other devices. If we look at the Compatibility Settings window from the MacTerminal program, we can see some of those options and others (figure 14.2).

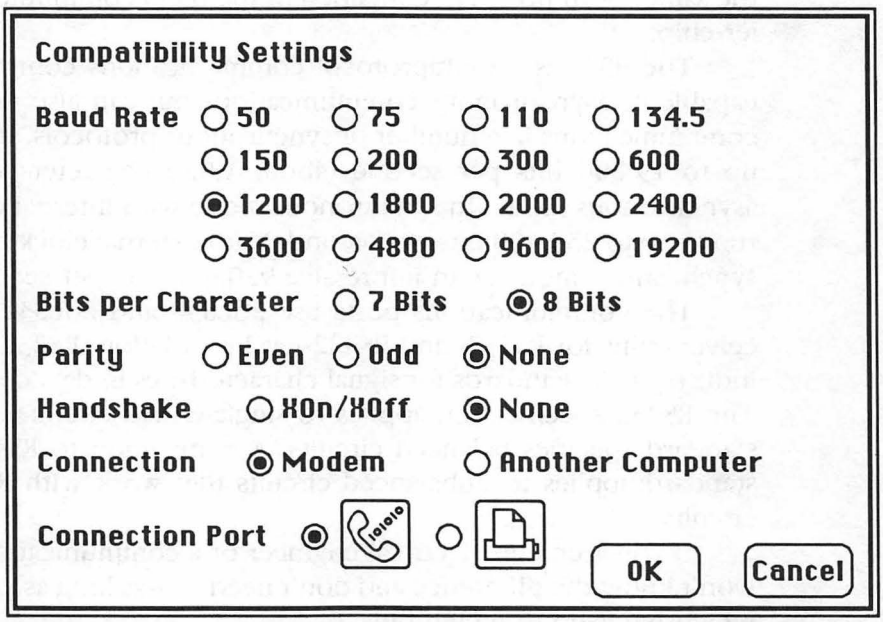


Figure 14.2 MacTerminal compatibility settings

The baud rate is the data rate in bits per second. Since devices send asynchronous characters with 8 data bits, 1 stop bit, and no parity bit—including the start bit, that makes 10 bits per character—you can get the data rate in characters per second by dividing the baud rate by 10.

One thing that we see in this window that we haven't discussed yet is the Handshake option. When two computers communicate, one of them may be temporarily unable to receive data because its buffers are full, because it is busy with disk I/O, or for some other reason. If data continues to arrive, it may be lost.

The XOn/XOff handshake option lets one computer tell the other to temporarily stop sending data. Sending an XOff character to the other computer tells it to stop transmitting. Sending an XOn character tells it to start again. If the handshake is to work, both devices must agree to use XOn/XOff.

THE COMMUNICATIONS PORTS

The Macintosh has two communications ports, one labeled for use with a modem, the other labeled for use with a printer. The ports are actually the same. Both ports are contained in the 8530 communications controller chip.

The 8530 is a multiprotocol communications controller chip. It is capable of asynchronous communications but can also do synchronous communications in a number of synchronous protocols. It can operate at up to 19,200 bits per second (about 1,920 characters per second) in asynchronous mode. In synchronous mode with internal clocking, it can run at up to 230,400 bits per second. With external clocking, it can run in synchronous mode at an impressive 920,000 bits per second.

The communications ports use 26LS30 and 26LS32 driver and receiver chips for RS232- and RS422-level translation. RS232 and RS422 are industrywide standards for signal characteristics in data communications. The RS232 specification applies to single-ended circuits, and the RS422 standard specifies balanced circuits. A companion to RS422, the RS423 standard applies to unbalanced circuits that work with RS422 balanced circuits.

If you aren't an electrical engineer or a communications fanatic, you won't know the difference and don't need to—as long as you can find out which wires go to which pins.

> ***Note: The wiring diagrams below are intended for use by someone who is familiar with data communications and has wired such cables before.***

If you don't already know how to make an RS232 cable, almost any computer repair technician can make one for you. It doesn't cost much, and you will feel much safer when you hook everything up and turn on the power.

Table 14.1 describes the Macintosh communications port.

If you are familiar with the RS232 interface, you know right away that something is wrong here. There are only 9 pins, instead of 25. The Macintosh does not use the 25-pin RS232 connector, nor does it conform to the RS449-connector layout (a connector usually used with RS422). You can, however, wire the Macintosh communications port to an RS232 device (table 14.2).

Table 14.1 Communications Port Pin Assignments

Pin	*Signal*
1	Ground
2	+5v
3	Ground
4	Transmit data +
5	Transmit data −
6	Filtered +12v
7	Handshake or external clock
8	Receive data +
9	Receive data −

Table 14.2 Communications Port RS232 Wiring

Macintosh	*DTE*	*DCE*
1	1	1
2	N/C	N/C
3	7	7
4	N/C	N/C
5	3	2
6	N/C	20
7	20	6
8	N/C	N/C
9	2	3

If you are wiring the Macintosh to a DTE (anything except a modem), wire the pins in the 9-pin connector that you plug into the Macintosh to the pins in the 25-pin connector that you connect to the other device, using the pin numbers in the DTE column. If you are connecting the Macintosh to a DCE (modem), use the pin numbers in the DCE column for the 25-pin RS232 connector.

What if you want to connect a Macintosh to another Macintosh, not an RS232 device? If you are using the telephone to communicate over a distance, each Mac should be connected to its own modem with a cable wired for RS232. If the two Macs are close together (in the same room), you can make up a cable to connect them without modems (table 14.3).

The Mac-to-Mac cable is symmetrical; it doesn't make any difference which end goes to which Mac.

Pin 7 in the Macintosh 9-pin connector goes to both the CTS (clear-to-send) and external-clock inputs to the 8530 chip. It can be used as either a handshake line or as the external-clock signal.

One more thing about the communications controller: in their passion for economy, the Apple engineers used an external input to the 8530 communications chip for the mouse-position signals. These signals don't appear at the 9-pin communications port connector. They are wired from the mouse connector to the 8530.

THE SERIAL COMMUNICATIONS DRIVER

The serial communications driver is Macintosh device-driver software for controlling the serial communications ports in the 8530 chip. The 8530 can do full duplex communications on both communications ports. *Full*

Table 14.3 Mac-to-Mac Null Modem Cable

First Mac	*Second Mac*
1	1
3	3
4	8
5	9
7	7
8	4
9	5

duplex means that the Macintosh can both transmit and receive through a communications port at the same time. Even though we have two communications ports, we are dealing with one device, the 8530 controller, so the two ports are related.

The Macintosh serial-communications-driver software does asynchronous communications. Currently, it does not support synchronous communications.

There are four communications drivers: a transmit driver for port A, a receive driver for port A, and the transmit and receive drivers for port B. There are two sets of communications drivers, the ROM drivers and the RAM drivers; the latter have a few more capabilities.

A program that wants to use the communications ports must open the drivers before doing any communications. With early versions of the Macintosh system software, closing the serial ports caused system problems, so many programs just leave them open all the time.

You can use the serial communications driver from application programs by calling the device manager to do reads and writes and by calling serial-communications-driver routines for control functions.

Device-Manager Calls for the Serial Driver

Write	Write data
Read	Read data
KillIO	Abort outstanding I/O requests and flush buffers

Serial Driver Routines

SerReset	Reset the driver and set new options
SerSetBuf	Specify an input buffer
SerHShake	Set handshake options
SerSetBrk	Set break mode
SerClrBrk	Clear break mode
SerGetBuf	Get the number of input bytes in the buffer
SerStatus	Get error, handshake, and I/O request flags

When calling SerReset, you supply the new options: baud rate; number of stop bits (1, 1½, or 2); parity (even, odd, or no parity); and the number of data bits (1 to 8). The serial driver baud rates are 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, 19200, and 57600.

The SerStatus routine returns 6 bytes with status and error information (table 14.4).

The software overrun flag in table 14.5 indicates that the buffer in the driver was full and the application program did not read the data from it

Table 14.4 Serial-Port Status Bytes

Field	*Bytes*	*Use*
cumErrs	1	Cumulative error flags
xOffSent	1	Flag, true if the driver sent an XOff
rdPend	1	Flag, true if there is a pending read request
wrPend	1	Flag, true if there is a pending write request
ctsHold	1	Flag, true if flow controlled by CTS
xOffHold	1	Flag, true if flow controlled by XOff

Table 14.5 Serial-Port Error Flags

Bit	*Error*	
0	swOverrunErr	Software overrun when receiving
4	parityErr	Parity error when receiving
5	hwOverrunErr	Hardware overrun when receiving
6	framingErr	Framing error (character received without stop bits)

before another character arrived from the 8530 communications controller. A hardware overrun is an 8530 error condition. It indicates that the 8530 three-character receive buffer was full and that another character arrived on the communications line before the driver read the data from the receive buffer. Both error conditions indicate that at least one data character was lost.

APPLETALK

AppleTalk is Apple's trademark for its local area network. The Macintosh connects to the AppleTalk network with one of its serial communications ports. The network consists of nodes up to 1,000 feet apart, connected by a 78-ohm, twisted-pair cable. A node consists of a device (like the Macintosh) that connects to the cable via a small transformer.

You don't need to have any configuration information when you connect a device to the network; you just plug it in. In most local area networks, you must add or remove cable terminators when you add or remove devices or extend the network. The terminations eliminate signal

reflections and distortion at network nodes and cable ends. The AppleTalk cable has terminations at its end points, and each node has termination resistors that automatically connect across the circuit when you unplug a device.

In most networks, you also need to know what addresses are assigned to each device. You must set switches in a new device to set its address before adding the device to the network. This procedure is automatic in the AppleTalk network. When you add a device to the network, the device first picks a random number for its address and then queries the network to see if the address is being used by another device. If the address is already taken, the new device repeats the procedure until it finds a free address. Once a device has established its address, it stores the address in a battery-powered CMOS memory chip that does not lose data when the power is turned off.

The network uses a bit-oriented synchronous communications protocol similar to High-Level Data Link Control (HDLC). The network is not limited to communicating with Apple computers, the Apple LaserWriter, and other devices. Apple supports connection to IBM personal computers with AppleTalk. Apple also has a cluster controller that emulates an IBM 3270 cluster controller. The cluster controller sits on the network, and any Macintosh on the network can establish a logical connection to it and be connected (as a 3270 terminal) to an IBM mainframe computer.

You can expect to see other equipment on the AppleTalk network in the near future: file servers with large storage capacities and bridges to other types of networks.

APPENDIXES

A Pascal Language Comparison

B Microsoft BASIC Commands and Reserved Words

C Macintosh BASIC Commands and Reserved Words

D Macintosh Compilers

E Toolbox Routines

APPENDIX A: PASCAL LANGUAGE COMPARISON

Reserved Words	*Macintosh*	*UCSD*	*Apple II*
and	✓	✓	✓
array	✓	✓	✓
begin	✓	✓	✓
case	✓	✓	✓
const	✓	✓	✓
div	✓	✓	✓
do	✓	✓	✓
downto	✓	✓	✓
else	✓	✓	✓
end	✓	✓	✓
external		✓	✓
file	✓	✓	✓
for	✓	✓	✓
forward		✓	✓
function	✓	✓	✓
goto	✓	✓	✓
if	✓	✓	✓
implementation		✓	✓
in	✓	✓	✓
interface		✓	✓
label	✓	✓	✓
mod	✓	✓	✓
nil	✓		✓
not	✓	✓	✓
of	✓	✓	✓
or	✓	✓	✓
otherwise	✓		
packed	✓	✓	✓
procedure	✓	✓	✓
process		✓	
program	✓	✓	✓
record	✓	✓	✓
repeat	✓	✓	✓
segment		✓	✓
separate		✓	
set	✓	✓	✓
string	✓		
then	✓	✓	✓

Reserved Words	*Macintosh*	*UCSD*	*Apple II*
to	✓	✓	✓
type	✓	✓	✓
unit		✓	✓
until	✓	✓	✓
uses	✓	✓	✓
var	✓	✓	✓
while	✓	✓	✓
with	✓	✓	✓

Data Types	*Macintosh*	*UCSD*	*Apple II*
array	✓	✓	✓
boolean	✓	✓	✓
char	✓	✓	✓
computational	✓		
double	✓		
enumerated	✓		
extended	✓		
file	✓	✓	✓
integer	✓	✓	✓
longint	✓	*	*
pointer	✓	✓	✓
processid		✓	
real	✓	✓	✓
record	✓	✓	✓
scalar		✓	✓
semaphore		✓	
set	✓	✓	✓
string	✓	✓	✓

Identifier Size and Significant Characters	*Macintosh*	*UCSD*	*Apple II*
maximum size	255	any	any
number of significant characters	all	8	all
case significant	yes	no	yes
underscore significant	yes	no	yes

Predeclared Identifiers	*Macintosh*	*UCSD*	*Apple II*
abs (real function)	✓	✓	✓
arctan (real function)	✓	✓	✓
atan (real function)		✓	✓
attach (procedure)		✓	

*Specify the integer size in decimal digits.

Predeclared Identifiers	*Macintosh*	*UCSD*	*Apple II*
blockread (integer function)		✓	✓
blockwrite (integer function)		✓	✓
boolean (data type)	✓	✓	✓
button (integer function)			✓
char (data type)	✓	✓	✓
chartype (turtle graphics procedure)		✓	
chr (character function)	✓	✓	✓
close (procedure)	✓	✓	✓
computational (data type)	✓		
concat (string function)	✓	✓	✓
copy (string function)	✓	✓	✓
cos (real function)	✓	✓	✓
creation	✓		
delete (procedure)	✓	✓	✓
dispose (procedure)	✓	✓	
double (data type)	✓		
drawblock (turtle graphics procedure)		✓	
eof (Boolean function)	✓	✓	✓
eoln (Boolean function)	✓	✓	✓
exit (procedure)		✓	✓
exp (procedure)	✓	✓	✓
extended (data type)	✓		
false (constant)	✓	✓	✓
filepos (function)	✓		
fillchar (procedure)		✓	✓
fillscreen (turtle graphics procedure)		✓	
get (procedure)	✓	✓	✓
gotoxy (procedure)		✓	✓
grafmode (turtle graphics procedure)		✓	
halt (procedure)		✓	✓
implementation	✓		
include (function)	✓		
initturtle (turtle graphics procedure)		✓	
input (file)		✓	✓
insert (procedure)	✓	✓	✓
integer (data type)	✓	✓	✓

Predeclared Identifiers	*Macintosh*	*UCSD*	*Apple II*
interface	✓		
interactive (file type)		✓	✓
intrinsic	✓		
ioresult (integer function)		✓	✓
keyboard (file)		✓	✓
keypress (Boolean function)			✓
length (integer function)	✓	✓	✓
ln (real function)	✓	✓	✓
log (real function)		✓	✓
longint (data type)	✓		
mark (procedure)		✓	✓
maxint (constant)	✓	✓	✓
memavail (integer function)		✓	✓
memlock (procedure)		✓	
memswap (procedure)		✓	
move (turtle graphics procedure)		✓	
moveleft (procedure)		✓	✓
moveright (procedure)		✓	✓
moveto (turtle graphics procedure)		✓	
new (procedure)	✓	✓	✓
newfilename (function)	✓		
nil (constant)	✓	✓	
note (procedure)	✓		✓
odd (Boolean function)	✓	✓	✓
oldfilename (string function)	✓		
omit (string function)	✓		
open (procedure)	✓		
ord (integer function)	✓	✓	✓
ord4 (longint function)	✓		
output (file)		✓	✓
pack (procedure)	✓		
paddle (integer function)			✓
page (procedure)	✓	✓	✓
pencolor (turtle graphics procedure)		✓	
pointer (pointer function)	✓		
pos (integer function)	✓	✓	✓
pred (scalar function)	✓	✓	✓
processid (data type)		✓	
put (procedure)	✓	✓	✓

Predeclared Identifiers	*Macintosh*	*UCSD*	*Apple II*
pwroften (real function)		✓	✓
random (integer function)	✓		✓
randomize (procedure)			✓
read (procedure)	✓	✓	✓
readln (procedure)	✓	✓	✓
real (data type)	✓	✓	✓
release (procedure)		✓	✓
reset (procedure)	✓	✓	✓
rewrite (procedure)	✓	✓	✓
round (integer function)	✓	✓	✓
scan (integer function)		✓	✓
screenbit (turtle graphics Boolean function)		✓	
screencolor (turtle graphics type)		✓	
seek (procedure)	✓	✓	✓
semaphore (data type)		✓	
seminit (procedure)		✓	
setchain (procedure)			✓
setcval (procedure)			✓
signal (procedure)		✓	
sin (real function)	✓	✓	✓
sizeof (integer function)	✓	✓	✓
sqr (real function)	✓	✓	✓
sqrt (real function)	✓	✓	✓
start (procedure)		✓	
str (string function)		✓	✓
string (data type)	✓	✓	✓
succ (scalar function)	✓	✓	✓
swapoff (procedure)			✓
swapon (procedure)			✓
text (file type)	✓	✓	✓
textmode (turtle graphics procedure)		✓	
time (procedure)		✓	
treesearch (integer function)		✓	✓
true (constant)	✓	✓	✓
trunc (integer function)	✓	✓	✓
ttlout (procedure)			✓
turn (turtle graphics procedure)			✓
turnto (turtle graphics procedure)		✓	

Predeclared Identifiers	*Macintosh*	*UCSD*	*Apple II*
turtleang (turtle graphics procedure)		✓	
turtlex (turtle graphics integer function)		✓	
unitbusy (Boolean function)		✓	✓
unitclear (procedure)		✓	✓
unitread (procedure)		✓	✓
unitstatus (procedure)		✓	
unitwait (procedure)		✓	✓
unitwrite (procedure)		✓	✓
unpack (procedure)	✓		
varavail (integer function)		✓	
vardispose (procedure)		✓	
varnew (integer function)		✓	
viewport (turtle graphics procedure)		✓	
wait (procedure)		✓	
wchar (turtle graphics procedure)		✓	
write (procedure)	✓	✓	✓
writeln (procedure)	✓	✓	✓
wstring (turtle graphics procedure)		✓	

APPENDIX B: MICROSOFT BASIC COMMANDS AND RESERVED WORDS

Arithmetic Functions

ABS	Absolute value
ATN	Arctangent
COS	Trigonometric cosine function
EXP	Base *e* exponent
LOG	Base *e* logarithm
RANDOMIZE	Randomize the random number generator
RND	Generate a random number
SGN	Get the sign of a variable or expression
SIN	Trigonometric sine
SQR	Square root
TAN	Trigonometric tangent

Array Operations

LBOUND	Get the lower bound of an array's dimension
UBOUND	Get the upper bound of an array's dimension
OPTION BASE	Set starting value for all array subscripts (0 or 1)

Conversion

ASC	Convert character to ASCII numerical value
CDBL	Convert to double precision
CHR$	Convert ASCII numerical value to character
CINT	Convert to integer
CSNG	Convert to single precision
CVI	Convert string to integer
CVS	Convert string to single-precision number
CVD	Convert string to double precision
CVSBCD	Convert decimal number to binary
FIX	Truncate to integer value
HEX$	Convert to hexadecimal string (no leading zeros!)
INT	Convert to integer
MKI$	Convert integer to string
MKS$	Convert single-precision number to string
MKD$	Convert double-precision number to string
MKSBCD$	Convert single-precision number to decimal string
MKDBCD$	Convert double-precision number to decimal string
OCT$	Convert to octal string

Conversion

STR$	Convert as number to a string
UCASE$	Convert a string to uppercase
VAL	Convert a string to a number

Definitions

COMMON	Define variables to be passed to another program via chain
DATA	Store data for the READ statement
DEF FN	Define a function
DEFINT	Declare variables integers
DEFSNG	Declare variables single-precision
DEFDBL	Declare variables double-precision
DEFSTR	Declare variables strings
DIM	Dimension arrays
SHARED	Declare shared variables
SUB	Define a subprogram

Dialog Box

BUTTON [CLOSE]	Control a button in a dialog box or window
DIALOG ON,OFF, STOP	Control dialog box
ON DIALOG	Go to a subroutine on a dialog event

Error Processing

ERR	Get the error code
ERL	Get the error line number
ERROR	Simulate error or define new error code
RESUME	Resume the program after error recovery routine

Event

BREAK [ON, OFF, STOP]	Break event
ON BREAK	Go to a subroutine on a break event
ON DIALOG	Go to a subroutine on a dialog event
ON ERROR GOTO	Go to a program line on an error
ON MENU	Go to a subroutine on a menu event
ON MOUSE	Go to a subroutine on a mouse event
ON TIMER	Go to a subroutine on a timer event
TIMER [ON, OFF, STOP]	Control timer events

Files

CLEAR	Close and reset practically everything
CLOSE	Close a file
EOF	Check for end of file
FIELD	Define a field in a file record and assign a string variable to it
FILES	List all files or test for a file's existence
FILES$	List all files or test for a file's existence
GET	Read a disk file record
INPUT$	Input a string from the keyboard or a file
INPUT#	Read from a sequential file
KILL	Delete a file from the disk directory
LINE INPUT#	Read a line from a sequential file
LOC	Locate a position in a file
LOF	Get a file's length
NAME	Rename a disk file
LSET	Move data to a random file's buffer
RSET	Move data to a random file's buffer
OPEN	Open a file
PRINT#	Output data to a sequential file
PRINT# USING	Formatted output to a sequential file
PUT	Put a record to a file or an image to the screen
RESET	Close all files
RSET	Move a string, right justified, to a random file's buffer
WRITE#	Write data to a sequential file

Graphics

CIRCLE	Draw a circle
LINE	Draw a line
PICTURE [ON, OFF]	Draw a picture or control picture recording
PICTURE$	Get the string of commands that recorded a picture
POINT	Read the value of a pixel
PRESET	Set a pixel in the output window
PSET	Set a pixel in the output window
PUT	Put a record to a file or an image to the screen

Keyboard I/O

INKEY$	Read a keyboard character
INPUT	Prompt user and input from the keyboard
INPUT$	Input a string from the keyboard or a file
LINE INPUT	Read a line of characters from the keyboard

Menu

MENU [ON, OFF, STOP]	Create and control menus
ON MENU	Go to a subroutine on a menu event

Miscellaneous

ERASE	Erase arrays from memory (to free their memory space)
DATE$	Get or set the current date
FRE	Get amount of free heap (memory) space
LET	Variable assignment statement
PEEK	Get the byte value in a memory location
POKE	Set the byte value of a memory location
REM	Remark
READ	Read values from data statements
RESTORE	Restore reading of data statements to the beginning or a line
SWAP	Swap the values to two variables
TIME$	Get or set the time of day
VARPTR	Get the memory address of a variable

Mouse

MOUSE [ON, OFF, STOP]	Read/control mouse
ON MOUSE	Go to a subroutine on a mouse event

Printing on the Imagewriter

LCOPY	Screen print
LLIST	List program on the printer
LPOS	Get the output position in the print buffer
LPRINT	Print on the line printer
LPRINT USING	Formatted print on the line printer
SPC	Generate spaces in PRINT
TAB	Tab in a PRINT
WIDTH	Set or get print line width

Printing on the Screen (in the output window, no font control)

CSRLIN	Get the line number of the pen in the output windov
LOCATE	Set the pen location in the output window
POS	Get the horizontal position of the pen
PRINT	Output data to the screen (in the output window)

Printing on the Screen (in the output window, no font control)

PRINT USING	Formatted data output to the screen
PTAB	Tab to a pixel position in the output window
WRITE	Write data to the current output window

Program Control

CONT	Continue after a break (keyboard command)
DELETE	Delete program lines (keyboard command)
END	End the program and close all files
LOAD	Load a program from disk
MERGE	Load an additional program from disk into memory
LIST	List program to the List window, to a file, or to a device
NEW	Set BASIC for a new program
RUN	Run the program currently loaded (keyboard command)
SAVE	Save the program in a disk file
STOP	Stop the program without closing files (program statement)
SYSTEM	Close files and exit to the system (Finder)
TRON	Tracing on
TROFF	Tracing off

Program Flow

CALL	Call a subroutine (BASIC or assembler)
CHAIN	Chain to (execute) another BASIC program
FOR . . . NEXT	FOR-NEXT loop
GOSUB . . . RETURN	Execute a subroutine
GOTO	Go to a program line
IF . . . THEN . . . ELSE	Conditional program flow control
IF . . . GOTO	Conditional GOTO
NEXT	See FOR . . . NEXT
ON . . . GOSUB	Multiway GOSUB (subroutine call)
ON . . . GOTO	Multiway GOTO (branch)
RETURN	Return from subroutine to the statement following the GOSUB
END SUB	End a subprogram
EXIT SUB	Exit a subprogram
WHILE . . . WEND	Program flow control

String Operations

INSTR	String search
LEFT$	Get the leftmost elements of a string
LEN	Get the length of a string
MID$	Get part of the middle of a string
RIGHT$	Get the rightmost characters of a string
SPACE$	Generate a string of spaces
STRING$	Generate a string, all of the same character

Sound

BEEP	Beep the speaker
SOUND	Make sounds with the speaker
WAVE	Control multivoice sound generation

Window

BUTTON [CLOSE]	Control a button in a dialog box or window
CLS	Clear a window
EDIT FIELD	Define an edit area in a window
EDIT$	Get the text in an edit area in a window
SCROLL	Scroll a window
WINDOW	Create or control a window

APPENDIX C: MACINTOSH BASIC COMMANDS AND RESERVED WORDS

Arithmetic Functions

ABS	Absolute value function
ANNUITY	Annuity function
ATN	Arctangent function
CLASSCOMP	Return the number class of a COMP number
CLASSDOUBLE	Return the number class of a DOUBLE number
CLASSEXTENDED	Return the number class of an EXTENDED number
CLASSSINGLE	Return the number class of a SINGLE number
COMPOUND	Compound interest function
COPYSIGN	Sign conversion function
COS	Cosine function
EXP	Base *e* exponent function
EXP2	Base 2 exponent function
EXPM1	Base *e* exponent minus 1 function
LOG	Base *e* logarithm function
LOG2	Base 2 logarithm function
LOGB	Function: Logb = largest *n*, 2**n < argument
LOGP1	Function: log(argument + 1)
NEXTDOUBLE	Next double-precision number function
NEXTEXTENDED	Next extended-precision number function
NEXTSINGLE	Next single-precision number function
OPTION COLLATE NATIVE	Set string comparison method to use alphabetical order
OPTION COLLATE STANDARD	Set string comparison method to use ASCII numeric value
PI	The extended precision value of π
RANDOMIZE	Initialize the random number generator routine with a new seed
RANDOMX	Psuedo random number function
REMAINDER	Remainder function
RINT	Round to integer function
RND	Random number function
SCALB	Function: argument2 * (2**argument1)
SGN	Sign function: −1, +1, or 0
SIGNNUM	Sign function: = 1 if sign(argument) negative, else = 0

Arithmetic Functions

SIN	Sine function
SQR	Square function: = argument**2
TAN	Tangent function
TRUNC	Truncate to integer function

Conversion

ASC	Convert a character to its numeric ASCII code
CHR$	Convert a decimal ASCII value to a character
DOWNSHIFT$	Convert a string to lowercase character
INT	Convert to integer
STR$	Convert a number to a string
UPSHIFT$	Convert a string to uppercase
VAL	Convert a string to a number

Definitions

DEF	Define a single line function
DIM	Set the dimensions of an array
END FUNCTION	End a multiple line function definition
EXIT FUNCTION	Exit a function before reaching the end
EXIT PROGRAM	Stop execution of the program
FUNCTION	Begin a multiple line function definition
UNDIM	Deallocate an array

Event Handling

IGNORE WHEN	Turn off event handling
WHEN CONTROL	Defines control event routine
WHEN ERR	Defines error event routine
WHEN KBD	Defines keyboard event routine
WHEN MENU	Defines menu event routine
WHEN WINDOW	Defines window event routine

Graphics

CLEARWINDOW	Clear the output window
ERASE OVAL	Erase an oval in the output window
ERASE RECT	Erase a rectangle in the output window
ERASE ROUNDRECT	Erase a round rectangle in the output window
FRAME OVAL	Draw an oval in the output window
FRAME RECT	Draw a rectangle in the output window
FRAME ROUNDRECT	Draw a round rectangle in the output window

Graphics

GPRINT	Print in the output window, using the current type font and size
GTEXTMODE	Set the mode for drawing characters in the output window with GPRINT
INVERT OVAL	Reverse the pixel values of an oval in the output window
INVERT RECT	Reverse the pixel values of a rectangle in the output window
INVERT ROUNDRECT	Reverse the pixel values of a round rectangle in the output window
PAINT OVAL	Fill an oval in the output window with the current pattern
PAINT RECT	Fill a rectangle in the output window with the current pattern
PAINT ROUNDRECT	Fill a round rectangle in the output window with the current pattern
PLOT	Plot a point in the output window

Input/Output

DATA	Define data for a READ statement
FORMAT$	Set the format for PRINT output
INKEY$	Return next character in the keyboard buffer
INPUT	Read data from the keyboard
KBD	Return the decimal ASCII value of the last keyboard key pressed
LINE INPUT	Read data from the keyboard a line at a time
PRINT	Output data to the display
READ	Read data from a DATA statement
RESTORE	Move read pointer back to the start of a DATA statement
TAB	Tab to the right in PRINT output
TABWIDTH	Set the width of tab columns

Interpreter Options

ASK ENVIRONMENT	Get the value of the ENVIRONMENT option PROCENTRY (see SET ENVIRONMENT)
ASK EXCEPTION	Get the value of the EXCEPTION option (see SET EXCEPTION)
ASK HALT	Get the value of the HALT option (see SET HALT)
ASK HPOS	Get the value of the HPOS option (see SET HPOS)
ASK FONT	Get the value of the FONT option (see SET FONT)
ASK FONTSIZE	Get the value of the FONTSIZE option (see SET FONTSIZE)

Interpreter Options

ASK PATTERN	Get the value of the PATTERN option (see SET PATTERN)
ASK PENMODE	Get the value of the PENMODE option (see SET PENMODE)
ASK PENSIZE	Get the value of the PENSIZE option (see SET PENSIZE)
ASK PENPOS	Get the value of the PENPOS option (see SET PENPOS)
ASK PRECISION	Get the value of the PRECISION option (see SET PRECISION)
ASK ROUND	Get the value of the ROUND option (see SET ROUND)
ASK SHOWDIGITS	Get the value of the SHOWDIGITS option (see SET SHOWDIGITS)
ASK VPOS	Get the value of the VPOS option (see SET VPOS)
PROCENTRY	Save the current numeric environment
PROCEXIT	Restore the numeric environment saved by PROCENTRY
SET ENVIRONMENT	Set the numeric environment (Exception, Halt, Precision, and Round options)
SET EXCEPTION	Enable floating-point exception events
SET HALT	Enable halt on floating-point exception
SET HPOS	Set the horizontal position of the mouse cursor
SET FONT	Set the font for printing in the output window with GPRINT
SET FONTSIZE	Set the font size for printing in the output window with GPRINT
SET PATTERN	Set the pattern for filling shapes in the output window
SET PENMODE	Set the pen mode for printing graphics in the output window with GPRINT
SET PENSIZE	Set the size of the graphics pen for drawing graphics and printing in the output window with GPRINT
SET PENPOS	Set the position of the graphics pen for drawing graphics and printing in the output window with GPRINT
SET PRECISION	Set the rounding precision for numeric calculations
SET ROUND	Set rounding direction
SET SHOWDIGITS	Set the value of the SHOWDIGITS option, the number of digits displayed by a PRINT statement
SET VPOS	Set the vertical position of the mouse cursor

Miscellaneous

FREE	Return the amount of free memory
LET	Assign a value to a variable or an array element
RELATION	Ordered relation function
REM	Remark (comment line)
TICKCOUNT	Return the value of the system variable TickCount, the count of 1/60-second clock ticks since the system was started (powered on or reset)
TIME$	Return the time of day as a string

Mouse

BTNWAIT	Wait until the mouse button is pressed with the cursor in the output window
MOUSEB	Return the state of the mouse button
MOUSEB~	Return the Boolean value of the mouse-button state
MOUSEH	Return the horizontal position of the mouse cursor
MOUSEV	Return the vertical position of the mouse cursor

Program Flow

CALL	Call a subroutine
CASE	Target of a multiway branch
DO-LOOP	Do loop
END	Stop execution of the program; defines the end of the program
ENDSELECT	End a SELECT statement (multiway branch)
FOR-NEXT	FOR-NEXT loop
GOSUB	Go to a subroutine
GOTO	Execute the specified program statement next
IF-THEN-ELSE-ENDIF	Conditional execution of included statements
PERFORM	Execute a subprogram (and return)
POP	Set the next RETURN to return to one level higher
RETURN	Return from a subroutine
SELECT	Multiway branch, used with CASE
STOP	Stop the program and execute the debugger

Sound

SOUND	Sound a tone on the speaker
SOUNDOVER	See if the sound generator buffer is empty
STOPSOUND	Stop the sound generator and flush its buffer
TONES	Diatonic note frequency function

String and Character Commands

ASC	Convert a character to its numeric ASCII code
CHR$	Convert a decimal ASCII value to a character
DOWNSHIFT$	Convert a string to lowercase
LEFT$	Substring function
LEN	String length function
MID$	Substring function
RIGHT$	Substring function
STR$	Convert a number to a string
UPSHIFT$	Convert a string to uppercase
VAL	Convert a string to a number

APPENDIX D: MACINTOSH COMPILERS

absoft Corporation 4268 North Woodward Avenue Royal Oak, Mich. 48072 (313) 549–7111	Fortran Compiler
Apple Computer, Inc. 20525 Mariani Avenue Cupertino, Calif. 95014 (408) 996–1010	Macintosh BASIC Macintosh 68000 Development System Macintosh Pascal
Consulair Corporation 140 Campo Drive Portola Valley, Calif. 94025 (415) 851–3849	C Compiler
Creative Solutions Inc. 4801 Randolf Road Rockville, Md. 20852 (301) 984–0262	MacForth
Hippopotamus Software 1250 Oakmead Parkway, Suite 210 Sunnyvale, Calif. 94086 (408) 730–2601	C Compiler
Mainstay 28611B Canwood Street Agoura Hills, Calif. 91301 (818) 991–6540	Assembler
MANX Software Systems P.O. Box 55 Shrewsbury, N.J. 07701 (201) 780–4004	C Compiler
MegaMax, Inc. P.O. Box 6015 Waco, Tex. 76706 (214) 987–4931	C Compiler

Microsoft Box 97200 10700 Northrup Way Bellevue, Wash. 98004 (206) 828–8080	Microsoft BASIC
Modula Corporation 1673 West 820 North Provo, Utah 84601 (801) 375–7400	Modula 2 Compiler
Softech Microsystems 16875 West Bernardo Drive San Diego, Calif. 92127 (619) 451–1230	Pascal Compiler Fortran Compiler
Softworks, Ltd. 607 West Wellington Chicago, Ill. 60657 (312) 327–7666	C Compiler
TMQ Software, Inc. 1110 Lake Cook Road Buffalo Grove, Ill. 60090 (312) 520–4440	TMON Debugger with Interactive Assembler

APPENDIX E: TOOLBOX ROUTINES

Event Manager

Event Record:

```
TYPE EventRecord = RECORD
    what:          INTEGER;
    message:       LongInt;
    when:          LongInt;
    where:         Point;
    modifiers:     INTEGER
  END;
```

Button;

Trap:	$AD74	
Type:	Function	Boolean
Parameters:	none	
Description:	Get the current state of the mouse button (TRUE = button down).	

EventAvail(eventMask, theEvent);

Trap:	$AD71	
Type:	Function	Boolean
Parameters:	eventMask	Integer
	theEvent	EventRecord
Description:	Get the next event of any type specified by the event mask from the event queue but do not remove the event from the queue.	

FlushEvents(eventMask, stopMask);

Trap:	$A032	
Type:	Procedure	
Parameters:	eventMask	Integer
	stopMask	Integer
Description:	Discard events in the event queue of types in the event mask until encountering an event of a type specified by the stop mask.	

GetNextEvent(eventMask, theEvent);

Trap:	$AD70	
Type:	Function	Boolean
Parameters:	eventMask	Integer
	theEvent	EventRecord Variable

Description: Get the next event of any type specified by the event mask from the event queue, removing the event from the queue.

StillDown;

Trap: $AD73
Type: Function Boolean
Parameters: none
Description: See if the mouse button is still down—that is, see if there have been no MouseUp events since the last MouseDown event. If there is a MouseUp event in the event queue, leave the event in the queue and return FALSE.

WaitMouseUp;

Trap: $AD77
Type: Function Boolean
Parameters: none
Description: See if the mouse button is still down—that is, see if there have been no MouseUp events since the last MouseDown event. If there is a MouseUp event in the queue, return FALSE and remove the event from the queue.

Menu Manager

Menu Record:

```
TYPE MenuInfo = RECORD
    menuID:        INTEGER;
    menuWidth:     INTEGER;
    menuHeight:    INTEGER;
    menuProc:      Handle;
    enableFlags:   PACKED ARRAY [0..31] OF BOOLEAN;
    menuData:      Str255
  END;
```

AppendMenu(menu, itemData);

Trap: $AD33
Type: Procedure
Parameters: menu MenuHandle
itemData String
Description: Add an item to a menu.

CheckItem(menu, item);

Trap:	$AD45	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
Description:	Put a check mark beside a menu item.	

ClearMenuBar;

Trap:	$AD34	
Type:	Procedure	
Parameters:	none	
Description:	Clear all menus from the current menu bar's menu list.	

CountMItems(menu);

Trap:	$AD50	
Type:	Function	Integer
Parameters:	menu	MenuHandle
Description:	Get the number of items in the menu.	

DeleteMenu(menuID);

Trap:	$AD36	
Type:	Procedure	
Parameters:	menuID	Integer
Description:	Delete the menu from the current menu bar's menu list.	

DisposeMenu(menu);

Trap:	$AD32	
Type:	Procedure	
Parameters:	menu	MenuHandle
Description:	Release the menu's heap space allocated by NewMenu.	

DrawMenuBar;

Trap:	$AD37	
Type:	Procedure	
Parameters:	none	
Description:	Draw the menu bar on the screen, using the current menu bar's menu list.	

FlashMenuBar(menuID);

Trap:	$AD4C	
Type:	Procedure	
Parameters:	menuID	Integer

Description: Change the menu's title in the menu bar to display in reverse video (white on black).

GetItem(menu, item, itemString);

Trap:	$AD8D	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	itemString	String Variable
Description:	Get the menu item's text (not including metacharacters).	

GetItemIcon(menu, item, icon);

Trap:	$AD3F	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	icon	Integer Variable
Description:	Get the menu item's icon number.	

GetItemMark(menu, item, markChar);

Trap:	$AD43	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	markChar	CHAR Variable
Description:	Get the menu item's mark character.	

GetItemStyle(menu, item, chStyle);

Trap:	$AD41	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	chStyle	Style Variable
Description:	Get the menu item's text style (typeface).	

GetMenuBar;

Trap:	$AD3B	
Type:	Function	Handle
Parameters:	none	
Description:	Copy the current menu bar's menu list and return a handle to it.	

GetMHandle(menuID);

Trap:	$AD49	
Type:	Function	MenuHandle

Parameters:	menuID	Integer
Description:	Get a handle to the menu.	

InitMenus;

Trap:	$AD30
Type:	Procedure
Parameters:	none
Description:	Initialize system data structures and clear the current menu list.

InsertMenu(menu, beforeID);

Trap:	$AD35	
Type:	Procedure	
Parameters:	menu	MenuHandle
	before ID	Integer
Description:	Insert the menu pointed to by the menu handle in the current menu bar's menu list before the menu identified by beforeID.	

HiLiteMenu(menuID);

Trap:	$AD38	
Type:	Procedure	
Parameters:	menuID	Integer
Description:	Highlight the menu's title and return any previously highlighted menu title to normal display.	

MenuKey(ch);

Trap:	$AD3E	
Type:	Function	LongInt
Parameters:	ch	CHAR
Description:	Select the menu item associated with the command character.	

MenuSelect(startPt):

Trap:	$AD3D	
Type:	Function	LongInt
Parameters:	startPt	Point
Description:	Track the mouse cursor from the start point, pulling down menus and highlighting menu items. Return the selected menu and menu item in the long integer.	

NewMenu(menuID, menuTitle);

Trap:	$AD31	
Type:	Function	MenuHandle
Parameters:	menuID	Integer
	menuTitle	String
Description:	Allocate heap space for the menu and return a handle to it.	

SetItem(menu, item, itemString);

Trap:	$AD47	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	itemString	String
Description:	Replace the menu item's text with the string.	

SetItemIcon(menu, item, icon);

Trap:	$AD40	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	icon	Integer
Description:	Replace the menu item's icon with the new icon.	

SetItemMark(menu, item, markChar);

Trap:	$AD44	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	markChar	Char
Description:	Add the mark character to the menu item's text.	

SetItemStyle(menu, item, style);

Trap:	$AD47	
Type:	Procedure	
Parameters:	menu	MenuHandle
	item	Integer
	style	Style
Description:	Set the menu item's style.	

SetMenuBar(menuBar);

Trap:	$AD3C	
Type:	Procedure	
Parameters:	menuBar	Handle
Description:	Make the menu bar the current menu bar. The handle is a handle to the new menu bar's menu list.	

SetMenuFlash(menu, count);

Trap:	$AD4A	
Type:	Procedure	
Parameters:	menu	MenuHandle
	count	Integer
Description:	Set the selection flash period for all items in the menu.	

QuickDraw

DrawString(string);

Trap:	$AC84	
Type:	Procedure	
Parameters:	string	String
Description:	Draw the characters in the string, starting at the current pen position and using the current typeface, type font, and type size.	

EraseRect(r);

Trap:	$ACA3	
Type:	Procedure	
Parameters:	r	Rect
Description:	Erase the rectangle, including its interior (fill it with the current background pattern).	

GetPort(gp);

Trap:	$AC74	
Type:	Procedure	
Parameters:	gp	GrafPtr variable
Description:	Get a pointer to the current graph port's data structure.	

InitCursor;

Trap:	$AC50	
Type:	Procedure	
Parameters:	none	

Description: Initialize the cursor (set it to a visible arrow).

SetRect(r, left, top, right, bottom);

Trap:	$ACA7	
Type:	Procedure	
Parameters:	r	Rect
	left	Integer
	top	Integer
	right	Integer
	bottom	Integer
Description:	Set the rectangle data structure to the specified coordinates.	

SetPort(gp);

Trap:	$AC73	
Type:	Procedure	
Parameters:	gp	GraphPtr
Description:	Set the current graph port to the graph-port data structure specified by the pointer.	

ShowPen;

Trap:	$AC97	
Type:	Procedure	
Parameters:	none	
Description:	Enable drawing.	

TextFace(face);

Trap:	$AC88	
Type:	Procedure	
Parameters:	face	Style
Description:	Set the current graph port's text face.	

TextFont(font);

Trap:	$AC87	
Type:	Procedure	
Parameters:	font	Integer
Description:	Set the current graph port's text font.	

TextSize(size);

Trap:	$AC8A	
Type:	Procedure	
Parameters:	size	Integer
Description:	Set the current graph port's text size.	

Toolbox Utilities

BitAnd(long1, long2);

Trap:	$AC58	
Type:	Function	LongInt
Parameters:	long1	LongInt
	long2	LongInt
Description:	Return the logical AND of the two long integers (double words).	

HiWord(x);

Trap:	$AC6A	
Type:	Function	Integer
Parameters:	x	LongInt
Description:	Return the high-order word of the long integer (double word).	

LoWord(x);

Trap:	$AC6B	
Type:	Function	Integer
Parameters:	x	LongInt
Description:	Return the low-order word of the long integer (double word).	

Window Manager

Window Record:

```
TYPE WindowRecord = RECORD
    port:          GrafPort;
    windowKind:    INTEGER;
    visible:       BOOLEAN;
    hilited:       BOOLEAN;
    goAwayFlag:    BOOLEAN;
    spareFlag:     BOOLEAN;
    structRgn:     RgnHandle;
    contRgn:       RgnHandle;
    updateRgn:     RgnHandle;
    windowDefProc: Handle;
    dataHandle:    Handle;
    titleHandle:   Handle;
    titleWidth:    INTEGER;
    controlList:   Handle;
    nextWindow:    WindowPeek;
    windowPic:     PicHandle;
    refCon:        LongInt
END;
```

BeginUpdate(theWindow);

Trap:	$AD22	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Begin accumulating changes into the update region.	

CloseWindow(theWindow);

Trap:	$AD2D	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Remove the window from the screen, delete it from the window list, and dispose of any heap areas allocated to it.	

DragWindow(theWindow, startPt, boundsRect);

Trap:	$AD25	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
	startPt	Point
	boundsRect	Rect
Description:	Drag the window over the screen, following the mouse cursor. The start parameter has the position of the mouse cursor in the drag region of the window. The bounds rectangle defines the window's limits of movement.	

DrawGrowIcon(theWindow);

Trap:	$AD04	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Draw the size-box and scroll-box area outlines in the window.	

EndUpdate(theWindow);

Trap:	$AD23	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Stop accumulating changes in the update region.	

FindWindow(thePt, theWindow);

Trap:	$AD2C	
Type:	Function	Integer

Parameters:	thePt	Point
	theWindow	WindowPtr
Description:	Find out if the point (usually the mouse cursor position) is in the window and, if so, what region of the window.	

FrontWindow;

Trap:	$AD24	
Type:	Function	WindowPtr
Parameters:	none	
Description:	Get a pointer to the window record of the frontmost window.	

GrowWindow(theWindow, startPt, sizeRect);

Trap:	$AD2B	
Type:	Function	LongInt
Parameters:	theWindow	WindowPtr
	startPt	Point
	sizeRect	Rect
Description:	Draw a size-change outline that follows the mouse cursor as it moves. The start point is the starting mouse cursor position. The size rectangle defines the limits on the size of the window. GrowWindow returns the new size of the window.	

HideWindow (theWindow);

Trap:	$AD16	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Hide the window (make it invisible).	

InvalRect(badRect)

Trap:	$AD28	
Type:	Procedure	
Parameters:	badRect	Rect
Description:	Add the rectangle to the update region (usually used to add scroll-bar rectangles to the update region).	

NewWindow(wstorage, boundsRect, title, visible, procID, behind, goAwayFlag, refCon);

Trap:	$AD13	
Type:	Function	Pointer

Parameters:	wstorage	Pointer
	boundsRect	Rect
	title	String
	visible	Boolean
	procID	Integer
	behind	WindowPtr
	goAwayFlag	Boolean
	refCon	LongInt
Description:	Create a new window and add it to the window list. The storage parameter is a pointer to the storage area for the window record. The bounds rectangle is the size of the window. Title is the window title that will appear in the title bar. The visible flag determines whether the window will be visible or invisible. The procID is an integer that identifies the window definition procedure; that is, it identifies the window type. The behind parameter points to the record of the window that will be in front of the new window. The goAway flag determines whether or not the window has a close box. The refCon parameter has the initial value of the window record refCon field, a user data item.	

SelectWindow (theWindow);

Trap:	$AD1F	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Select the window (make it the active window).	

ShowWindow (theWindow);

Trap:	$AD15	
Type:	Procedure	
Parameters:	theWindow	WindowPtr
Description:	Make the window visible.	

SizeWindow (theWindow, w, h, fUpdate);

Trap:	$AD1D
Type:	Procedure

Parameters:	theWindow	WindowPtr
	w	Integer
	h	Integer
	fUpdate	Boolean
Description:	Change the window to a new size defined by the width and height parameters. If fUpdate is TRUE, accumulate new window areas in the update region.	

TrackGoAway(theWindow, thePoint);

Trap:	$AD1E	
Type:	Function	Boolean
Parameters:	theWindow	WindowPtr
	thePoint	Point
Description:	Track the mouse cursor after a mouse down event in the close box. If the mouse button is released in the close box, return TRUE.	

GLOSSARY

Activate Event An event that reports to the program that a window has just become the active window. An activate event could be the result of the front window's being closed (the next window becomes active) or the user's clicking the mouse in a window.

Active Window The frontmost window on the screen. The only window in which any interaction with the user may take place. There can be only one active window at any given time.

Alert Box A type of window in which the Macintosh displays an alert message when it needs to notify the user of an unusual event or error condition.

Allocation Block A group of *n* logical blocks, where *n* is fixed for a particular disk format. Disk space is allocated in units of allocation blocks. See Logical Block and Sector.

Apple Menu A menu that contains items for invoking the desk accessories. It is called the Apple menu because it has an Apple symbol in the menu bar instead of a title.

Button 1. The push button on top of the mouse. 2. A type of control in a window, most often in a dialog or alert box. A button is a rectangle with rounded corners. When you click the mouse inside the button, you have, in effect, pushed the button (selected the action represented by the button).

Click The act of placing the mouse cursor over an object on the screen and clicking the mouse button (pushing the button and rapidly releasing it). Clicking an object identifies it to the Finder. Usually, clicking selects the object. You then have a choice of several actions that the Finder can take with the selected object; for instance, it can open a document, duplicate a document, or eject a disk.

Clipboard A file in which Macintosh programs temporarily store data that has been cut or copied from a document.

Close Box A box in the upper part of a window's frame. When you click the mouse inside the close box, the window closes (goes away). The window-manager documentation sometimes calls the close box the *go-away region*.

CRT Cathode-ray tube. The screen on which the Macintosh displays text and graphics is the screen of a CRT.

Desk Accessory A program that you can invoke while you are using another application. The desk accessory's window overlays the application's window, and the application is dormant until you have finished with the desk accessory. The Scrapbook, Calculator, Control Panel, Alarm Clock, and Note Pad are desk accessories. You invoke a desk accessory by selecting it from the Apple menu.

Desk Top The working environment of the Macintosh, the user interface. The Macintosh displays the icons for all of its programs and documents on the screen as if the screen were a desk top. The user moves the icons around and manipulates them as if they were objects on a desk top.

Dialog Box A type of window used by the Finder and application programs to carry on a dialogue with the user. For instance, a dialog box may have a series of questions, a choice of actions, or a list of files from which to select.

Double-Click Clicking an object on the screen twice rapidly. Double-clicking has a special meaning to the Finder, depending on the object and the circumstances. For instance, double-clicking an application's icon causes the Finder to execute the application.

Drag To drag an object on the screen, you place the mouse cursor over the object, hold down the mouse button, and move the mouse. The mouse, cursor, and an outline of the object move as you move the mouse. You may use dragging to move an object to a different location on the screen or to copy an object.

Eject When you put a disk in a Macintosh disk drive, the drive locks the disk inside so that you cannot remove it. The drive locks the disk inside to prevent it from being removed before the Macintosh has completed writing data to it and updating its directory. To get your disk back, you must select the disk icon and choose Eject from the File menu.

Event An event is something that happens that would be of interest to a program. Examples are the user's pressing a keyboard key, the user's pressing the mouse button, a disk insertion, and moving a window so that it or another window needs to be updated (for example, uncovering part of a window).

Finder The Macintosh programs that provide the desktop environment—the user-interface programs for the Macintosh operating system.

Folder A type of disk subdirectory—not really a subdirectory, but a subset of the files in the main disk directory. It appears on the screen as a folder in which you store documents and applications.

Font A set of type with all of the characters having the same appearance. Some Macintosh fonts are New York, Geneva, Chicago, Monaco, and Venice.

Fork A part of a Macintosh file. Each Macintosh file has two parts, a data fork and a resource fork.

Handle A pointer to a pointer. Some data and all program-executable code is in dynamically allocated segments and may be moved during the course of program execution. The system maintains one pointer to the data or code segment, and all programs reference it with a pointer to that pointer (a handle).

Heap A portion of memory from which the system dynamically allocates segments of memory to application programs for local variables and executable code.

Icon A small picture on the screen that represents an object such as a file, disk, application program, or document.

Imagewriter The Macintosh dot matrix printer that prints graphics and the type fonts that are on the Mac.

Invert To change the way the Macintosh displays an object from black on white to white on black. A feature that is used to identify an object by highlighting it.

Logical Block The data portion of one disk sector, 512 bytes.

Menu The means by which you set an application program's options or parameters and select actions that it will take. A menu has a list of items from which you may choose.

Metacharacter A character used in a menu item definition to set attributes for the item. Metacharacters do not appear in the menu item, but they affect the appearance of the menu item.

Minifinder A type of dialog box that an application displays. It has a scrolling window in which it displays a list of files so the user may choose one.

Mouse A pointing device used by the Macintosh. When you move the mouse over the surface of a desk or table, the mouse notifies the Macintosh as to how much it has moved and in which direction, and the Macintosh moves the cursor on the screen.

Note Pad A desk accessory that stores eight very small pages of notes. You can use the Edit menu's Cut, Copy, and Paste functions with the Note Pad to paste part of a document into the Note Pad, to include a note in a document, or to exchange data between documents.

Operating System A collection of programs that manages a computer's resources and provides services to application programs.

Pointer A variable that holds a memory address. It holds the address of a data item or structure, instead of the item itself.

QuickDraw A set of routines for drawing pictures and text on the screen. Everything in the Macintosh that draws an image on the screen does so by calling QuickDraw routines.

RAM Random access memory. A type of memory that can be read or written. RAM is volatile memory; its contents are lost when the machine's power is turned off. The Macintosh has either 128K or 512K RAM and uses it to hold application programs it is executing, data, and some operating-system routines.

Region A QuickDraw data structure that describes an area of the screen with an arbitrary shape.

Resource A resource can be any number of things that an application needs: a segment of executable code, an icon, a menu template, or a window template. A programmer may invent new resources. All of an application's executable code is stored as resources in the resource fork of a file. Some application programs, particularly those still under development, have a separate file for the resources that they expect to change frequently.

ROM Read-only memory. A type of memory used to permanently store data and programs. The Macintosh has 64K ROM, which holds the user-interface toolbox routines.

SANE Standard Apple Numeric Environment. A collection of mathematical routines (floating-point, multiprecision, and such) that is used by Macintosh Pascal and Macintosh BASIC and will be the basis for arithmetic in other Apple compilers.

Scrapbook The Scrapbook is a desk accessory. It has a file in which the user can store data clipped from a document. The Scrapbook is used to hold frequently used text, pictures, formulas, or other data and to transfer such data from one application to another.

Scroll To change the area of a document that a window displays. The document appears to move under the window as if it were a scroll being wound from one spool to another.

Scroll Bar A rectangular area running along the right side of a window or along the bottom. The scroll bar is usually filled with a pattern and has a small white box somewhere in the bar. The white box is called the *thumb*, or scroll box. Dragging it causes the window to scroll the document in the direction of movement. Most windows with scroll bars also have arrows at either end of the scroll bar. Clicking the mouse in an arrow scrolls the document and moves the scroll box. The position of the scroll box in the scroll bar indicates where you are in the document. When the scroll box is at the top of a vertical scroll bar, the window displays the first part of the document; when it is at the bottom, the window displays the last part of the document.

Sector A fixed-size portion of a disk track. On Macintosh disks, a sector contains 524 bytes, 512 of which may be used for data (12 bytes are used by the system).

Segment A dynamically allocated portion of memory that is part of the heap.

Size Box A box in the lower right-hand corner of a window. Dragging the size box drags an outline of the window. When you release the size box, the window assumes the size of the outline.

Text-Only A type of document that contains only printable ASCII characters, carriage returns, line feeds, and tabs. A text-only document has no type font or style information.

Thumb Another name for the scroll box. See Scroll Bar.

Title Bar The upper part of a window frame where the title is displayed.

Toolbox A collection of subroutines that do the detailed work necessary for the user interface. There are toolbox routines for drawing on the screen, managing menus, managing windows, managing events, and many other things.

Track One of eighty concentric circles that are the recording areas on the floppy disk.

Trash When you want to delete a file (document or application), you move its icon to an icon on the desk top that represents a trash can. The Finder marks the document as being in the trash, and the

document doesn't show up in the directory window. You can recover the document from the trash unless you have selected Empty Trash from the Special menu or have run out of room on the disk. In either case, the Finder deletes the file's directory entry and uses the disk space for another file.

Typeface The particular style in which a type font is displayed (for example, boldface, italic, or outline).

Update Event An event that reports to the program that a region of a window needs to be updated. An update event could be the result of enlarging a window or the result of another window's being moved so as to uncover part of the update window.

Update Region A region data structure that identifies the area of a window that needs to be updated, usually as the result of an update event.

Volume A floppy disk. Each disk is a volume and should have a unique volume name. When you insert a disk into the Macintosh disk drive, the Mac displays a disk icon on the screen with the disk's volume name under the icon.

Window A rectangular area of the screen used by the Finder or an application program to display data. There are a number of predefined window types, and a programmer may define new ones. An application can display data on the screen only in a window.

INDEX

.D, 279, 296
.Link, 279, 280–282, 292–294
.Rel, 279, 280, 292–299
.Rsrc, 279, 281, 298, 299
3270 terminal, 316, 317, 325
6522 VIA, 232–237, 241–243
68000
  addressing, 251–255
  autovector process, 230, 257–259
  exception processing, 250, 257–260
  interrupts, 237, 250
  memory, 223–228
  registers, 248–249
  STOP & RESET, 250
  supervisor & user states, 249, 250
  tracing, 250, 251
8021 microprocessor, 237
8530
  error conditions, 323, 324
  interrupt priority, 230
  interrupt vectors, 231
  mouse position signals, 241, 243, 322
  ports, 320–323
  serial controller chip, 223

Activate window. *See also* Event
  close option in file menu, 78
  introduced, 33
  multiple windows, 180–182
  observe window, 63
  the mouse, 196
Address, device, 229, 230
Address registers
  68000 processor, 248–249, 251–255
  assembler, 291
Address, 72, 79. *See also* Memory
  68000 address bus, 248
  addressing, 251–255
  device address, 229–230
  dispatch table, 225, 226
  instruction set, 261–269
  I/O addressing, 256
  Mac address space, 223
  vector, 257–259
Alert box
  and dialog manager, 47
  and system error handler, 48
  for finder messages, 13
  resource type, 299
  use of the, 30–32
Aliasing, 37
Allocation block, 309–312
Alto, 3
Amplitude, 140
Annuity, 133
ANSI Pascal, compatibility with Macintosh Pascal, 67–70
AppendMenu
  defined, 162
  use, 157
Apple II Pascal
  turtle graphics, 59
  compared to Macintosh Pascal, 67–68
Apple symbol, 29, 33, 34, 152
AppleTalk, 324–325
Application
  and activate events, 199–201
  and data files, 41
  and event manager, 238
  and mouse events, 196
  and resource files, 42
  and windows, 180, 184–186, 188
  consistency, 28–30
  development guidelines, 4
  developing, 273–283
  debugging, 300–302
  font, 84, 134
  linking, 292–295
  MacAsm, 272–273
  MacDS, 272, 273
  memory, 226–228
  menu, 152–153
  prototypes, 6, 52, 123
  refCon, 188, 191
  resource compiler, 298–299
  suspending the, 33–34
  window activated, 165
Application global area, 227, 275
Application heap, 224–228, 232, 303
Array, Macintosh BASIC, 130. *See also* PAL
ASCII. *See* Characters
Assembler. *See also* MacDS
  assembly-link process, 273
  developing applications, 273
  MacAsm, 275, 276
  MacAsm and MacDS compared, 272–273
  MacDS, 276–283, 287–292
  symbol table, 295
Assignment compatible, 53
Asynchronous
  communications, 317–320, 323
  event, 134
Autovector, 230, 257–259

Background loop, 106–110, 142, 163, 178, 196
BASIC, 6. *See also* Macintosh BASIC & Microsoft BASIC
  data types compared to Pascal, 53
Baud rate, 320, 323
BCD (Binary Coded Decimal), 251
BeginUpDate, 201

BInLineF, *see* InLine
Block allocation, 309
Boldface. *See also* Font & TEXTFACE
  BASIC reserved words, 74, 82, 83
  Pascal reserved words, 56
  style, 40
Boolean
  Macintosh BASIC data type, 130, 132
  Pascal data type, 53
Boot button, *see* Programmer's switch
Bomb icon, 48
Branch instructions, 250, 261, 289
Button
  BUTTON statement, 104
  Cancel button, 31
  Execute button, 283
  in CALC program, 141
  in Windows & dialog boxes, 104–106
  OK button, 13, 86
  Open button, 299
  Pack button, 297
  types in Microsoft BASIC, 104, 105

C
  language, 6
  toolbox use, 49
Carry, condition code, 261–264
CASE, 134, 196
Check mark
  on menu item, 101–103, 153
  metacharacter, 158
  setting and removing, 176
Characters. *See also* Font
  Alto, 3
  ASCII, 102, 103, 238–241
  code, 164
  dimmed & undimmed, 30
  identifiers, 69–71
  keyboard command, 158
  Macintosh BASIC, 130
  meta, 157–159
  Pascal data type, 53
  printer, 244–245
  serial communications, 317–320
  special, 130, 154
  Sync, 317
CheckItem, 162, 176
Check option (run menu), 64
Circle, drawing, 83, 96
ClearMenuBar, 160, 162
Clear to Send, 322
Click, 10
Clipboard, 15–17, 47, 63
Clock
  6522, 233–236
  68000 processor, 222
  alarm, 47
  communications port, 317, 320–322
  ticks, 165
  time of day, 223, 229
Close
  file menu item, 18, 22, 78, 91, 283
  BASIC statement, 94
  WINDOW CLOSE, 103, 109
  window attribute, 182
Close All, 18
Close box, 32–35, 78, 79, 180, 196, 197
CloseWindow, 191
Code segment, 274, 292, 294, 295
Command character, 158
Command key
  to suspend program, 80
  using, 158, 163–167, 174–176, 239
Command window, 77, 78, 81, 82
Comments
  assembler, 289, 290
  in link control file, 292
  Macintosh BASIC, 122
  Microsoft BASIC, 75
Communications. *See* Serial communications & AppleTalk
Communications driver, 322, 323
Compiler, 49
  differences, 68
  Pascal, 52, 53, 56, 59
  resource, 275–277, 292, 298
  toolbox use, 98
COMPOUND, 133, 144
Compressed, Microsoft BASIC save format, 91
Compuserve, 316
Computational, Pascal data type, 68–71
Condition code, 249–250, 260–264
Constant
  assembler, 290
  Macintosh BASIC, 130
  Macintosh Pascal, 53–56
  memory bounds, 224
Content region
  drawing in, 193–194
  updating, 198, 199
  use, 183
Continue, 79, 82
Control Manager, 47, 188
Copy
  document, 10, 43
  clipboard, 15
  item in edit menu, 15
  text edit function, 47
CountMItems, 162
CP/M, 2
Creation time, 311
CRT. *See also* Video Display & Memory
  description, 228
  photographing, 222
  retrace, 232, 244
CTS. *See* Clear to Send
Cursor
  stop sign, 65
  using, 10, 15, 21, 152
  vertical retrace interrupt processing, 232
Cut
  edit menu item, 15
  Microsoft BASIC, 78
  text edit, 47

Data communications equipment, 317, 321
Data terminal equipment, 317, 322
Data rate, disk, 307. *See also* Baud rate
Data registers, 236, 248–251, 253, 291
Data type
  checking with InLine, 156
  conversion in Macintosh Pascal, 53
  event mask, 164
  InLine, 156
  Macintosh BASIC, 130
  Macintosh Pascal, 53, 56, 59, 69–71
  Microsoft BASIC, 75
  windows, for, 187, 188

DCE. *See* Data communications equipment
Deallocate
  menu, 155, 160
  window, 191
Debugging
  Macintosh BASIC, 122, 126, 129, 134
  Macintosh Pascal, 59, 63, 64
  Microsoft BASIC, 80–82, 89
  MacASM, 276
  MacDS, 276–277, 300–302
DeleteMenu, 162
Desk accessory
  note pad, 34, 35
  standard menu, 29
  use with edit menu, 280
  using, 33–35
Desk manager, 47
Desk top
  as a user interface, 29
  conceptual model, 43
  finder creates, 278
Device manager, 48, 323
Dialog
  find dialog box, 285
  for finder questions, 14
  manager, 47
  Microsoft BASIC, 91
  minifinder, 31
  packSyms, output dialog box, 297
  print dialog box, 106
  print format dialog box, 286
  resource type, 299
  save dialog box, 14, 21
  search dialog box, 124–125
  use of, 30–32
  user interface routine, 8
  window type, 104
Disable
  DisableItem, 162, 176
  disabled item, 158, 159
  menu item metacharacters, 158, 159
  using menu manager, 158
  with MENU statement, 101, 102
Disk. *See also* File
  controller, 10, 223, 207–309
  drive, 306–309
  driver, 48
  floppy, 273
  format, 309–313
  initialization, 308
  insertion event, 164, 232
  interface, 228
  sector, 307, 312–313
  track, 307
  volume, 309, 310
  winchester, 273
Dispatch table, 225–226
Displacement, 225, 226, 253, 262, 291, 301
DisposeMenu
  deallocate menu space, 155, 160
  defined, 162
Document,
  introduction, 10–22
  & windows, 32, 33, 40–43, 180, 183, 188, 191
  Macintosh BASIC, 123
  Macintosh Pascal, 60–62, 72
  Microsoft BASIC, 104
Double (data type), 68–71
Dow Jones News/Retrieval Service, 281

Drag
  defined, 10, 15, 21, 43
  region, 183
  windows, 183, 184
DragWindow, 196–198
DrawGrowIcon, 199
Drawing Window, 59, 62, 67
DrawMenuBar, 160, 162
DTE. *See* Data Terminal Equipment
Duration, sound, 140

Edit
  & the clipboard, 14–16
  & the scrapbook, 16, 17
  BASIC toolbox statements, 99
  in MacAsm, 272, 275
  in MacDS, 277, 278, 283–287
  MacsBug text edit record display, 302
  search menu, 124
  text edit, 47
  window, 12, 283
Edit Menu
  items, 15, 30, 47
  Macintosh BASIC, 123, 124
  Microsoft BASIC, 78
  MacDS, 276, 280, 283, 285
Effective address, 253, 261, 262
Eject disk, 18, 22
Empty trash, 43
Enable
  with menu manager, 158
  with MENU statement, 101, 102
EnableItem, 162, 176
EndUpdate, 201
End of file, 310–312
Enumerated data type. *See* Scalar
Erase
  Macintosh BASIC graphics, 135, 139
  window, 191, 192
Error
  68000 processor, 257, 259
  assembler, 281–283
  communications, 318–319
  Macintosh Pascal message, 64
  InLine, 156
  serial port flags, 323, 324
  stack & heap collision, 227, 232
  syntax, 86
  system, 48, 156
  undefined symbol, 292–294
Event
  activate, 164, 196, 199
  error, 133
  keyboard, 164–167, 238
  keyboard and mouse handler, 48
  OS & toolbox event managers, 48
  mask, 164
  menu 163–164
  modifier field, 165
  mouse, 166, 167, 196, 197, 232
  network, 164
  priorities, 164
  queue, 166, 174, 196
  record, 164, 165, 189, 238
  type, 163–166, 175, 196
  update, 185, 188, 200, 201
EventAvail, 166
Event manager
  & menus, 152, 162
  & windows, 166, 180, 185
  Macintosh Pascal routines, 71
  polling, 163
Exception
  68000 processing of, 250
  & the debugger, 300
  address, 251
  defined, 257–260
  handler, 226
  illegal instruction, 226
  priorities, 259
  return from, 250, 269
  trace, 251
  vector, 225, 257, 259, 264
Exec
  command file, 278, 282
  menus, 282
  minifinder, 280
  program, 279, 282
Execute Menu, 283
Extend, condition code bit, 264
Extended
  Macintosh Pascal data type, 68–71
  precision, 71, 130, 133, 141
  Typeface, 86
External speaker, 244

File. *See also* Disk & Resource file
  & disk organization, 309
  allocation blocks, 311, 312
  directory, 309–311
  finder information on, 277
  format, 273–274
  MacDS, 278, 280–287, 292, 295–299
  Macintosh, 40–43
  manager, 48, 311, 313
  servers, 316, 325
  standard package, 49
  type, 273, 278–280
File extensions, MacDS. *See* .D, .Link, .Rel, & .Rsrc
File Menu
  assembler, 288
  linker, 292
  Macintosh BASIC, 125
  MacDS, 282, 283
  Microsoft BASIC, 78
  packSyms, 297
  RMaker, 298
  use of, 11, 18, 20–22
Fill
  grafPort record fillPat, 193
  Macintosh BASIC PAINT, 135, 139
  toolbox routines, 100
Finder
  file ID codes, 273
  information field in file directory, 310–311
  use, 10–14
  user interface, 46
FindNextEvent, 167
FindWindow, 166, 167, 174, 196
FlashMenuBar, 162
Flicker
  & vertical retrace manager, 48
  & retrace interval, 232
Floating point
  IEEE standard, 123
  exceptions in Macintosh BASIC, 133
  Macintosh BASIC, 123
  Macintosh Pascal, 71
  Microsoft BASIC, 76
FlushEvents, 166
Folder
  defined, 11
  number, 311
  selecting and opening, 12
  using, 40–43
Font. *See also* TEXTFACE
  & MacDS Editor, 283, 286
  & window example program, 185
  application, 84
  defined, 40
  field grafPort, 192, 193
  in scrapbook, 17
  in window program example, 185, 192–194
  Macintosh BASIC, 122, 124–126, 139
  Macintosh Pascal, 63, 72
  manager, 47
  menus, 30, 154, 158, 162
  Microsoft BASIC, 81, 83, 85, 94, 100, 107
  mover program, 84
  printing, 92–94
  SET FONT, 138, 139
  SET FONTSIZE, 138, 139
  size, 85, 94, 99, 100
  size field in grafPort, 192
  size menu, 158, 283, 286
  style, 3, 17, 39, 40, 47
  style field in grafPort, 192
  system, 84
Fork
  & disk allocation blocks, 311–313
  & the linker, 292, 294
  defined, 274
  in file directory, 310, 311
  type, 313
Frequency
  Macintosh BASIC SOUND statement, 140
  CRT horizontal sweep, 228, 244
Function
  control, 323
  definition, 53–56
  dialog, 104
  Macintosh Pascal, 70, 71
  Microsoft BASIC, 75, 100
  Toolbox, 98–100
  transcendental, 49

Geneva font
  40, 83, 84, 239
Get Info, 18
Get menu item routines, 162
GetMenuBar, 160, 162
GetMHandle, 162
GetNextEvent, 163–166, 196
Go, 64, 65, 126
Go-away region, 183
Go-Go, 66
GOSUB, 86, 93, 133, 142
GPRINT, 138, 139
GrafPort
  & NewWindow, 191
  & Quickdraw, 192, 194
  font variables in, 191–193
  in window record, 187–189
  in window example, 194
  record definition, 192, 193
  saving and restoring, 194
  use with BeginUpdate, 201
  use with InvalRect, 199
  use with update region, 201

Graph port. *See* GrafPort
Graphics
  CRT display, 228
  documents, 11, 12
  Macintosh BASIC, 134–139
  Macintosh capabilities, 36–40
  Microsoft BASIC, 75, 80, 81, 87, 92–94
  Pascal & Quickdraw, 47, 72
  printing, 244, 245
Grow region, 183
GrowWindow, 198–199
GTEXTMODE, 138
Halt
  pause menu option, 66
  with program menu, 126
Handle
  Graph Port record, 193
  menu handle, 155–156
  menuProc field, 155
  window record, 187
Handler
  exception handler, 226, 250, 300
  keyboard and mouse handler, 48
  System Error Handler, 48
  trace exception handler, 261
Handshake, 232–233, 236–237
  line in serial port, 322
  Serial Driver Routines, 323
  Xon/Xoff, 320
Hard disk, 317
HDLC, 325
Heap, 189, 191, 274
  application, 225, 227–228
  check the heap, 302
  heap memory blocks, 301
  MacsBug's heap commands, 303
  Scramble the heap, 302
  Stack/Heap collision, 232
  system, 224–226
Hexadecimal
  calculator, 144
  MacsBug Memory Display Format, 302
  memory dump format, 103, 106
HideWindow, 192, 197
Highlighted
  active window, 199
  hilited flag, 187–188
  menu items, 152
  selected icons, 20
Hilite Menu, 162, 167, 172

I-beam
  cursor, 21
  MacBASIC text window, 123
I/O
  68000 processor, 256
  controllers, 256
  Microsoft BASIC device independent, 75
  procedures, Macintosh Pascal and UCSD Pascal differences, 68
I/O devices
  address space, 223
  device manager, 48
  events, 163
  Macintosh, 228–229
  Macintosh memory-mapped, 105, 106, 248
I/O instructions, 106, 223, 256
Icon
  defined, 10, 28
  grow icon, 192, 194–196, 199
  MacDS file icons, 278
  metacharacter, 158
  number, 158
  pointer, 65
  representing system resources, 4
  status, 129
  thumb down, 64
IEEE. *See* Floating point
IER (6522 Interrupt Enable Register), 231, 233
IFR (6522 Interrupt Flag Register), 231, 233
Imagewriter. *See* Printer
Implicit type conversion, 53
Index
  address register, 262
  MacDS address modes, 291
  program counter, 253
  register indirect, 253
Initmenus, 107–108, 160, 162
INKEY$, 142
Inline, 70–72, 156–157, 166
InlineP *see* InLine
Insertion point, 21
InsertMenu, 159, 162
Instant window, 63
Integer
  INTEGER, 164, 187
  Macintosh Basic, 132
  Pascal assignment-compatible data types, 53
  Pascal compiler supported, 53
Interpreters
  Macintosh BASIC, 122–124
  Macintosh Pascal, 52, 64–65
  Microsoft BASIC, 75–81, 89–91
  use of toolbox routines, 98–99
Interrupt
  6522 chip, 223
  68000 processor, 257–260
  programmers switch, 300
  external, 250
  OS Core, 49
  priorities, 230
  processing, 230–237
Interrupt handler
  1 second timer, 232
  6522, 231–232
  8530, 231
  operating system, 163
  vertical retrace, 232
InvalRect
  use, 199
Invert
  BASIC graphics routines, 135
  graphics statements, 139
  toolbox routines, 100
Invisible
  Show Invisibles, 286
  window attribute, 181–183, 191
Italic. *See also* TEXTFACE
  menu item type styles, 158
  style, 40, 154
Item ID, 166, 167, 174–178

Kernel
  OS Core, 49
Keyboard
  buffer, 142
  command character, 158
  events, 134, 163–166, 174
  hardware, 237–240
  input, 142
  I/O device, 228
  keyboard mouse handler, 48
Key code, 142, 164, 238–239

Labels, 75, 86, 122, 289
Latch, 233, 236
Library
  Macintosh Pascal, 71
  Microsoft BASIC, 75
  Pascal, 59, 71, 72
Line numbers, 75
Linker
  command file, 279
  commands, 294
  example, 295
  files, 278, 280
  map, 288
  program, 275, 277, 281, 292, 293
  segmentation, 274
LInlineF. *See* InLine
Lisa, 3–5
  assembler macros, 275
  MacDS debuggers, 301, 302
  Pascal, 70, 71
List window
  closing, 91
  displaying a program, 77
  tracing in, 80, 89
  with output window, 81, 86, 87
Literal, 130, 134
Local area network, 316, 324, 325
Logical block, 309, 310, 312, 313
Long integer, 53, 191
LPRINT, 86

MacAsm, 272, 273, 275, 276
MacDS, 272, 273, 276
  assembler, 287–291
  debuggers, 264, 300–305
  editor, 283–287
  exec, 279, 281–283
  files, 278
  linker, 292–295
  packSyms, 295–297
  programs, 277
  rmaker, 298, 299
  transfer menu, 278, 279
Macintosh BASIC, 6, 100
  introduction, 122
  compatibility with Macintosh Pascal, 123
  data types, 123, 130
  debugging, 129
  editing a program, 123–125
  example program, 141–149
  graphics, 134–139
  numeric operations, 133
  operators, 132
  windows, 126
Macintosh 68000 Development System. *See* MacDS
Macintosh Pascal, 6, 52–73
  characteristics, 59
  compared to UCSD & Lisa Pascal, 67–71
  data types, 53
  debugging, 64–67
  introduction, 52
  data types compared to Macintosh BASIC, 123
  program structure, 54–56
  rules of scope, 53–54

toolbox & libraries, 71, 72
windows, 62–63
Macpaint, 11, 12, 42, 137, 273
Macro, 275, 276, 289, 290
Macsbug, 276, 301–303
Macworks, 301, 302
Macwrite
running, 21
open dialog box, 31
font and style menus, 31, 40
file, 273
use with MacAsm files, 275
use with MacDS files, 278
Master directory block, 309, 310
Memory, 5, 52, 222–224
addressing, 204, 223, 224, 227, 251–255
allocation, 226
allocate/deallocate for menus, 155, 160
deallocate for windows, 191
debugger display, 301, 302, 303
display, 36, 226
dump program, 105–121
manager, 48, 226
management in Pascal, 71
map, 224–227
OS routines, 48
resident assembler, 272, 275, 276
segment, 274
sound buffer, 224, 226
Menu bar, 29, 153, 155, 159, 160
Menu bar, saving and restoring, 178
Menu designing, 29, 30
Menu ID, 154–156, 159
Menu item
check marks, 158, 176
command key, 158
defining, 157
disabled, 30
enable/disable, 153, 155, 158, 159, 176
flash, 232
icons, 153, 155, 158
ID, 174, 175
Microsoft BASIC, 101, 102
routines, 162
text, 153, 155
type styles, 158
MenuKey, 162, 166, 167, 174
Menu manager, 47, 152
CheckItem, 176
initialization, 160
new menu, 155, 156
MenuSelect, 166
routines, 162
Menu routines, 162
MenuSelect, 162, 166, 167, 175
Menu title
HiLiteMenu, 167
MenuSelect, 167
Microsoft BASIC menu statement, 102
NewMenu, 155
Metacharacter, 158, 159
Microsoft BASIC, 6, 49, 74–119
Microsoft BASIC-Macintosh BASIC comparison, 122, 134, 140
Minifinder, 31
assembler, 288
MacDS exec, 280
packSyms, 297
Rmaker, 298
Mode, 2, 3, 28, 29, 35
communications, 320, 232
printer, 244, 245, 286
transfer in grafport, 193
6522 times, 233–235
68000 address, 251–255, 261, 262, 276
68000 trace, 261
Modem, 223, 228
Modem port, 301, 316, 317
Modem plug wiring, 321, 322
Modification time
disk directory entry, 311
disk sector tag field, 313
Modifier bits, event 238, 239
Modifier field, 165
Modifier key, 239, 240
Mouse, 8, 10, 48
event, 163–167, 196, 232
hardware, 240–243
in example programs, 166, 167, 174, 175, 198, 199
interface, 229, 231, 322
Microsoft BASIC, 99
Macintosh BASIC example, 141, 142
position signals, 322
use with menu, 12, 71, 152
use with window, 184, 198, 199
MS-DOS, 2
Multiplan, 42

Negative image, 41
Negative condition code flag, 261, 262, 264
Nested Procedure, 54
Network. *See* Local area network & AppleTalk
NewMenu, 155, 156
NewWindow, 191
Note pad. *See* Desk accessory
Nub, 301
Numeric key pad, 238

Observe window, 63–65
Octal, 103
ON MENU GOSUB, 101
Open
document and start application, 273, 274
file, 17–21
file attribute, 313
folder, 41, 43
grafPort, 191
Macintosh BASIC output window, 126, 127
Macintosh BASIC text window, 125
Macintosh BASIC variable window, 129
Macintosh Pascal observe window, 64
Microsoft BASIC file, 93
object, 12
window, 181
window attribute, 182
Operating System, 5, 8–10
event handling, 163
routines, 46, 48, 49
supervisor state, 250
OS core, 49
OS event manager, 48
OS utilities, 49
Outline
font style, 40
font style in menu, 154, 158
object, 10
scrollbar, 199
window, 183, 196, 198
Oval
Macintosh BASIC, 135, 139
Microsoft BASIC, 100
Overflow condition code, 262
Overrun, 323

Package manager, 48
Packed string, 53
PAL, 223
P.A.R.C., 2, 3
Parity
in serial communications, 318–320, 323
memory, 223
Pascal. *See also* Macintosh Pascal & Lisa
string display format in MacDB, 301
strings, 290
Paste, 15, 16, 78
Pattern
in grafPort, 193
in Macintosh BASIC, 135–137, 139
scrollbar, 199
setting in Microsoft BASIC, 100, 135
Pause menu, 66
PEEK, 110
Pen, 59, 100, 134, 137. *See also* PENSIZE, PENMODE
PENMODE, 100, 137, 139
PENSIZE, 100, 134, 137, 139
Picture
in scrapbook, 16, 17
Macpaint, 42
Microsoft BASIC, 94–97
quickdraw, 184, 188, 200, 201, 245
Pixel, 36, 37, 137, 288
Plain text, 40, 86
Plot, procedure in Macintosh Pascal example, 60
PLOT, Macintosh BASIC statement, 134, 135, 139
Pointer, software
assignment error, 156
exception vector, 225
in disk blocks
in jump table, 228
Pascal data type, 53
relation to handle, 155
stack, 249, 254, 255, 258
stack notation, 291
to front window, 187
to memory block, 226, 227
to window, 188, 193
Pointer, image
icon in Macintosh BASIC, 129
icon in Macintosh Pascal, 65, 66
mouse, 10–12, 28
mouse, in menu, 153
Polygon, 66
Predecrement, 253, 254, 291
Primary display area, 224
Print
in file menu, 17, 18
Microsoft BASIC command, 77, 82, 83, 86, 97
Microsoft BASIC program listing, 91, 125, 138, 139
Microsoft BASIC, 44, 92, 93
Microsoft BASIC example, 106, 109
MacDS editor, 286
Printer, 48, 244, 245
communications, 316, 317
event, 163
port, 223, 320
Procedure
external, 71
Macintosh Pascal, 54–56
resource type, 299

Program counter, 244, 249, 253, 258
  assembler notation, 291
  MacDB window, 301
Programmer switch, 230. *See also* Reset
Protected, Microsoft BASIC program, 91, 92
Protocol, 320, 325

Quickdraw, 47, 59, 72, 123, 134, 180, 192
  globals, 227
  grafPort, 181, 187, 189, 194
  picture, 188, 200, 245

RAM. *See also* Memory
  chips, 222
  map, 227
  toolbox routines in, 5, 46, 49, 157, 225, 323
Read, Pascal I/O procedure, 68
Readln, 62, 68
Real data type, 130, 133. *See also* Floating point
Record
  event, 164–166, 238
  grafPort, 193, 194
  Macintosh Pascal, 53
  menu, 153, 154, 155, 160
  printer data, 245
  window, 187, 188, 191, 192, 200
  window & text edit display in MacsBug, 302
Rectangle, 72
  in window example program, 189, 190, 193, 194
  in window grafPort, 192, 193, 201
  Macintosh BASIC, 135, 139
Region
  of window, 180, 182, 183, 188
  window content, 183, 187, 193–196, 199–201
  window update, 185, 198, 199, 201
Register
  68000 processor, 222, 248–258
  6522 VIA, 231–237
  debugger use, 277, 300, 301, 302
  device status, 256
  field in instruction, 261, 262
  list in assembler, 290
  notation in assembler, 291
  use in assembler subroutines, 275
Relocatable block, 226, 303
Reset
  68000 machine instruction, 250, 256
  68000 vector, 258
  button, 156, 300
  menu RESET, Microsoft BASIC, 101
  option in Macintosh Pascal, 164
  serial driver, 292
Resolution
  display, 36, 228
  mouse, 240
  printer, 245
Resource compiler. *See* RMaker
Resource file, 42, 43, 47, 153, 155
  icon numbering in, 158
  key translation in, 238
  linker, 292
  MacDS, 275, 278, 280, 281
  manager, 47, 155
  RMaker, 298, 299
  type, 299
  window definition in, 186, 187
RMaker, 275–278, 280–282, 298–300
ROM
  toolbox, 5, 46, 48, 156, 157, 226, 323
  chips, 223
  addresses, 223, 227
Round rectangle, 72, 135
RS232, 316, 320–322
RS422, 320, 321
RS449, 321
Rules of scope, *see* Macintosh Pascal
Run menu
  Macintosh Pascal, 64, 65
  Microsoft BASIC, 79, 80, 82, 88, 89

SANE, 71, 102
Save
  before using InLine, 156
  dialog box, 14, 21
  Macintosh BASIC program, 125
  Microsoft BASIC program, 91–92
  registers, 275
Scalar data type, 53, 68
Scope, rules of. *See* Macintosh Pascal
Scrap manager, 47
Scrapbook, 16, 17
Scroll, 33
Scroll bar, 180, 192, 194, 195, 198–200
  MacDS editor, 283
  Microsoft BASIC, 78
Scroll button, 199
Search menu
  MacDS, 285
  Macintosh BASIC, 124, 125
  Microsoft BASIC, 79
Secondary display area, 323, 324
Sector, 307–309, 312, 313
Sector tag, 308, 312, 313
Segment, 48, 228, 274, 289, 292, 294, 295
Segment loader, 48, 228, 274
Sequential file, 75
Serial communications, 223, 245, 316, 322–324
Serial driver, 48, 323
Serial port, 222, 228, 323, 324
SetEventMask, 164, 166
Set menu item routines, 162
SetMenuBar, 160, 162
SetMenuFlash, 162
SetRect, 190
Shaft encoder, 240–243
ShowWindow, 191, 192
Signature, 273, 274, 294, 295, 299, 311
Single step. *See also* step-step
  InLine, 156
  MacDB, 302
  Macintosh BASIC, 129
  Macintosh Pascal, 64, 65
  Microsoft BASIC, 79
Size box, 134, 180, 182
SizeWindow, 198
Smalltalk, 3
Sound, 224, 226, 228, 243, 244
  buffer, 140, 224, 226
  driver, 48, 71, 231
  Macintosh BASIC, 140
  Microsoft BASIC, 75
  synthesizer, 77, 244
SOUND OVER, 140
Special menu, 47
Stack pointer, 227, 249, 254, 258, 291
Standard Apple Numeric Environment. *See* SANE
Start bit, 318, 320
Status register, 249
  assembler notation, 291
  condition code, 250, 260
  exception processing, use in, 217
  supervisor state bit, 250
  supervisor status byte, 260, 261
Step-Step, 65
StillDown, 166
Stop. *See also* Stop sign
  68000 machine instruction, 249, 250, 262
  Macintosh Pascal, 65, 66
  Microsoft BASIC, 79, 82, 87, 89–91
  sound, 140
Stop bit, 318–320, 323
Stop sign, 65, 66
String
  assembler expressions, use in, 290
  data type in Macintosh BASIC, 130
  Macintosh Pascal data type, 53
  Macintosh BASIC example, 142
  Macintosh BASIC, search in, 124
  Macintosh Pascal, manipulation in, 71
  MacDB display format, 301
  menu definition, use in, 155
  Microsoft BASIC, 75
  resource type, 229
Style, type. *See* Font style
Subdirectory, 43. *See also* Folder
Subprogram, 75
  in example program, 85, 86, 110
Supervisor state, 249, 250, 261
  in exception processing, 258
Suspend
  application, by desk accessory, 33, 34
  Microsoft BASIC program, 80
Symbol. *See also* Apple symbol
  assembler, 289, 290
  command key, 159
  file, 277, 278, 287, 290, 295–297
  Macintosh BASIC, data type in, 130, 132
  Macintosh BASIC, operator in, 132
  MacDB, 301
  undefined, in link, 292, 294
Sync character, 317
Synchronous communications, 317, 320, 323, 325
Syntax
  assembler, 276, 287, 291
  Macintosh BASIC, checking in, 122, 124, 126
  Macintosh Pascal & Microsoft BASIC, checking in, 86
Sysbeep, 174
System dispatch table, 225
System error handler, 48
System global, 153, 160, 224–226
  device addresses, 230
  keyboard translation routine pointer, 238, 239
  tick count, 232
System resource file, 42, 47. *See also* Resource file
  menu resource in, 153, 155

Tag field, 309, 312, 313
Text. *See also* Font
  document, 11
  edit record format in Macsbug, 302
  editing, 14–16, 47
  files for assembler, 282, 283, 295, 296
  file type, 278
  format for saved Microsoft BASIC program, 91
  in menu, 153–155, 157, 162
  in window title, 191

Macintosh Pascal, input in, 62
Macintosh Pascal, window in, 62
MacDS, editor, 283, 286
Microsoft BASIC, 84–86, 92, 93, 99, 100
mode, 39
printing, 145
scrapbook, 16, 17
window, 180
window example, 192, 193
Text only
file, 273, 278
editor, 277, 283
Text window
Macintosh BASIC, 123–126, 129
Macintosh Pascal, 65
menu program example, in, 175, 176
TEXTFACE, 85, 86, 94, 100
Thumb, 199, 200
Timer, in 6522, 231–234
Title bar, 32, 180, 183, 188, 191, 196
TONES, 140
Toolbox, 30, 32, 46–48
assembler usage, 274, 275
documentation, 46
InLine, access via, 156, 157
languages, usage by, 49
Macintosh BASIC usage, 123, 134, 140
Macintosh Pascal usage, 59, 71, 72
menu routines, 152, 153, 155, 160, 162
Microsoft BASIC usage, 75, 77, 94, 98–100
RAM routines, 46
ROM, 223
routines, locating, 225, 226
window routines, 180, 185
Trace
68000 processor, 250, 258, 259, 261
Macintosh BASIC, 129
MacDB, 301, 302
Microsoft BASIC, 80, 88, 89
TrackMouse, 174, 175
Transfer menu, 278, 283, 286, 292
Trap, 226, 259
assembler usage, 275
Macsbug, 302, 303
number in InLine, 72, 156, 157
number constant, 174
Trash, 311
empty, 43
icon, 20
TROFF, 80, 88, 89
TRON, 80, 88, 89
Type. *See* Font
Typeface. *See* Font

UNIX, 8, 43
Update event, 164, 196–198, 200, 201
Update region, 185, 198–201
User interface, 2–6, 8–10, 14, 29
MacDB usage, 272
Microsoft BASIC support, 74, 75, 80, 98, 100
toolbox, 46–48

Variable
global, 224, 226, 227, 230, 232
InLine, 156
link map, in, 294
local, 274
Macintosh BASIC, 122, 125–127, 129, 130
Macintosh BASIC example, 142
Macintosh Pascal, 52–54, 56, 63, 64
menu, 153
menu example program usage, 155, 160, 163, 174, 175, 178
Microsoft BASIC, 75, 80, 82, 90
Microsoft BASIC example, 81, 83, 107–110
window example program usage, 189–220
Variables window, 126, 129
Vector, 157, 230, 231, 255, 257–259, 300
Versatile interface adapter, 232, 233–237
Vertical retrace manager, 48, 232
VIA. *See* Versatile interface adapter
Video display, 223, 224, 228. *See also* Vertical retrace manager
Visible window, 181–183, 185
in window example program, 185, 187, 191, 192, 197, 200
Visible icon, flag in disk directory, 311
Volume
sound, 140
disk, 309, 310

WaitMouseUp, 166
Wedge, 72
Window, 8, 12, 32, 33, 34
as a mode indicator, 28
closing with file menu, 18
designing a program to use, 32
events, 163, 165
folder, 12, 41
frame, 183–185, 188, 198
linker, 295, 296
Macintosh BASIC, 122–127, 129, 141, 142
Macintosh Pascal, 59, 62–66, 71
Macwrite, 21
MacDB, 301
MacDS editor, 283, 284
manager, 47, 152, 180–201
manager, use for controlling menus, 152, 166, 167, 174
Microsoft BASIC, 74, 75, 77
Microsoft BASIC, closing, 91
Microsoft BASIC command, 77, 78, 81
Microsoft BASIC, controlling in, 103, 104
Microsoft BASIC example, 102, 106–110
Microsoft BASIC list, 77, 81
Microsoft BASIC, offscreen, 92, 93
Microsoft BASIC output, 77, 79, 81
Microsoft BASIC, refresh, 94
minifinder, 282, 283, 288, 297, 298
plane, 187
pointer, 188, 193
record, 187–189, 201
record in example program, 189, 191, 192, 194
record Macsbug display format, 302
resource, 274
RMaker, 299
ROM routines, 46
size box, 134
smalltalk, in, 3
structure region, 188
update event, 164, 196
WINDOW, 99, 109
WINDOW CLOSE, 109
WINDOW OUTPUT#, 86
WindowKind, 187
WindowPic, 188, 256
Write
6522 registers, 233
6522 data, 237
68000, 222
button in PackSym, 297
CRT, 232
disk, 48, 307, 309
file, 312
in Pascal text window, 62
I/O, 223, 248
memory, 5, 256
Microsoft BASIC statement, 77
Pascal statement, 62, 68
serial driver, 48, 323
text, 192
window, inactive or active, 181
Writeln, 62, 68

Xerox Corp., 2, 3. *See also* P.A.R.C.; Alto
XOn/XOff, 320, 323

Yeti, 59
program, 59–67
YetiCalc function, 59, 60, 66

More Macintosh Books from Scott, Foresman and Company

Microsoft BASIC Programming for the Mac

Master the *latest* version of Microsoft BASIC with this easy-to-understand guide. Includes 131 sample programs and sophisticated programming techniques. By Aker. $17.95, 336 pages

Doug Clapp's Jazz Book

Best-selling author Doug Clapp gives complete information on using Lotus's new integrated software package, Jazz, on the Macintosh. $19.95, 368 pages

Multiplan Mastery on the Macintosh

Leads you from the basics to advanced concepts in spreadsheet design. Includes dozens of illustrations and sample applications. By Townsend. $18.95, 368 pages

MacGraphics for Business

Learn to create high-quality business graphics with your Macintosh, using MacPaint, MacDraw, and Microsoft Chart software. Offers over 100 illustrations and numerous examples. By Mar. $17.95, 192 pages

MacCats: 99 Ways to Paint a Cat with MacPaint

This solid tutorial helps you master a variety of graphics skills on the Macintosh. Fully illustrated. By Flanagan. $9.95, 176 pages

Mac Power: Using Macintosh Software

Learn to use 11 major software packages on your Macintosh—includir MacPaint and MacDraw, Multiplan, Microsoft Word, Chart, ThinkTank and Filevision. By Munro. $15.95, 336 pages

To order,

contact your local bookstore or computer store, or send this order form to

Scott, Foresman and Company
Professional Publishing Group
1900 East Lake Avenue
Glenview, IL 60025

In Canada, contact
Macmillan of Canada
164 Commander Blvd.
Agincourt, Ontario M1S 3C7

Order Form

Send me:

____ Microsoft BASIC Programming, $17.95, 18167

____ Doug Clapp's Jazz Book, $19.95, 18266

____ Multiplan Mastery on the Macintosh, $18.95, 18182

____ MacGraphics for Business, $17.95, 18158

____ MacCats, $9.95, 18143

____ Mac Power, $15.95, 18110

☐ **Check here for a free catalog**

Please check method of payment:

☐ Check/Money Order ☐ MasterCard ☐ Visa

Amount Enclosed $_____

Credit Card No. _____

Expiration Date _____

Signature _____

Name (please print) _____

Address _____

City _____ State _____ Zip _____

Add applicable sales tax, plus 6% of Total for U.P.S.

Full payment must accompany your order. Offer good in U.S. only.

A18250

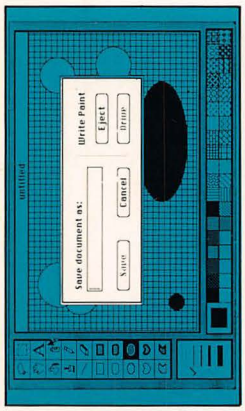
>$19.95

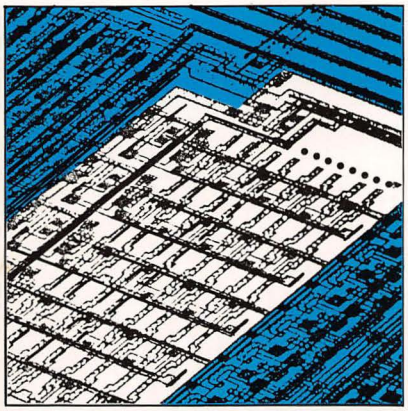
"This is the kind of book that inspires you to learn more. . . . I am especially impressed with the author's ability to present complex concepts in an extremely readable format."
—Paul Mithra, GTS, Laser Publishing Analyst

One of the first Macintosh books written for experienced programmers, this handbook explores the fundamentals of the Macintosh and its operating system in depth.

Programming the Macintosh: An Advanced Guide gives you a wealth of technical information on Macintosh hardware, software, and peripherals. This book

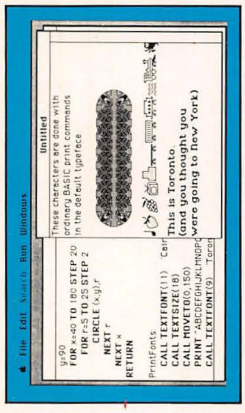
- shows you how to use Macintosh system software
- offers an introduction to the 68000 microprocessor
- explains how to program in Macintosh Pascal, Microsoft BASIC, and Macintosh BASIC
- briefly discusses each of the compilers available for the Macintosh
- shows how to use the system routines that control menus and windows
- includes sample programs in BASIC and Pascal, and more





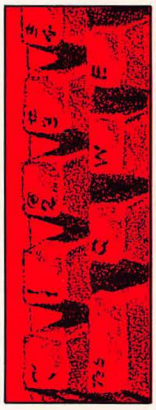
Programming the Macintosh also includes information on Macintosh communications, advice on designing programs that fit the user interface, and many useful program examples. Packed with useful information, this book will give you an in-depth understanding of the inner workings of the Macintosh.







Bill Twitty is a cofounder of a Silicon Valley consulting firm and makes his living consulting and writing. He has been designing computer hardware and software for 20 years. A resident of California, Mr. Twitty is currently working on three more books on programming the Macintosh for Scott, Foresman and Company.



Scott, Foresman and Company

ISBN 0-673-18250-9