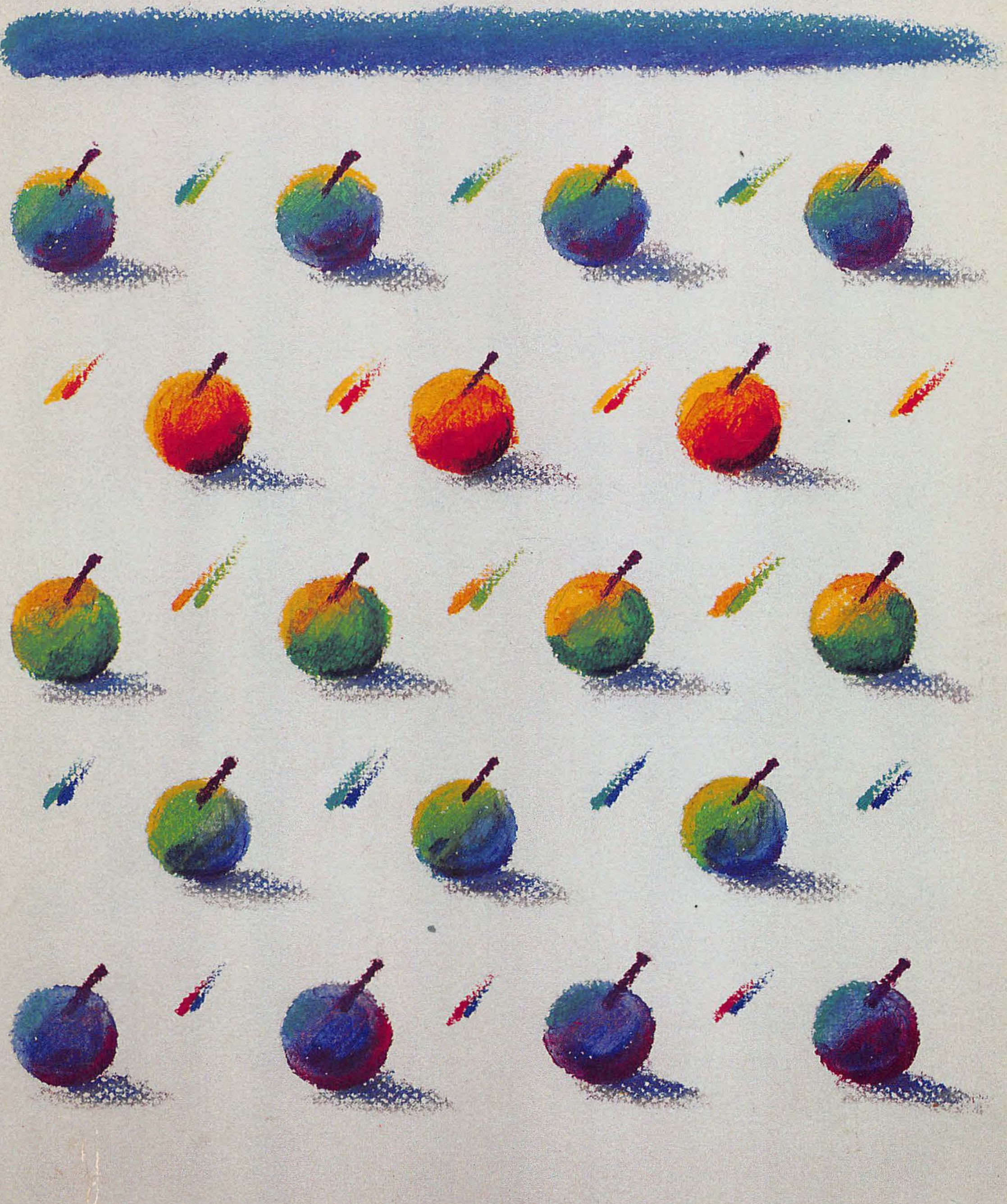
Osborne McGraw-Hill

using

Macintosh™ BASIC

Richard Norling



B177-XXT
SOFTWARE
SUPERMARKET
$17 95

Using Macintosh™ BASIC

Richard Norling

Osborne **McGraw-Hill**
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

The manuscript for this book was written using MacWrite™ running on 512K Macintosh and a Lisa 2/5 with MacWorks™.

11234567890 DODO 9989765

ISBN 0-07-88157-0

Karen Hanson, Acquisitions Editor
Dave Ushijima, Technical Editor
Ted Gartner, Copy Editor
Nancy Leahong, Text Design
Donna Behrens, Composition
Yashi Okita, Cover Design

Library of Congress Cataloging in Publication Data

Norling, Richard, 1943-
Using Macintosh BASIC.

Includes index.
1. Macintosh (Computer)—Programming. 2. Basic (Computer program language) I. Title.

QA76.8.M3N67 1985 001.64'2 85-2934

ISBN 0-07-88157-0

Contents

Acknowledgments

I want to thank all the people who worked on Apple Computer's team to develop the Macintosh BASIC language, without which this book would not have been possible.

Three of these people made special contributions. **Donn Denman,** the chief creator of the language, was informative, patient, and understanding even when I managed to interrupt his work. **Edward Spiegel** answered many questions and helped me get all the technical details right as new features were added to the language. **Marianne Hsiung** helped me understand several points. Their contributions have made this a better book.

RDN

Introduction

This book is a comprehensive guide to Apple Computer's Macintosh BASIC language. It is a complete reference to Macintosh BASIC that progresses from a simple one-line program to complicated programs using advanced computer concepts.

BASIC is the most widely used programming language on microcomputers because it is the easiest language to learn. Macintosh BASIC is even easier to learn and use than the versions of BASIC found on most other machines. It is an interactive language that displays corrective messages as soon as you enter a line that contains an error. The messages are written in normal English instead of computer jargon and are usually clear enough to enable you to correct the error without referring to written documentation.

Macintosh BASIC is incrementally compiled, a technique that makes it faster than most other versions of BASIC. Its programming environment includes a full-program editor that allows you to use all of the normal Macintosh editing techniques.The built-in interactive debugger is an effective learning and programming tool that is fun to use.

A concerted effort has been made to use normal English in this book. Many books about computers use technical terms and computer jargon so frequently that it is difficult for the non-expert to understand the concepts being discussed. This book discusses concepts in English, using technical terms and jargon only where they add to understanding.

HOW TO USE THIS BOOK

You do not have to finish reading this book before you write a computer program in Macintosh BASIC. Computer programming is best learned by doing. The most important part of learning to program computers is the experience you gain by actually writing programs. So start using commands in your programs as soon as you read about them. You will learn much more quickly if you do.

This book was organized to present the concepts of computer programming and the Macintosh BASIC language in a logical manner, laying the foundation carefully with simple concepts before building up to advanced concepts. If you want to proceed carefully and cautiously, you should start with Chapter 1 and work through the book.

The explanations, guidance, help, suggestions, reference information, and practice exercises in the book are designed both to make your learning experience go smoothly and to serve as reference materials once you have started to write programs. If you are particularly curious about a particular command or subject, feel free to jump ahead and read about it. Just keep in mind that a complete understanding of it may depend on something you skipped earlier in the book.

HOW THIS BOOK IS ORGANIZED

This book is organized into five parts. Part One, Fundamentals of Macintosh BASIC, contains eight chapters that explain the fundamentals of Macintosh BASIC. The topics include a description of Macintosh BASIC's windows and menus, operators, program editing techniques, conditional statements, simple control structures, functions, and string manipulations. By the time you finish these chapters, you will be able to write a variety of BASIC programs.

The seven chapters in Part Two, Intermediate Techniques, introduce subjects that will help you write more complicated programs. These subjects include defining data arrays, formatting output, defining your own functions, reading and writing data files on disk, using Macintosh BASIC's interactive debugger, and passing parameters to subroutines.

Part Three, Special Macintosh Techniques, describes the Macintosh BASIC commands that let you handle graphics, the mouse, and sound. While many commands are identical or similar from one version of BASIC to another, the commands described in these chapters are unique to Macintosh BASIC because they were specifically designed for use with Macintosh hardware.

Part Four, The Macintosh Toolbox, describes how to use the routines in the Macintosh toolbox. The features of Macintosh BASIC described in Part Four are not described in the Macintosh BASIC reference manual published by Apple Computer. These undocumented features of Macintosh BASIC allow you to use the Macintosh toolbox routines to handle windows, menus, controls, graphics objects, resources, and assembly language programs.

The last part, Programming Style, briefly discusses some of the things involved in polishing your program to professional quality.

A NOTE TO EXPERIENCED PROGRAMMERS

If you are an experienced programmer, you may be surprised to see what has happened to BASIC. Macintosh BASIC is a structured programming language. It gives you the ability to use either labels or line numbers, and provides new control structures like CASE and DO/LOOP, re-entrant subroutines with parameter passing, and separate programs with parameter passing and local variables.

This book teaches structured programming techniques. Standard BASIC GOTO statements are used in Chapter 5 to introduce the concept of flow of control, but appear nowhere else in the book. POP is listed in Appendix A for completeness, but is omitted from the text.

DISK OF PROGRAMS

If you prefer not to type long programs, the programs labeled as Figures in this book are available on a single Macintosh disk. The programs are intended for educational purposes only. For more information about ordering one of these disks, write to:

Language Systems Corp.
1217 E Street, S.E.
Washington, DC 20003

Part one

Fundamentals of Macintosh BASIC

Chapter 1

Getting Started

You have a disk that contains Macintosh BASIC and are ready to start programming. But where do you begin? In this chapter you will start by reviewing some elementary concepts. You will learn what a computer program does and will get an overview of how Macintosh BASIC translates programs that you write into a form that the machine can understand. Finally, a quick look at Macintosh BASIC's programming environment, including windows and menus, will get you started.

PROGRAMS

A computer is a collection of electronic parts based on simple yes/no or on/off logic. Included in the computer are a central processing unit (CPU) and random-access memory, which holds instructions and data while the computer is operating.

The central processing unit is the center of the computer's operation. It is an integrated circuit that could fit in the palm of your hand. This component performs calculations and makes logic decisions; it also sends electronic signals to the other parts of the computer to tell them when to act. The central processor chip in the Macintosh is a Motorola 68000 chip.

The computer's memory is made up of a series of units called *bits* that can store either a 1 or a 0. The bits are grouped into *bytes* (8 bits per byte). Each byte has its own address and can be accessed by using its address at any time.

While it is the center of the computer's nervous system, the central processor does not really have a mind of its own; it can only follow a strict regimen of repeatedly executing one instruction from memory and then looking for the next instruction. The series of instructions is called a *computer program.*

Once a program has been entered into the computer's memory, it can later be stored on a diskette or a hard disk. You can then reload the program at any time and tell the computer to execute it. Eventually you will write and develop a library of computer programs. When you want the computer to do something, all that will be required will be to select the proper program from your library. You will only have to write a new program when you want to do something you have not done before.

THE BASIC PROGRAMMING LANGUAGE

The instructions the central processor understands are a series of 1's and 0's called *machine language.* Only a few people directly write machine language. Instead, most people use higher-level languages that are closer to everyday spoken language. One of these higher-level languages, BASIC, has become the most popular language on small computers because it is so flexible and easy to learn. The name BASIC is an acronym of the words Beginner's All-purpose Symbolic Instruction Code.

The version of the BASIC language described in this book allows you to harness the many capabilities of the Macintosh with a total vocabulary of less than 250 words. Macintosh BASIC includes all of the standard commands of the original BASIC language. It contains new commands specially written for the Macintosh and some of the best features of another popular language, Pascal.

To allow a computer to understand programs written in BASIC, another program is needed to translate from BASIC into the machine language that the computer's central processor understands. The program labeled Macintosh BASIC does the translating.

Until recently, language translation programs could be divided into two types: compilers and interpreters. A *compiler* does its translating and error checking after the entire program has been entered. A typical compiler produces a *binary* or *object code file,* a collection of low-level machine language instructions. The binary file is executed when you run the program.

When new hardware enabled people to interact with computers on-line, *interpreters* appeared. An interpreter does not go through the intermediate step of creating a binary or object file; instead, the interpreter reads and translates each line of program text just before it is executed. This makes the interpreted program run more slowly than the equivalent compiled program. The advantage of an interpreter, however, is that it is much more convenient to use for program development because program changes can be tested without waiting for the entire program to be compiled. Most versions of the BASIC language are interpreted.

Macintosh BASIC is one of the first of a new breed of programming languages that combine the best features of compilers and interpreters. Instead of waiting until the entire program has been entered, Macintosh BASIC compiles each program line into a specially compressed form of code right after the line is entered. This reduces the time required by BASIC to compile the program before execution begins. When you run the program, the precompiled, compressed code allows Macintosh BASIC to run substantially faster than versions of BASIC that interpret everything while you run the program.

Compiling each program line just after it is entered also allows Macintosh BASIC to make programming a little easier. BASIC checks each line for simple spelling or syntax errors and notifies you immediately, so you can fix them while the program line is still fresh in your mind. (However, some types of errors that involve several lines of code can only be discovered when the program is running. You find out about these errors when they happen.)

A BASIC PROGRAM

Let's take a quick look at a short program written in Macintosh BASIC:

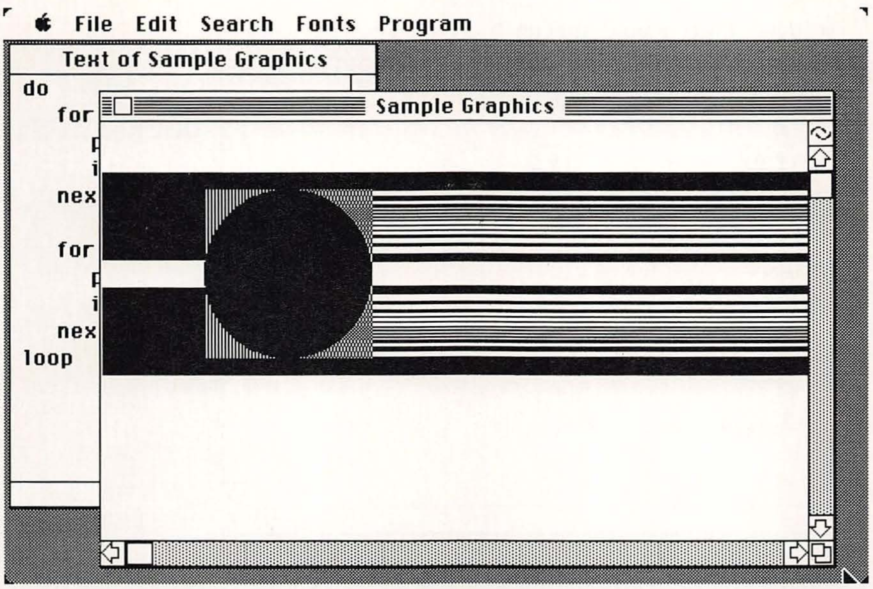
```
10 PRINT "Let's add 3 and 5 together."
20 LET A = 3 + 5
30 PRINT "3 and 5 make "; A
40 END
```

For the moment, you don't need to worry about how this program works—you just want to become familiar with how a program looks. In this example, each line contains a separate *instruction,* or program statement. No special punctuation is used at the end of the line. Each line is labeled with an identifying number. These line numbers are required in many versions of BASIC. They are optional in Macintosh BASIC—you may use line numbers if you wish, but they are not required.

This first example of a program written in Macintosh BASIC is traditional in its format. The program in Figure 1-1 shows some of

```
do
    for i = 1 to 500
        paint rect i,30;i+120,150
        invert oval i,30;i+120,150
    next i

    for i = 500 to 1 step -1
        paint oval i,40;i+100,140
        invert rect i,40;i+100,140
    next i
loop
```

Figure 1-1. Sample graphics program



```
do
    for
        p
        i
    nex

    for
        p
        i
    nex
loop
```

Figure 1-2. Output from sample graphics program

the differences in style that Macintosh BASIC allows and also shows off some of the graphics capabilities of the Macintosh. If you were to type this program into your Macintosh and run it, you would see that it moves an object back and forth across the screen, leaving a varied trail behind it. Figure 1-2 shows a picture of the program's output on the Macintosh screen.

STARTING MACINTOSH BASIC

If you have already purchased Macintosh BASIC and have a Macintosh computer handy, just insert the Macintosh BASIC disk so you can use the computer to follow along with the rest of this introductory discussion.

When the initial whirring of the disk drive stops, your screen should look something like Figure 1-3. The icon Macintosh BASIC represents the Macintosh BASIC programming language. The narrower icon with a similar design represents an individual

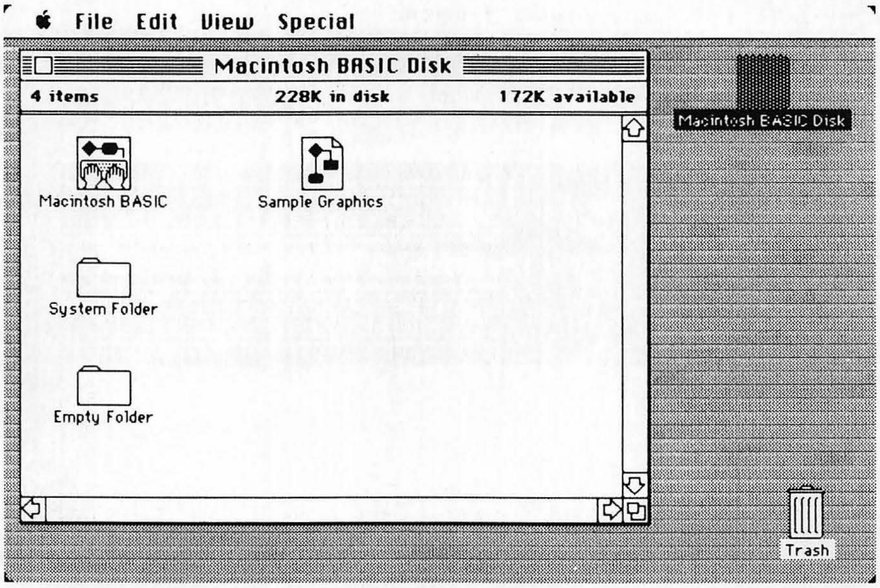


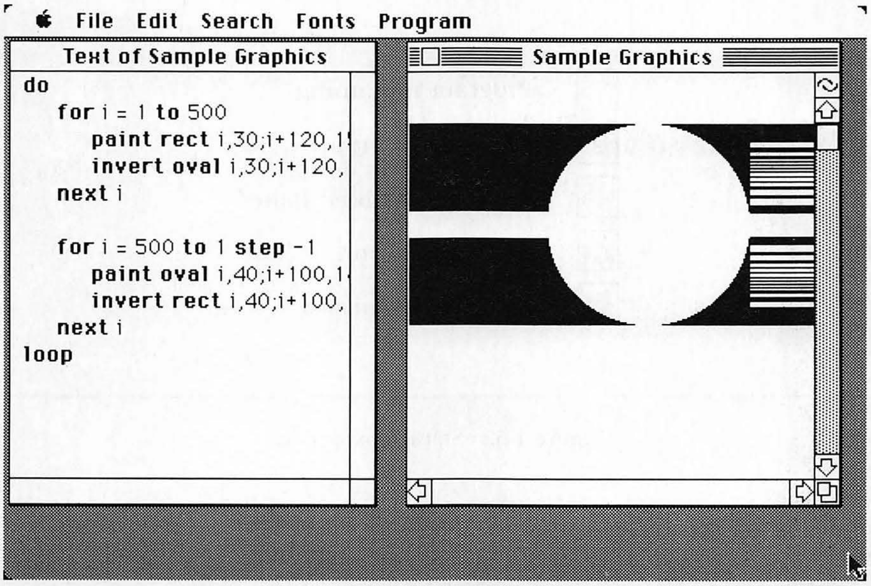
Figure 1-3. Screen after inserting Macintosh BASIC disk

program written in Macintosh BASIC. If you move the mouse so that the cursor points to the Macintosh BASIC icon and double-click the mouse button, the Macintosh BASIC program will start running. If you put the cursor on the icon of a program written in Macintosh BASIC and double-click, the Macintosh BASIC language *and* the program whose icon you clicked will be loaded into memory.

SURVEYING BASIC'S WINDOWS

Macintosh BASIC uses two major kinds of windows. One kind displays the text of your program, and the other displays the material generated by your program while it is running. Figure 1-4 shows a Macintosh screen that contains both of these kinds of windows.

Macintosh BASIC's windows have the standard Macintosh features. You can move a window by dragging its title bar and change its size by dragging the size box. If you have several windows open, you can click on a window to make it active.



```
do
   for i = 1 to 500
      paint rect i,30;i+120,1
      invert oval i,30;i+120,
   next i

   for i = 500 to 1 step -1
      paint oval i,40;i+100,1
      invert rect i,40;i+100,
   next i
loop
```

Figure 1-4. Macintosh BASIC windows

The windows have vertical and horizontal scroll bars, which become active if there is information beyond the visible area of the window. You can use the arrows at the ends of a scroll bar to scroll slowly through the document, click in the gray area to scroll one window at a time, or drag the scroll box to move directly to the part of the document you want to see.

Text Windows and Output Windows

The window on the left in Figure 1-4 is a *text window.* As its name implies, it displays the text of a program. When a program is opened, the words "Text of" precede the name of the program in the title bar of the text window. You can use all the standard Macintosh editing techniques on the program in the text window. You can type a new program directly into an untitled text window. Chapter 4 describes program entry and editing techniques.

The window on the right in Figure 1-4 is a *program output*

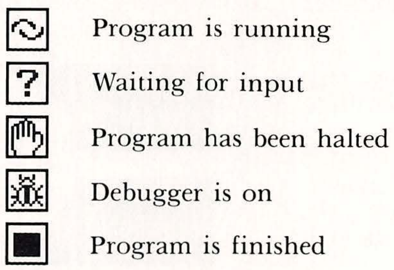


Figure 1-5. Status box designs

window, or *output window* for short. Macintosh BASIC uses output windows to display any text or graphics that your program produces. The title at the top of the output window is the name of the program file, with no extra words added.

The little square at the top of the vertical scroll bar is called the *status box.* Its purpose is to tell you the current status of your program. Figure 1-5 shows the different status box designs and their meanings. The circling design means that the program is running. The question mark indicates that the program is waiting for input from the keyboard or the mouse. An open hand indicates that you have halted the program, and the "bug" indicates that you are using the interactive debugger. (Operation of the debugger is explained in Chapter 14.) Finally, the solid rectangle indicates that your program has finished running.

THE BASIC MENUS

Now let's take a first look at the menus that define the environment in which you will work using Macintosh BASIC. Macintosh BASIC includes six full menus of commands. These are the Apple, File, Edit, Search, Fonts, and Program menus. You select items on these menus either with the mouse or with the COMMAND key options.

The Apple menu contains the About Macintosh BASIC option and lists the names of all the available desk accessories. Selecting the first option presents a dialog box about Macintosh BASIC including the author's name and the version you are using. Macintosh BASIC fully supports the desk accessories, both while you are writing a program and while your BASIC program is running. If a program that is running needs input from the keyboard or the mouse while you are using a desk accessory, the program stops and waits until you make the program's output window active again so it can receive the input.

The File menu contains commands for handling program files. The Edit and Search menus provide full editing, search, and replacement options for use in editing programs. The Fonts menu allows you to change the display font in any of Macintosh BASIC's windows. The Program menu contains options related to running, compiling, and finding errors in your programs. The items on these menus are described in the next three chapters.

Chapter 2
Creating Programs

In this chapter you will see what it takes to write a simple Macintosh BASIC program. Once you have entered the program, you will run it and save it on disk. Several of the Macintosh's menus will be indispensable in helping you create programs. In particular, the Program menu allows you to run your programs, the Fonts menu allows you to change the way your programs look, and the File menu allows you to save your programs on disk.

A SIMPLE PROGRAM

No matter how long or short a program is, it must communicate if it is to be useful. A program that predicts interest rates, the next hot stock, or the winner of Saturday's eighth race is of no use at all if it fails to communicate its results in time for you to act.

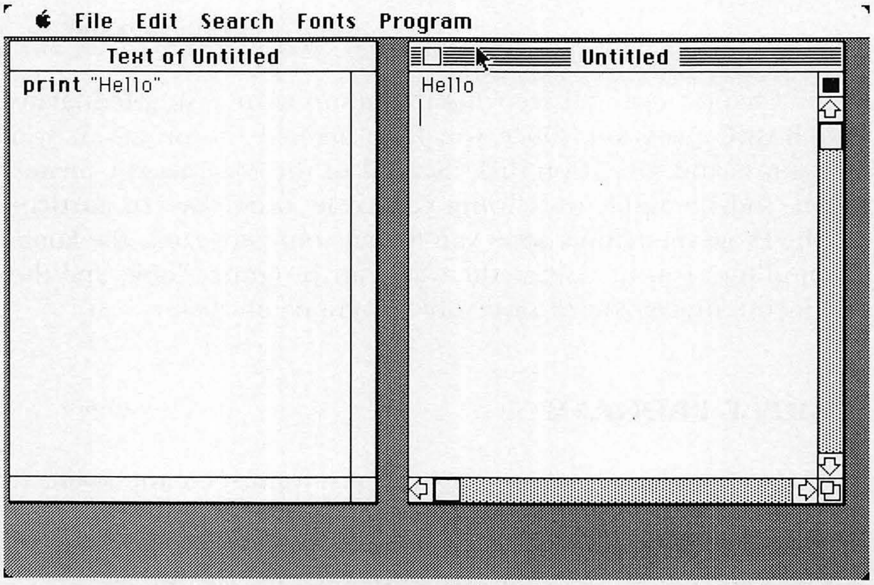
There are many ways in which a program can communicate. It can generate sound, draw pictures, print on a piece of paper, or save information on disk. The simplest way for a program to communicate, however, is to display text on the screen. That is what you will do in your first Macintosh BASIC program.

When you are ready to start, insert the Macintosh BASIC disk and double-click on the Macintosh BASIC icon. Let's write a program to print the word "Hello" on the screen. Just type in

```
print 'Hello'
```

and press the RETURN key. Now, using the mouse, move the cursor to the Program menu and select Run. If your typing was correct, the screen looks like Figure 2-1. Macintosh BASIC created the output window labeled "Untitled" to hold the program's output, and then the program printed "Hello" in the window.

Macintosh BASIC's vocabulary contains special words that are sometimes called *keywords* or *reserved words*. The word "print" is



```
print "Hello"
```

```
Hello
```

Figure 2-1. First Macintosh BASIC program

one of those words. When Macintosh BASIC recognizes one of its keywords while compiling a program line, it displays that word in boldface type in the text window. The boldface makes it easier to locate keywords in your program's text.

The word in quotation marks, "Hello", is called a *string*. A string is a collection of characters such as letters, digits, and punctuation marks. Strings are often used to hold words and sentences. You can enclose a string in either single or double quotation marks as long as you use the same type of quotation mark at both the beginning and the end of the string.

Now select the New command from the File menu, and you are ready to write another program. You do not have to close the windows from the first program before you start, because Macintosh BASIC will allow you to work on several programs at once. The only limit to the number of programs you can work on simultaneously is the amount of memory the programs occupy in the machine.

First you should decide what message you want your program to print. Once you have decided, type **print**, a space, a quotation mark, the string you want to print, and a quotation mark matching the first one. Then press the RETURN key. If you want to print more than one line, use a separate PRINT statement for each line. To see your program run, select Run from the Program menu.

PROGRAM FORMAT

Each line in your program should contain only one program statement. Macintosh BASIC allows you to put several statements on the same line if you separate them with colons. However, there is almost never any reason to put more than one statement on a line. Your program will be much easier to read if you start each program statement on a new line.

You can insert blank lines or leave extra spaces anywhere in a line to improve readability. There is no limit to the length of a statement in a program line; however, you will usually want the line to fit within the width of your text window.

You may have noticed that the word "print" in the program in this chapter is in lowercase (small) letters, while it was in uppercase (capital) letters in the first program in Chapter 1. You can use either convention. If you wish, you can even mix upper- and lower-case letters together like this:

```
pRiNt
```

The only time Macintosh BASIC pays attention to whether a letter is in upper- or lowercase is when the letter is part of a string value. Keywords can contain any mixture of upper- and lowercase letters. For consistency, Macintosh BASIC commands or keywords, like PRINT, will appear in all capital letters in most of the programs in this book.

WORKING WITH PROGRAM FILES

Figure 2-2 shows the File menu. The items on this menu are selected most often while writing and editing programs. The New command opens a new text window for entry of a new program. The window is labeled "Untitled." The Open Program file command presents a dialog box from which you may select an existing program to be retrieved from disk and displayed in a text window.

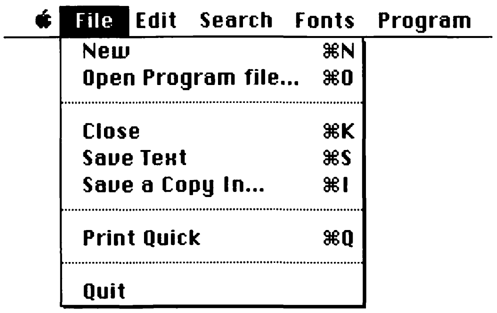


Figure 2-2. The File menu

Once in the window, the program can be edited or executed. When you open a long program, there may be a short delay before the listing of the program appears in the text window. Macintosh BASIC checks the program for errors and compiles it into a shorter form during this delay.

The Close command closes the active window. If a text window is active and the program has not been saved since it was last changed (or if it is a new program that has never been saved at all), a dialog box asks whether you want to save the changes before closing. If you answer no, the latest version of the program will not be saved, and the copy of the program on the disk will not include any changes made since the last time the program was written to the disk.

The Save Text command copies the program in the active text window to the disk with the same file name that appears at the top of the window. If the window is untitled, Save Text will present a dialog box that asks you to give the program a name. The name you assign is given to the disk file, and at the same time it is displayed at the top of the text window. The Save a Copy In command allows you to save a copy of the program in the active text window under a different name. Save a Copy In does not change the title of the text window. The two Save commands work only when a text window is active. Both Save commands leave your program in the text window unchanged.

RUNNING YOUR PROGRAM

The Program menu, shown in Figure 2-3, contains the commands to run a program once it has been written. The Run command opens an output window and starts program execution from the beginning. If an output window is already open for the program, Run starts the program again from the beginning in the same window.

You can run several programs at once in Macintosh BASIC. You can even run more than one copy of the same program. The price you pay is that each program runs more slowly. The only limit on the number of programs running at the same time is the amount of memory in your machine. You can also edit a program while another program, or a copy of the same program, is running.

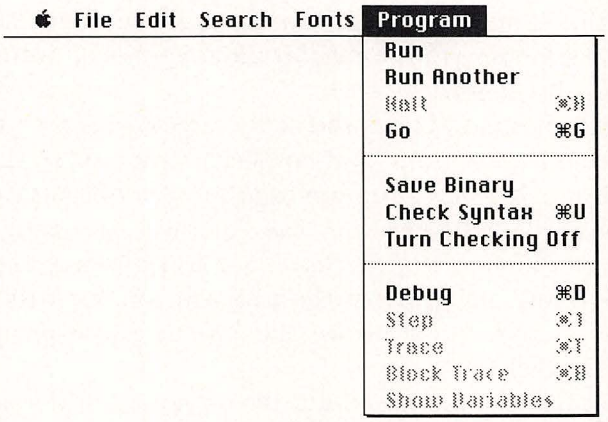


Figure 2-3. The Program menu

Macintosh BASIC accomplishes this feat by dividing up the available time among all the competing tasks, switching from one task to another as often as sixty times a second. The Run Another command opens a new output window and runs another copy of the program whose window is active.

The Halt command interrupts program execution, and the Go command resumes it. Both commands affect the program whose output window is active. If you select Go when a program text window is active, Go starts program execution just like the Run command.

The Save Binary command saves a program to disk in a compact form that cannot be translated back into the original program text. This is the form in which programs are saved once they appear to be working properly. A binary program file occupies less space on the disk than a normal text file, and it also loads faster because compilation and error checking do not need to be performed every time the program is read from disk. Save Binary appends the letters ".Bin" to the program's file name to give the binary file a different name on the disk. The design on a binary file's icon is outlined instead of solid. Even though you may be absolutely convinced that a program has been perfected, you should always keep at least one copy of the full text of the program in case further changes are needed.

The Check Syntax command lets you scan a program for errors whenever you wish and also lets you update a running program. The Turn Checking Off command turns off BASIC's automatic error checking. When checking is off, the command on the menu toggles to Turn Checking On. The Debug command turns on the Macintosh BASIC debugger, a tool that helps locate program errors. The Step, Trace, Block Trace, and Show Variables commands are all part of the debugger, which is described thoroughly in Chapter 14.

CHANGING FONTS IN A WINDOW

The Fonts menu lists the seven most common font sizes plus all of the type fonts currently available. The list of fonts in the menu depends upon which fonts are in the System file on the Macintosh BASIC disk. The System file on the disk for Figure 2-4 contained four fonts. The Font Mover program on the Macintosh System

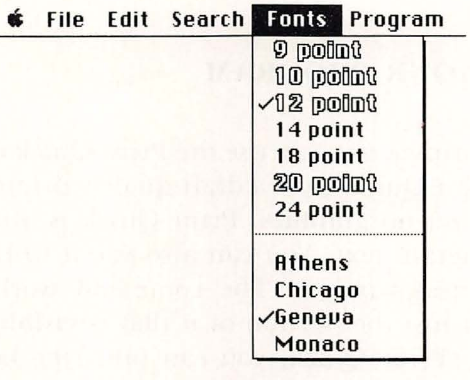


Figure 2-4. The Fonts menu

Disk will move fonts in and out of the System file. If a font you want to use is missing from the menu, you can copy it into the System file with the Font Mover program. If the list on the Fonts menu contains any fonts that you never use, you can save space on your diskette by using the Font Mover to remove any unwanted fonts.

The Fonts menu allows you to change the appearance of text displayed in any window, whether it is a program's output window, a text window, or even the Clipboard. To change the font or size of text in a window, first make that window active by clicking the mouse button while the cursor is inside the window. Now any selections from the Fonts menu will affect the font and size of text in that window. Checkmarks appear on the menu alongside the font and size that are currently selected. Unless you change the font or type size, Macintosh BASIC uses the Geneva font in the 12 point size for all windows.

If you change the font or type size for a window that already contains text, BASIC will erase the text and redraw it in the font and size you specify. Any non-text material in the window, such as a graphics design drawn by a program, will be erased from the display and will not be redrawn. This erasure affects only the display of the information, not the information itself. Thus, if the information was in a place like the Clipboard, you would still be able to retrieve it, even though it no longer appeared on the display.

PRINTING YOUR PROGRAM

If you own a printer, you can use the Print Quick command on the File menu. Print Quick gives a draft-quality printout that includes ordinary text but no graphics. Print Quick is often used to print listings of program text. You can also use it to print text from a program's output window. The command works on the entire document, not just the portion of it that is visible in the window. In addition to Print Quick, you can press the key combinations shown in Table 2-1 at any time to print an image of the active window or the full screen, or to copy the screen to disk for use with the MacPaint program.

Table 2-1. Finder Commands to Print or Copy

COMMAND-SHIFT-3	Copy screen to disk
COMMAND-SHIFT-4	Print active window
CAPS LOCK-COMMAND-SHIFT-4	Print entire screen

QUITTING MACINTOSH BASIC

The Quit command on the File menu provides an exit from BASIC back to the Finder where you will see the familiar desktop. If you have made changes to a program since it was last saved, BASIC presents a dialog box and gives you an opportunity to save the latest version of the program.

Chapter 3
Statements and Operators

Commands:

- LET, PRINT, ?, INPUT, REM, !
- END, END PROGRAM, END MAIN

Operators:

- =, +, −, *, /, ^, DIV, MOD

This chapter introduces several of the most common BASIC commands and shows you how to organize them into statements in a working program. The first step in writing a computer program is deciding what you want the computer to do. Simple programs that do only one thing are easy to write. For more complicated programs, however, deciding what you want the computer to do is often the most difficult part of writing a program.

In addition to introducing the simple commands and operators that are used in almost every program, this chapter begins the practice of specifying *what* a program does before it is written. This simple habit will make planning and writing programs easier later on when you tackle more complex programming problems.

ASSIGNING VALUES TO VARIABLES

■ LET, =

Variables are useful in programs because variables can hold values for future reference. However, each variable can hold only one type of value; thus the variables are classified by their content. The two most common types are *numeric variables* (which hold numbers) and *string variables* (which hold strings). Each variable holds only one value. When a new value is stored in the variable, it replaces the previous value. As shown in Figure 3-1, variables can be visualized as specialized mailboxes that hold only a single piece of mail at a time. If you attempt to store a value such as a string into a numeric variable or a number into a string variable, Macintosh BASIC will present you with a "Type Mismatch" message.

Each variable has a name, which refers to the value stored in the variable. A variable name starts with a letter and may contain letters, numbers, and some special characters. However, the name of a variable cannot contain certain characters used in Macintosh BASIC statements as operators, commands, or punctuation marks.

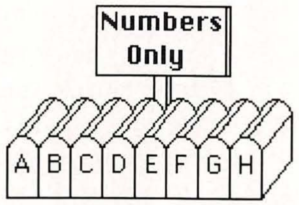


Figure 3-1. Numeric variable "mailboxes"

Table 3-1. Characters to Avoid in Variable Names

Character	Function
(space)	Word separator
, ; ( )	Punctuation
:	Statement separator
=	Replacement operator
' "	String delimiters
!	Remark character
+ − * / ^	Arithmetic operators
&	String operator
$< > = \neq \geq \leq$	Relational operators
@	Two-way parameter marker
?	PRINT statement abbreviation
#	Channel designator

Table 3-1 lists characters that may not be used in variable names.

A variable name may contain a Macintosh BASIC reserved word, but the name may not be identical to the word. Macintosh BASIC treats upper- and lowercase letters the same in variable names, so even though *varname* and *VARNAME* look different, they are in fact different ways of writing the same variable name. Variable names will usually be lowercase in this book.

Macintosh BASIC allows variable names to be as short as one letter or as long as 255 characters. This improves a program's readability because descriptive and easy-to-understand variable names can be used. Statements like

```
pr = rcts - c
```

can become

```
profit = receipts - cost
```

If you use descriptive variable names, your programs will be easy to improve or modify should the need arise.

Because variable names can be long but must be all one word with no spaces, people have invented different ways of writing long names. These include running the words together or using some special character like a period to replace spaces. Thus, depending on your style, a variable that contains the value of a test score might be labeled *testscore, testScore,* or *test.score.* As a practical matter, names longer than eight or ten characters can lead to errors because they are difficult to remember and can be difficult to type accurately every time.

Several characters have special meaning when used as the last character of a variable name. The most common of these is the dollar sign ($), which marks the name of a string variable. Any variable that does not have a special character at the end of its name is a numeric variable.

The most common way to assign a value to a variable is with the replacement statement, LET. The statement

```
LET a = 3
```

creates a numeric variable named *a* and puts the value 3 in it. When a variable name is used in a formula or a BASIC statement, Macintosh BASIC uses the value of the variable. Thus, the sequence

```
LET a = 3
PRINT a
```

sets the variable *a* equal to 3 and then prints the number 3 in the program's output window. The LET command works just the same for string variables. The program

```
message$ = "Hello"
PRINT message$
```

prints the word "Hello" in the output window just like the program you saw in Chapter 2.

```
PRINT "Hello"
```

The replacement operation is used very frequently in BASIC programs. To save keystrokes when typing in programs, the use of the word LET is entirely optional. A statement that starts with a variable name followed by an equal sign has the same effect as if it started with LET. Thus,

```
a = 7
PRINT a
```

prints the number seven in the output window just as

```
LET a = 7
PRINT a
```

would do. Whether or not you actually type the word LET is up to you.

DOING CALCULATIONS

■ +, −, *, /, ^, DIV, MOD

Four of the arithmetic operators — the signs for addition, subtraction, multiplication, and division — are probably familiar to you. The other arithmetic operators — exponentiation, integer division, and modulo — may not be nearly as well known. All seven operators are summarized in Table 3-2.

Table 3-2. Arithmetic Operators

Operator	Operation	Example
+	Addition	3 + 2 = 5
−	Subtraction	3 − 2 = 1
*	Multiplication	3 * 2 = 6
/	Division	3 / 2 = 1.5
^	Exponentiation	3 ^ 2 = 9
DIV	Integer division	5 DIV 2 = 2
MOD	Modulo	5 MOD 2 = 1

Note that while a formula such as a=3b+2 is allowed in mathematics, in BASIC the formula must be written a=3*b+2 with the multiplication operator used instead of implied.

Exponentiation is sometimes described as "raising a number to a power." The normal mathematical notation for exponentiation is x^n, which would be read as "x raised to the *n*th power." To avoid the difficulties in trying to use superscripts, Macintosh BASIC uses the notation x^n. The ^ symbol is above the 6 on the Macintosh keyboard and can be obtained by typing 6 while holding the SHIFT key down.

The *n* in x^n is called an *exponent,* which is where exponentiation gets its name. The value of x^n is x multiplied by itself *n* times. For example, the value of 3^2 is 3*3, or 9. The value of 2^3 is 2*2*2, or 8. Exponentiation is not as common as other arithmetic operators, but it does show up in a number of useful formulas.

DIV and MOD are integer operators. DIV is called the integer division operator. It does a normal division and then returns the integer portion of the result. The modulo operator, MOD, returns the remainder of an integer division.

```
a = 7 DIV 2  ! Puts 3 in a
b = 7 MOD 2  ! Puts 1 in b
```

Evaluating Expressions

An *expression* is any combination of values, operators, variables, and parentheses that can be evaluated to produce a single value. A numeric expression can be a simple number or numeric variable name, or it can be a complicated formula. This section discusses how BASIC evaluates formulas.

If a formula contains more than one arithmetic operation, it is important to know in what order the operations will be performed. The formula a=7+3*4^2−6/2 would set a to 797 if every operation were done in a strict left-to-right order, but in BASIC, the formula yields 52 because BASIC follows the rules of mathematics regarding which operations get performed first (officially called the *order of precedence*).

Table 3-3 lists the order of precedence for the seven arithmetic operators. Exponentiation is always performed first. Multiplication,

Table 3-3. Order of Precedence

Operator	Operation	Order
^	Exponentiation	First
* / DIV MOD	Multiplication, division, integer division, modulo	Second
+ −	Addition, subtraction	Third

division, integer division, and modulo are done next. Addition and subtraction are done after all the other arithmetic operations. Thus, in the formula in the previous paragraph, the exponentiation 4^2 is performed first, giving a value of 16. Now the formula can be reduced to a=7+3*16−6/2, which contains one multiplication and one division. Since those two operations are equal in precedence, they are performed in a left-to-right order. Completing the multiplication and division reduces the formula to a=7+48−3, and the calculation can be completed by doing the addition and subtraction in left-to-right order to give the answer 52.

Using Parentheses in Formulas

You can force BASIC to evaluate a formula in the order you desire, regardless of the rules of precedence, by using parentheses in your formula. BASIC evaluates everything inside a set of parentheses before proceeding with a calculation. If several sets of parentheses are nested inside each other, the calculations in the innermost set of parentheses are performed first. If the formula you evaluated earlier is changed to a=((7+3)*4)^2−6/2, then the expression in the innermost set of parentheses, 7+3, is evaluated first, producing 10. The remaining set of parentheses forces the computation 10*4 to be performed next. This reduces the formula to a=40^2−6/2, which can now be evaluated by the normal order of precedence to 1600−3, or 1597.

Even when it is not necessary to change the computation order, parentheses are sometimes used to clarify the order of calculations. It also may be easier to use parentheses in a formula than to look up the official order of precedence. If an attempt is made to enter a formula that has a different number of left and right parentheses, BASIC will present an error message.

MORE ABOUT PRINT

■ PRINT, ?

You have already used the PRINT statement in your first program, but there is still more to learn about it. You can print as many numbers and strings as you want in a single PRINT statement, or you can use more than one PRINT statement to print things on the same line of the output window. The statement

```
PRINT "Testing... "; 1; "   "; 2; "   "; 3
```

prints the string "Testing..." immediately followed by the number 1, the spaces in the next string, the number 2, the final string, and the number 3.

Items to be printed must be separated from each other by semicolons or commas. Semicolons, as used in this example, cause the items to be printed adjacent to each other. Macintosh BASIC creates tab stops in the output window a little more than an inch apart. When a comma is used as a separator, the comma causes the next item to be printed at the next tab stop.

A simple PRINT statement with nothing listed after it prints a blank line. Occasionally, you need to use more than one PRINT statement for things on the same output line. Using either a semicolon or a comma at the very end of the PRINT statement will cause the next PRINT statement to print on the same line. A trailing semicolon will cause the next character to be printed just after the last item printed. A trailing comma will cause the next character to be printed at the next tab stop.

An average output window will hold about fifteen lines of information printed in the 12 point font size. When the window becomes full, Macintosh BASIC starts scrolling the information vertically so that the last line printed is visible in the bottom of the window. You can view the lines that were scrolled off the top of the window by using the scroll bar along the right side of the window.

The items to be printed can be any valid BASIC expression—as simple as the actual numbers and strings in the previous example or as complex as you want. The statement

```
PRINT ((7 + 3) * 4) ^ 2 - 6 / 2
```

prints the number 1597 just as surely as

```
PRINT 1597
```

does. BASIC allows you to use a question mark instead of the word PRINT if you wish.

GETTING INFORMATION FROM THE KEYBOARD

■ INPUT

INPUT is the command that allows a program to receive information typed from the keyboard and place it in a variable for further use. The statement

```
INPUT a
```

puts the value typed from the keyboard into the variable named *a*. The information typed at the keyboard must be of the same type as the variable which is to receive the information. Thus,

```
INPUT a$
```

will receive a string of information and store it in the variable *a$*, while

```
INPUT a
```

will receive a number and store it in the variable *a*. If a non-numeric character is typed when the INPUT statement requires a number, Macintosh BASIC will refuse the keyboard input and give an "Expected a Number" message.

You need to receive some indication when a program is waiting for you to type something. To meet this need, BASIC allows you to specify a prompt string as part of your INPUT statement. The format is

```
INPUT "Prompt"; variable
```

BASIC prints the specified prompt string before waiting for the typed input. The prompt specified in the INPUT statement must be an actual string, not the name of a string variable. If you do not supply a prompt string, the INPUT statement prints a question mark. The statement

```
INPUT "Type your age, please: "; age
```

prints the prompt string "Type your age, please:" and then waits for you to type a number. You can backspace, cut, paste, retype, and use all of the Macintosh editing techniques while you are entering the number. You press the RETURN key to tell BASIC to accept your input line.

If you do not want either a prompt string or the question mark, you can use an empty string (just a set of quotation marks with nothing between them) as the prompt. Thus,

```
INPUT ""; a$
```

does not print anything; it just waits for a string to be typed. You should not use this technique unless you have already used a PRINT statement to display an appropriate prompt message.

LEAVING NOTES TO YOURSELF

■ REM, !

REM, which is short for "remarks," allows you to leave comments in the text of a program without affecting the way the program operates. Anything you put on a program line after REM will be kept in the program text as a note, but will be ignored during compilation and execution. Many programmers put several REM statements at the beginning of each program to describe what the program does, when it was last updated, the author's name, and other pertinent information. Comments are also used frequently to identify major sections of a long program. When REM is used, it must be preceded by a colon and followed by a space.

REM has an abbreviation, the exclamation point (!). The exclamation point can be used anywhere REM can be used, and it can also be used where REM cannot be—on a program line to add comments after another BASIC statement, for example. The exclamation point does not have to be followed by a space.

```
REM This is a comment
! This is a comment also
a=3 !Use ! but not REM here
```

ENDING THINGS NEATLY

■ END, END PROGRAM, END MAIN

The END statement is used to mark the end of a BASIC program. If END is not included, a program will stop executing when it runs out of instructions. This does not usually cause any difficulty, but it is still a good idea to use END in every program to mark the point where execution stops.

END PROGRAM and END MAIN are alternate ways of writing the END statement. END does not have to appear on the last line of a program. If it is executed earlier in the program, execution stops instantly. This feature is sometimes used to end a program early.

```
END  ! Ends the program
END PROGRAM  ! Also ends it
END MAIN  ! Another way to end it
```

EXAMPLE PROGRAMS

Your Macintosh BASIC vocabulary now includes commands that allow you to write programs that get data from the keyboard, use formulas to make calculations from that data, and display the answers on the screen. An example of the type of program that you can write (shown in Figure 3-2) converts a distance from kilometers to miles.

The first line is a remark that describes what the program does. It is ignored during program execution. The INPUT statement in the second line prints the prompt string "Kilometers:" and then waits for you to enter a number to store in the variable named *kilometers*. The LET statement in the third line divides the number in *kilometers* by 0.62 and stores the result in a new variable, *miles*. The formula to the right of the exclamation point is the programmer's note and will not be executed. The first PRINT statement skips a line, and the next PRINT statement displays the answer in the output window along with the appropriate measurement units. The END PROGRAM statement concludes the program.

The example program in Figure 3-3 calculates gasoline mileage. After the initial remark statement that tells you what the program does, this program prints a title in the output window and then skips a line to separate the title from the questions that will be asked next. Two INPUT statements ask you to enter the present and past mileage readings from the car's odometer. The program skips another line and then requests the number of gallons of gasoline to fill the tank.

The next line contains an implied LET statement. The parentheses around the expression "now—then" force the subtraction to be performed before the division. Finally, the program skips a line

```
! Convert kilometers to miles
INPUT "Kilometers: "; kilometers
LET miles = kilometers / 0.62   ! 1 kilometer= 0.62 miles
PRINT
PRINT kilometers; "  kilometers = "; miles; "  miles."
END PROGRAM
```

Figure 3-2. Convert kilometers to miles

```
! Calculate gasoline mileage
PRINT "GASOLINE MILEAGE CALCULATOR"
PRINT
INPUT "What is your odometer mileage right now? "; now
INPUT "What was it last time you bought gasoline? "; then
PRINT
INPUT "How many gallons did it take to fill your tank? "; gas
mileage = (now - then) / gas
PRINT
PRINT "Your car traveled "; mileage; " miles per gallon."
END PROGRAM
```

Figure 3-3. Calculate gasoline mileage

and displays the answer in a clear manner that includes the units of measurement.

Notice that only one of the eleven lines in this program actually performed a calculation. The other ten lines helped to make the program easily understandable and to request and display information in a clear and friendly manner. This ratio is not unusual. Your programs should communicate with people in clear language instead of computer jargon. This may result in programs that are slightly longer, but they will be useful to a much larger group of people.

PRACTICE EXERCISES

1. Which of the following will be accepted by Macintosh BASIC as names for numeric variables? for string variables?
   a. Total sales
   b. answer$
   c. streetNumber
   d. White-Cell.Count
   e. bachelor#1
   f. The.total.amount.of.money.I.made.last.year
   g. 4teen
2. What value is stored in each of these variables?
   a. amount = 3 + 3 * 2
   b. size = 6 − 3 ^ 2 + 7
   c. number = 2 / (3 − 1) * 8
   d. rate = 3 * (2 + 1) ^ 2
3. Write a program that asks you for your name and then greets you by name.
4. Write a program that converts meters to inches (hint: each meter contains 39.37 inches). Don't forget to have the program ask you for the number of meters to convert and display the results.

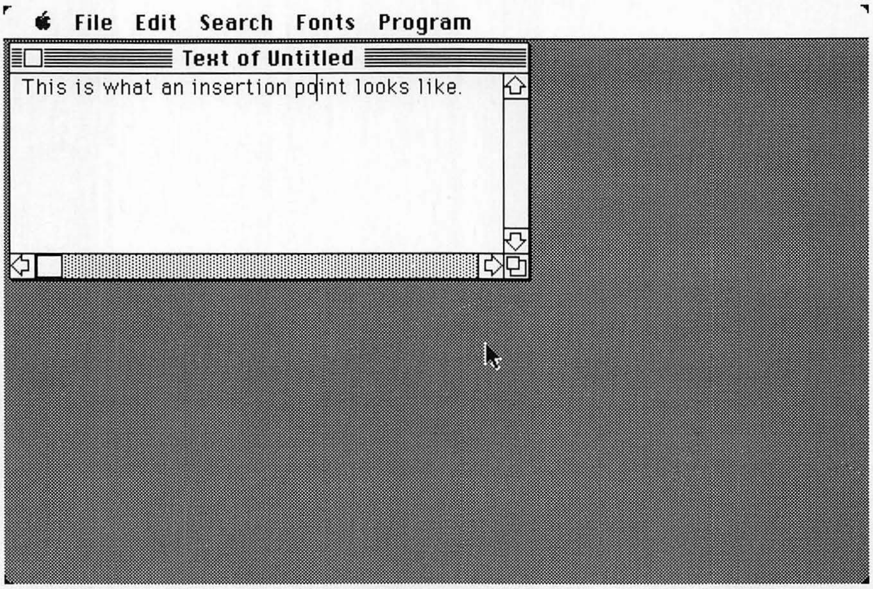
Chapter 4

Editing Programs

This chapter describes the techniques used to enter and edit programs in Macintosh BASIC. If you have experience with MacWrite or another Macintosh word processing program, the text selection and editing techniques will be familiar. You could, if you wished, type your program into a word processing program and then let Macintosh BASIC read the program from the text file created by the word processor. However, Macintosh BASIC provides a full set of text-editing commands, including global search and replace, so typing your program directly into a Macintosh BASIC text window is the best and simplest way to enter it.

SELECTING TEXT WITH THE MOUSE

When you open a new listing window, the *insertion point* is located in the first character position. The insertion point is



This is what an insertion point looks like.

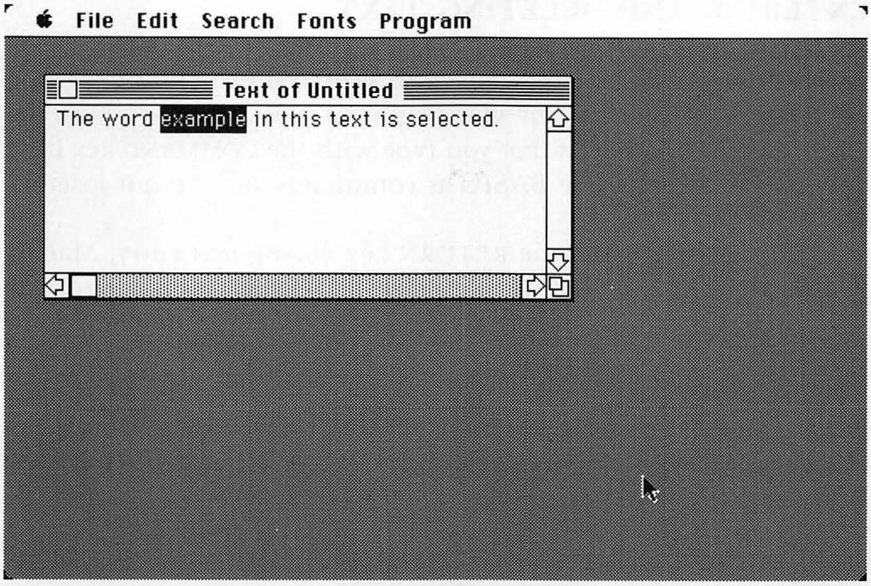
Figure 4-1. Insertion point in text

marked by a blinking vertical line. Figure 4-1 shows an insertion point between the "o" and "i" in the word "point." Any text that you type at the keyboard or transfer with the Paste command will be inserted into your program at the insertion point.

To select a portion of a program line, or to select several lines, move the mouse until the cursor is at either the beginning or the end of the area you want to select. Then press and hold the mouse button down while you move the cursor to the opposite end of the area you want to select (this is called *dragging*). As you move the mouse, the selected area is displayed as white letters on a black background. Release the mouse button to mark the end of your selection. Figure 4-2 shows a program line in which the word "example" has been selected.

You can select parts of your program that extend beyond the edge of the window, as long as you can see the beginning of the area you want to select. When you drag the cursor slightly outside the edge of the text window, the text will automatically scroll.

The SHIFT-click technique is an excellent way to select a large block of your program. To select text with SHIFT-click, set the



The word example in this text is selected.

Figure 4-2. Selected text

insertion point at one end of the area you want to select. Then you can scroll the window to position the cursor at the other end of the block. Hold the SHIFT key down while you click the mouse button, and you will select the portion of your program between the insertion point and the cursor.

Two additional shortcuts can help select parts of your program text. To select a single word, you can place the cursor anywhere on that word and double-click the mouse button. This is usually faster than dragging the cursor over the word. To select the entire program in the active listing window, you can use the Select All command on the Edit menu.

Once you have selected part of your program, you can delete or copy it. Selected text can be deleted by typing on the keyboard or by selecting the Cut, Paste, or Clear command from the Edit menu. You can copy the selected block with the Copy command. Because selected text can be easily deleted, it is not wise to leave important portions of your program selected for any length of time. Deselect text by clicking the mouse button in the text window to reposition the insertion point.

ENTERING AND DELETING TEXT

The simplest way to enter text is to start typing. If part of your program is selected when you begin to type, that part will be replaced. Any characters that you type with the COMMAND key held down are interpreted by BASIC as commands and are not inserted into your text.

Every time you press the RETURN key during text entry, Macintosh BASIC inspects the line you have just entered for any errors. If you want this error checking to be postponed until after you have entered all of your program text, you can select Turn Checking Off from the Program menu. Error checking also occurs when you paste material into your program from the Clipboard.

The most common way to delete program text you have selected is to press the BACKSPACE key located in the upper-right corner of the keyboard. The CLEAR key on the optional numeric keypad also deletes selected text. If you want to replace the selected text with new text, you do not have to delete it first. It is deleted automatically as soon as you type the first character of the replacement text. Several of the edit commands introduced in the next section — Cut, Paste, and Clear — also delete text that has been selected.

THE EDITING COMMANDS

Figure 4-3 shows the Edit menu. When a command appears in gray type instead of black, it is disabled and cannot be used at that particular time. Many of the edit commands can be issued from the keyboard as well as from the menu. The COMMAND-key codes are listed to the right of the command names in the menu.

Cut

Cut moves the selected text from the program to the Clipboard. The insertion point remains at the location where the selected text was removed. The previous contents of the Clipboard are completely replaced. If no text is selected, Cut empties the Clipboard. COMMAND-X is the keyboard command for Cut.

Cut is used most often to move a block of text from one place to

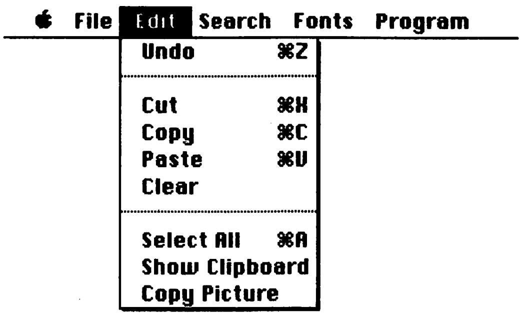


Figure 4-3. The Edit menu

another. First you cut the text from its old location; then you paste it into its new location. The Clipboard is merely the place that holds the text on its way to the new location.

Copy

Copy moves a copy of the selected text to the Clipboard. The selected text completely replaces the previous contents of the Clipboard. If no text is selected, Copy empties the Clipboard. Copy does not change the text in the listing window. The keyboard command for Copy is COMMAND-C.

Like Cut, Copy is used with Paste to move information to a new location. However, Copy does not delete the text from its original location.

Paste

Paste replaces the selected text with the contents of the Clipboard. If no text was selected, Paste inserts the contents of the Clipboard at the insertion point. COMMAND-V is the keyboard command for Paste.

There must be some text on the Clipboard, or the Paste command will have no effect. The most common ways to transfer text to the Clipboard are by using the Cut or Copy commands. Text can also be transferred to the Clipboard by another application program. Paste works only with text in Macintosh BASIC and does nothing if the Clipboard contains a picture.

Clear

Clear removes the selected text from the listing window. The Clear command has no effect on the Clipboard. Selecting the Clear command is the same as pressing the BACKSPACE key (or the CLEAR key on the numeric keypad) while text is selected.

Select All

Select All selects your entire program. Once the program is selected, you can copy it to the Clipboard or delete it with Cut or Clear. The keyboard abbreviation for Select All is COMMAND-A.

Undo

Undo restores both the text in the active listing window and the Clipboard to the status they had just before you executed the last edit command. Insertion points and selected text are also restored. Undo works after typing or after selecting the Cut, Copy, Paste, Clear, or Replace commands. However, Undo does not have the ability to reverse the effects of Replace All.

Events in another window may prevent the use of Undo. Typing or editing commands in another window can affect the command to be undone in the original window and may also change the contents of the Clipboard. The best rule is to use Undo immediately after the event you want to undo, without any intervening actions.

If you choose Undo twice in a row, the second Undo command reverses the effects of the first. This leaves your program and the Clipboard as they were before you used Undo. The keyboard command for Undo is COMMAND-Z.

Show Clipboard

The Show Clipboard command changes to suit the situation. If the Clipboard is not visible, the Show Clipboard command appears in the Edit menu. If the Clipboard is showing, the command becomes Hide Clipboard. When Show Clipboard is selected, Macintosh BASIC opens the Clipboard window in the lower-left corner of the screen. Hide Clipboard has the same effect as clicking on the Clipboard to make it the active window and then closing it by clicking on the close box or selecting Close from the File menu. The advantage of Hide Clipboard is that you do not have to make the Clipboard window active before you close it.

Copy Picture

Copy Picture makes a copy of the graphics portion of the currently active window and puts the copy on the Clipboard. Any text in the window that was printed with the PRINT command will not be copied onto the Clipboard. Even when the Clipboard contains a picture, the Clipboard window on the screen will display its contents—you will see the picture itself. Once a picture has been copied onto the Clipboard, it can be pasted into the Scrapbook desk accessory or into any Macintosh application that accepts pictures.

USING THE SEARCH MENU

Figure 4-4 shows the Search menu. The commands on this menu give you the ability to find a specified string and to replace it. The string can be a variable name, a label, a marker you left for some special reason, or any other arbitrary set of characters.

What to Find

The What to Find dialog box, shown in Figure 4-5, allows you to specify the parameters to be used by the Find, Replace, and Replace All commands. This dialog box appears whenever you select What to Find or whenever you invoke a command from the

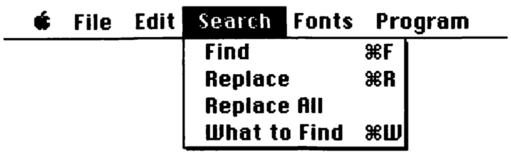
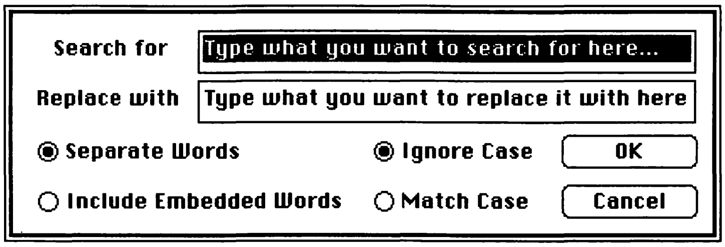


Figure 4-4. The Search menu

Search menu for which the parameters have not yet been set. Once the parameters have been set, they remain set until you change them, even if you begin editing a different program.

You enter the string you want to find on the top line. On the second line you enter the replacement string. The TAB key moves the cursor between the search and replace fields. You can enter as long a string as you wish on either line. To delete a previous entry, select it with the mouse and then press BACKSPACE or type a new string. You do not need to specify the Replace With string unless you will be using the Replace or Replace All command.

The most common search is for a whole word, such as a variable name or label. For this reason, you will usually set the Separate Words option. The search string "dim" will not be found in "dimple" if Separate Words is set, but it will be found if Include



Search for: Type what you want to search for here...

Replace with: Type what you want to replace it with here

[✓ Separate Words] / Include Embedded Words

[✓ Ignore Case] / Match Case

Figure 4-5. The What to Find dialog box

Embedded Words is set. To change the setting, just click on the circle next to the option you want.

You can require the case of the search string as well as the letters to be matched by clicking on Match Case. When Match Case is selected, the search routine looking for the string "Print" treats "PRINT" as a different string and does not stop. When Ignore Case is selected, the search routine treats the two as equal. The search parameters are normally set to ignore the difference between upper- and lowercase when determining whether a match has been found.

You should usually use Ignore Case because Macintosh BASIC ignores the case of the letters in BASIC commands, variable names, labels, and comments. The only time case really matters to BASIC is when the text is a string inside quotation marks.

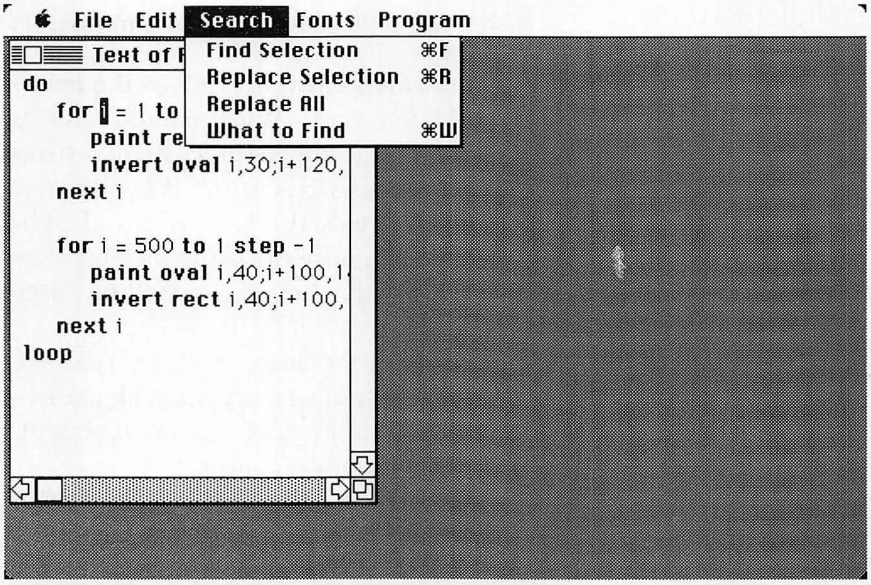
Clicking on the OK button records the selections you have made and executes the Find or Replace command if it was previously selected. Pressing the RETURN key has the same effect as clicking the OK button. Clicking on the Cancel button cancels any changes you have just made and restores the parameters to the settings they had when the box first appeared. If the box appeared as the result of a Find, Replace, or Replace All command, clicking on the Cancel button cancels that command as well.

Find

Find allows you to locate a program line or string of characters in a program. Find starts at the current insertion point in the text and stops when it finds the string for which it is searching. Find leaves the text of the target string selected and highlighted. You can specify the string to search for (the target string) in two ways: by selecting a string in the text before you invoke Find or by typing the string in the What to Find dialog box. If any text is selected, the Search menu contains the Find Selection command as shown in Figure 4-6. Find Selection locates the next match for the selected text string.

You can invoke Find from the keyboard by typing COMMAND-F. Repeatedly typing COMMAND-F is a quick way to flip through the text to check every reference to a specified string.

One very practical way to use the search capability is to develop your own system of markers for special locations in your programs.



```
do
   for i = 1 to
      paint re
      invert oval i,30;i+120,
   next i

   for i = 500 to 1 step -1
      paint oval i,40;i+100,1
      invert rect i,40;i+100,
   next i
loop
```

Figure 4-6. Search menu with text selected

Markers, for instance, can be used to mark program locations that need more work or that need to be rechecked later. With the exclamation point, you can leave a comment containing your marker on any program line. The marker can be your initials or any other combination of characters that is unlikely to be used for another purpose. Instead of keeping notes on separate pieces of paper or waiting for the Note Pad desk accessory, you can leave your notes in the program itself. Then, when you think the program is complete, you can search for your marker to find any notes about things you may have forgotten.

Replace

The Replace command replaces the target string with the replacement string that was set with the What to Find command. If any text is selected when you choose Replace, the selected text will be replaced. If no text is selected, Replace uses the target string from the What to Find dialog box.

Like Find, Replace starts at the insertion point in the text and stops after it finds the target string. You can reverse the effect of the Replace command with Undo as long as you use Undo right away. The keyboard sequence for Replace is COMMAND-R.

Replace All

Replace All searches the program in the active listing window and replaces every occurrence of the target string. Both the target string and the replacement string for Replace All must be set in the What to Find dialog box. The Replace All command presents that dialog box if either the search string or the replacement string has not been specified.

There is no automatic way to undo a Replace All operation. The Undo editing command does not work with Replace All. The only way to put the text into its previous form is to find the places that may need to be changed and examine them one by one. Because of this, Replace All should be used very carefully, especially if you have set the Include Embedded Words option. For safety's sake, it is a good idea to make an extra copy of your program before you start to use Replace All.

ERROR CHECKING

Because the largest number of errors involve mistyping or simple mistakes in the syntax for a single command, BASIC can detect most errors by examining a single program line as soon as it has been entered. Other errors, such as those involving the flow of program execution, cannot be detected until the program is run. Most error messages occur when you are entering program lines.

Turning Checking On and Off

Usually you want to be notified of an error as soon as it occurs so that you can correct the error while the line is still fresh in your mind. However, sometimes error checking after each line becomes bothersome, as it might to a touch typist with a long program to enter. In such situations, line-by-line error checking can be turned

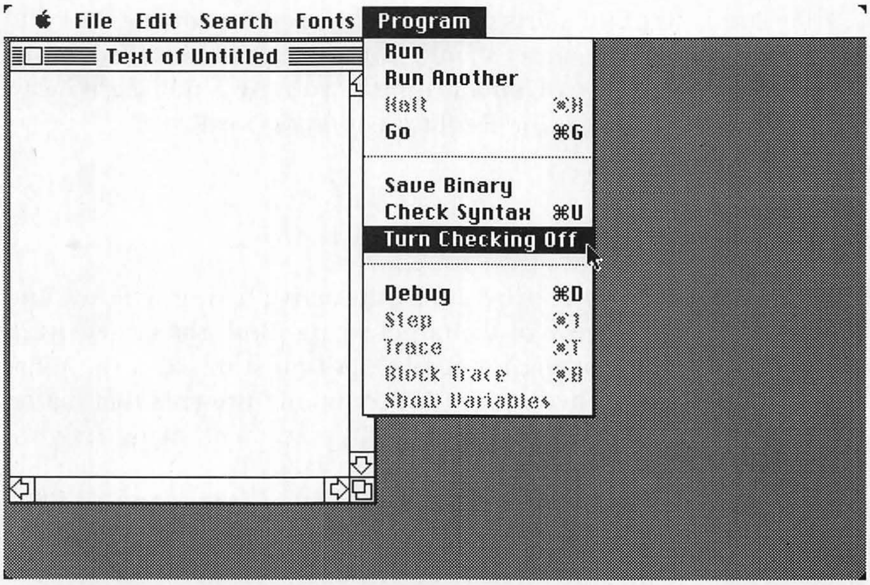


Figure 4-7. Turning checking off

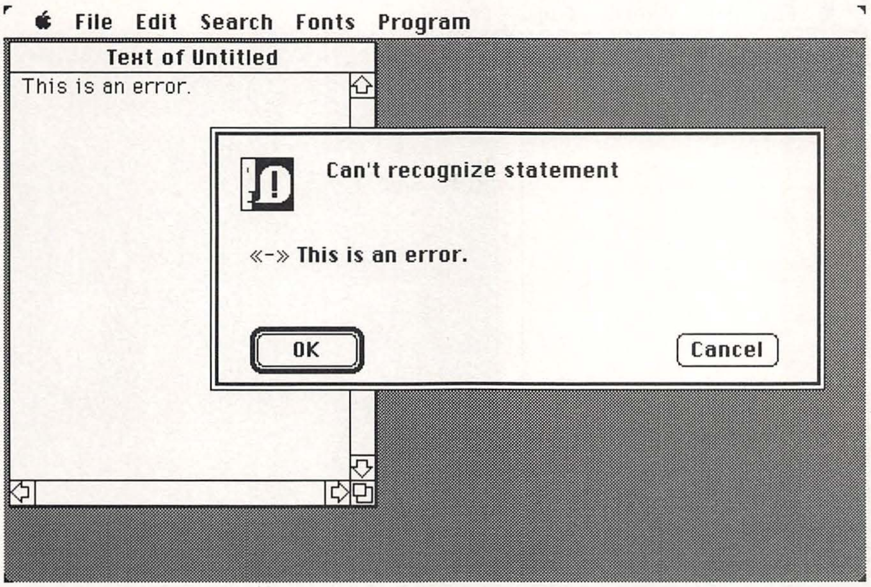
off by choosing Turn Checking Off from the Program menu, as shown in Figure 4-7. Once Turn Checking Off is selected, the entry on the menu changes to Turn Checking On to allow you to turn the line-by-line checking back on.

Selecting Turn Checking Off does not prevent the error checking from occurring; rather, the checking is deferred until you run the program. The ability to switch line-by-line error checking off and on allows you to shape the BASIC environment to match your own programming style.

Correcting Errors

Figure 4-8 shows an error message box. Most of the Macintosh BASIC error messages are reasonably clear and easy to understand.

Once an error message box appears, you have only two choices. You can choose OK, or you can choose Cancel. As is the case with most Macintosh dialog boxes, pressing the RETURN key on the keyboard has the same effect as clicking on the OK button. The



This is an error.

Can't recognize statement

«-» This is an error.

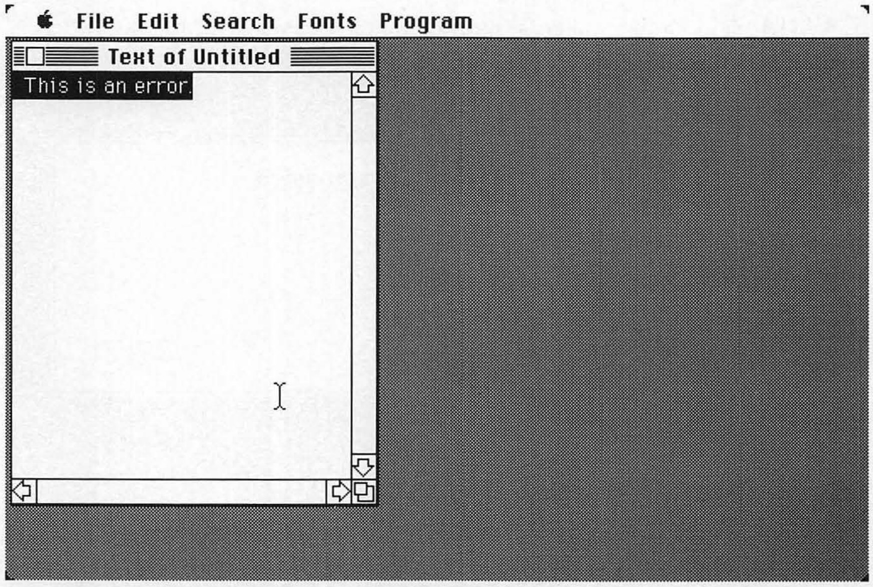
Figure 4-8. A typical error message box

Cancel button stops line-by-line error-checking of the text you just entered.

When you click the OK button, BASIC returns you to the window into which you were entering text, and the line that contained the error is selected, as shown in Figure 4-9. If you want to retype the entire line, just start typing—your first keypress will delete the selected text. If you want to edit the line, position the cursor and click to position the insertion point.

Using Check Syntax

The Check Syntax command on the Program menu performs any deferred error checking and allows you to update a running program. Check Syntax does not change the Turn Checking On/Turn Checking Off setting. You can use Check Syntax as often as you want to receive your delayed error messages without changing the program entry environment. The keyboard sequence for Check Syntax is COMMAND-U.



```
This is an error.
```

Figure 4-9. Text selected after error message

Updating a Running Program

Macintosh BASIC allows you to edit a program while it is running. This capability is especially valuable when programming graphics or sound routines.You can write a program, watch it run, and keep making changes until the program produces the graphics or sounds that you want.

When editing a program while it is running, you need to remember that there are two versions of your program in the machine: one is the program source listing you see in the text window, the other is the compiled version of your program maintained by Macintosh BASIC. The compiled version is the one that is executed when you run the program.

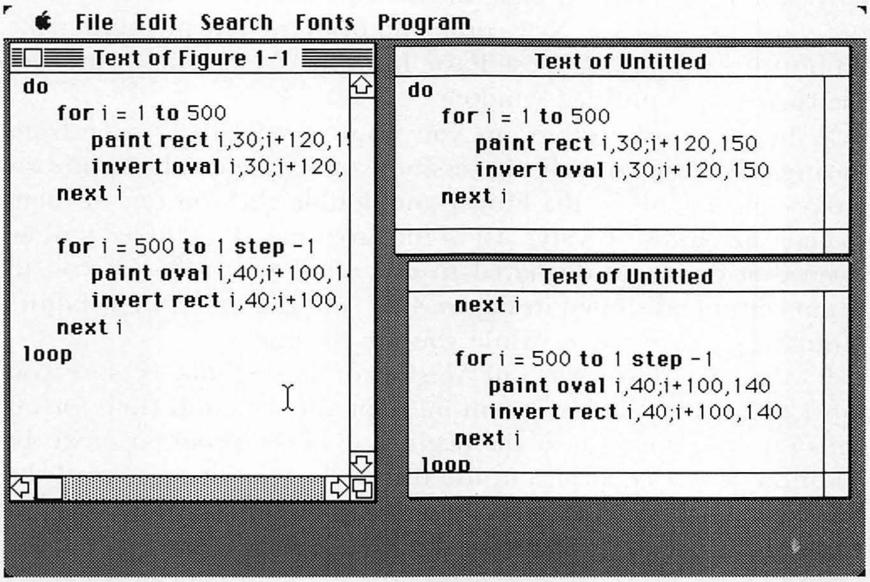
Once you make editing changes to the source program in the text window, you need to use Check Syntax to tell BASIC to update the compiled version. The update process will incorporate your changes into the running program, and you can watch your changes take effect in the output window.

PROGRAM EDITING SHORTCUTS

If you need to refer to one part of your program while you are writing another part, you could scroll to the place you need to check and then scroll back to the place where you are working. A faster way is to keep a second copy of your program in another window. Figure 4-10 shows the screen arranged with three copies of the same program.

The safe way to work with more than one copy of the same program is to have only the version of the program that you are modifying in a listing window that bears the program's name. All of the remaining copies should be in untitled windows so that you can easily tell which copy of the program you are changing. This way, when you finish, you will have only one copy of the program to save.

To make more than one copy of the same program, first load the program. Then choose Select All from the Edit menu followed by Copy. This places a copy of your entire program on the Clipboard

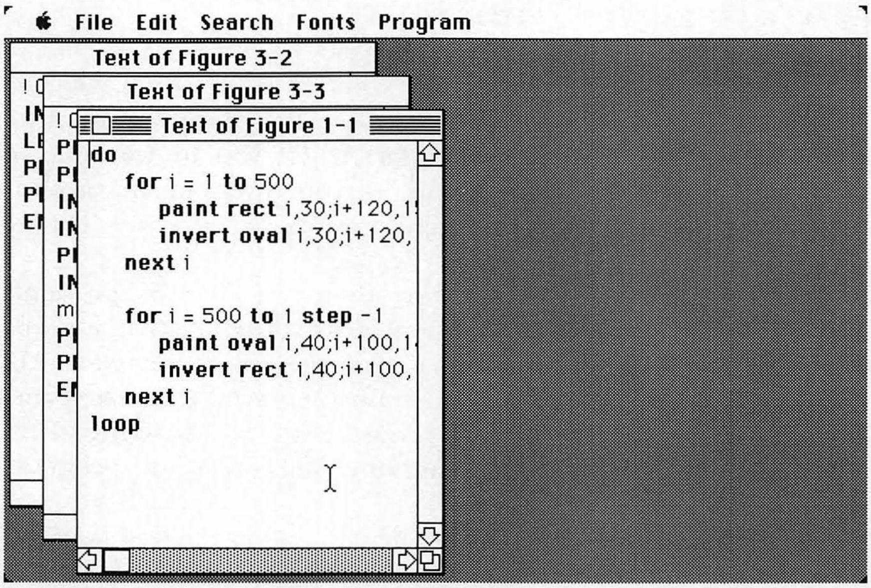


```
do
   for i = 1 to 500
      paint rect i,30;i+120,1
      invert oval i,30;i+120,
   next i

   for i = 500 to 1 step -1
      paint oval i,40;i+100,1
      invert rect i,40;i+100,
   next i
loop
```

```
do
   for i = 1 to 500
      paint rect i,30;i+120,150
      invert oval i,30;i+120,150
   next i
```

```
   next i

   for i = 500 to 1 step -1
      paint oval i,40;i+100,140
      invert rect i,40;i+100,140
   next i
loop
```

Figure 4-10. Several copies of the same program



```
do
   for i = 1 to 500
      paint rect i,30;i+120,1
      invert oval i,30;i+120,
   next i

   for i = 500 to 1 step -1
      paint oval i,40;i+100,1
      invert rect i,40;i+100,
   next i
loop
```

Figure 4-11. Several programs opened from the Finder

Click the mouse button once in the text window to deselect your program. Now use the New command from the File menu to make an untitled window. Choose Paste from the Edit menu to transfer the copy to the untitled window.

If there are other programs you want to refer to or copy from during your program editing session, you can select the appropriate program icons in the Finder and double-click on one of them to enter Macintosh BASIC. All of the programs you selected will be opened at once and displayed in text windows, ready for you to begin editing, as shown in Figure 4-11. You can move the windows around to arrange them while you are editing.

If you save more than one version of a program, be sure you label them. Put the version number or the date and time (or all three) in a comment near the beginning of the program's text. In addition, it is a good idea to use the version or date as part of the file name so you can immediately identify the most recent version without having to open up all the files.

PRACTICE EXERCISES

1. Open a new listing window and type in the Macintosh sample graphics program from Chapter 1. If you had error checking turned off while you typed, check for syntax errors. Save the text of the program on disk.
2. Open MacWrite and type a Macintosh BASIC program. When you are finished, select the Save As command from MacWrite's File menu, and click on the Text Only option in the dialog box. After you have saved the program, quit MacWrite. Now click on the program's icon to select it; then hold down the SHIFT key and click on the Macintosh BASIC icon. With the two icons selected, double-click on the Macintosh BASIC icon to load the program into a text window. Fix any errors BASIC identifies while reading and compiling the program. Which way of entering programs do you prefer?
3. Type a program in a listing window and experiment with the editing commands. Select Show Clipboard so you can watch what each command does to the contents of the Clipboard. Try Undo after each editing command to observe its effect.
4. Open the sample graphics program. Open a new, untitled window and copy the program into the new window. Close the original window. With the program in the untitled window, experiment with Search, Replace, and Replace All. Try these commands with and without the Include Embedded Words and Match Case options. When you are finished experimenting, do not save the untitled program unless you give it a name different from the original program.
5. Open the sample graphics program from exercise 1, and make a copy of it in a new window. Close the original program's window, and use the copy for the rest of this exercise. Run the copy. Use the editing techniques in this chapter to change the numbers in the running program. When you are finished experimenting, do not save the untitled program unless you give it a name different from the original program.

Chapter 5
Making Decisions

Statements:

- GOTO, IF/THEN/ELSE/ENDIF
- SELECT/CASE/CASE ELSE/END SELECT

Operators:

- $=$, $\neq$, $>$, $<$, $\geq$, $\leq$, AND, OR, NOT

A program is much more useful when it is able to react in different ways to different situations. The commands explained in this chapter allow a program to test whether a predefined condition is met and to execute a different sequence of instructions for each different condition. This is the foundation on which all interactive programs are based.

LABELS AND BRANCHING

- GOTO

The order in which the instructions in a computer program are executed is called the *flow of control*. The flow of control normally starts with the first line of a program and proceeds from one line to the next until an END statement is encountered or there are no more lines in the program.

The GOTO statement changes the flow of control every time it is executed by branching, or transferring control, to the line designated in the GOTO statement. The line to which a GOTO statement transfers control must start with either a label or a line number. Here are some examples of lines starting with labels and line numbers:

```
90 a=3  ! 90 is a line number
Newline: a=4  ! Newline is a label
880 PRINT "This line has a line number"
L33: PRINT "This line has the label 'L33'"
```

In the following program,

```
Do.it: PRINT "I love apple pie."
    GOTO Do.it
```

the GOTO statement transfers program execution to the line labeled "Do.it", which prints "I love apple pie." and then executes the GOTO statement again. This is called an *infinite loop,* because it will continue for an infinite amount of time if left alone. The program cannot get out of the loop by itself. The loop will end only if you stop the program by closing the output window or choosing the Halt command from the Program menu.

You can use either line numbers or labels, and you can mix the two in the same program. A line number or label must be the first thing (except for spaces) on a new line. Failure to follow these rules is likely to result in an error message when the line containing the incorrect line number or label is entered.

Here is a summary of the requirements for labels. A label must start with a letter and may contain letters, digits, and some special characters. The characters that cannot be used in a label are the same as the characters that cannot be used in a variable name. They are listed in Table 3-1. A label must be followed by a colon.

Rules for Labels

- Must be the first thing on a line
- Must start with a letter
- May not contain spaces or other special characters
- Must be followed by a colon (:)

A line number may contain only the digits zero through nine and must be followed by a space. Macintosh BASIC treats line numbers as specialized labels and does not require numbered lines to be in numerical order. Thus, you can move lines from one program to another or to a new location within the same program without having to change the sequence of line numbers.

Rules for Line Numbers

- Must be the first thing on a new line
- May contain only the digits 0 through 9
- Must be followed by a space
- Lines do not need to be in numerical order

A program that contains too many GOTO statements can be very hard to read and understand. In addition, erroneous use of GOTO statements can result in infinite loops and other problems that are hard to find and correct. You need to understand the GOTO command, but you should avoid using it if at all possible. Macintosh BASIC provides all the additional control structures you will need to write your programs without using GOTO. Some of these control structures are described in the next chapter.

MAKING COMPARISONS

The commands discussed in the remainder of this chapter are *conditional branches*—that is, they change the normal flow of control only if a specified condition is met. The ability to test whether or not a condition is satisfied is what makes it possible for a program to make decisions. Conditions to be tested are specified with relational and logical operators.

Relational Operators

■ =, ≠, >, <, ≥, ≤

Relational operators test the relationship between two numbers, strings, or other expressions. Table 5-1 lists all of the relational operators. These include the familiar concepts of equality and inequality and greater than or less than. You can also use the combinations greater than or equal to and less than or equal to.

The Macintosh keyboard can generate the one-character symbols for the operators "not equal to," "greater than or equal to," and "less than or equal to." These characters are produced by holding down one of the two OPTION keys while pressing the equal, comma, or period keys. Table 5-2 summarizes the special keystrokes for these three characters. As shown in Table 5-1, the "not equal to" operator can also be represented by combining the greater than and less than symbols in either order. Similarly, the compound operators "greater than or equal to" and "less than or equal to" can be represented by combining the operators for the two parts, again in either order.

When an expression containing relational operators is evaluated,

Table 5-1. Relational Operators

Symbol	Meaning
=	Equal to
≠, <>, ><	Not equal to
>	Greater than
<	Less than
≥, >=, =>	Greater than or equal to
≤, <=, =<	Less than or equal to

Table 5-2. Keystrokes for Special Characters

Character	Keystroke	Meaning
≠	OPTION =	Not equal to
≤	OPTION ,	Less than or equal to
≥	OPTION .	Greater than or equal to

the result is either true or false. The expression 4>3 evaluates as true because 4 is greater than 3, and the expression 4=3 evaluates as false because 4 is not equal to 3. The expression *a*=5 is true only when variable *a* is set to the value 5; it is false for any other value of *a*.

Relational operators can also be used to compare strings. Each character in a string is represented inside the computer by a code between 0 and 255. Appendix C contains a complete listing of these codes. Two strings can be compared by looking at them one character at a time and comparing the codes for each character. If the first parts of two strings are equal, but one string has more characters, the shorter string is determined to be less than the longer string. Thus, the expression "BERT"<"BERTRAM" is true because, while the first four characters are identical, the shorter string runs out of characters.

You should exercise caution when comparing or sorting strings that contain numbers. A string comparison will find that "33"<"9" is true, because the first character (3) of the first string is less than the first character (9) of the second string. If you need to compare numbers that are contained in strings, you should either check the lengths of the strings first or convert the strings to numbers, as will be described in Chapter 7.

Logical Operators

- AND, OR, NOT

Expressions that have a value of either true or false, like relational expressions, are called *logical expressions.* The simple logical

Table 5-3. Logical Operators and Their Effects

Operator	Operation	Result
NOT	**NOT** TRUE	FALSE
	NOT FALSE	TRUE
AND	TRUE **AND** TRUE	TRUE
	TRUE **AND** FALSE	FALSE
	FALSE **AND** TRUE	FALSE
	FALSE **AND** FALSE	FALSE
OR	TRUE **OR** TRUE	TRUE
	TRUE **OR** FALSE	TRUE
	FALSE **OR** TRUE	TRUE
	FALSE **OR** FALSE	FALSE

expressions can be combined into more complex expressions by use of special words called *logical operators.* There are only three of these in Macintosh BASIC: AND, OR, and NOT. The results of their use with the different possible combinations of logical values are summarized in Table 5-3.

The use of AND and OR in BASIC follows the rules of Boolean algebra. The logical operator AND returns a value of true only if both expressions are true. The logical operator OR returns a value of true if either of the expressions is true. The NOT operator changes true to false and false to true. If used twice, the NOT operator leaves a variable in its original state.

Order of Precedence

The three logical operators can be combined with relational and arithmetic expressions to make very complex logical expressions. Just as with the arithmetic operators, rules of precedence determine the order in which operations are performed when evaluating a complex expression. Table 5-4 lists the arithmetic, relational, and logical operators in a descending order of precedence.

The operations at the top of Table 5-4 are performed before

Table 5-4. Order of Precedence

Operator	Operation
^	Exponentiation
+ − **NOT**	Unary operators
* / **DIV MOD**	Multiplication, division, modulo
+ −	Addition, subtraction
$= \neq > < \geq \leq$	Relational operators
AND	Logical AND
OR	Logical OR

operations farther down the table. Operations on the same line of the table are performed in the order they occur in the expression being evaluated. Parentheses may be used to override the normal order of precedence for relational and logical operators, just as with the arithmetic operators.

If no parentheses are present, the first operation performed is exponentiation. After exponentiation come the *unary operators,* which operate on only one expression. The plus and minus signs do double duty: they are unary operators in expressions with only one number, like −4 and (+5), and they are arithmetic operators in expressions with two numbers, like 3−4 and 3+5. When all the unary operations are completed, the multiplications and divisions are performed followed by additions and subtractions. Evaluation of the expression then continues with the relational operators, followed by the logical ANDs, and finally the logical ORs.

Parentheses and extra white space can often make long expressions more understandable. For example, the expression $a <= 3$ OR $a > 9$ AND $b = 5$ OR $b = 10$ is much easier to read when you write it as ($a <= 3$) OR ($a > 9$ AND $b = 5$) OR ($b=10$).

ACTING ON COMPARISONS

■ IF/THEN

If the expression following the keyword IF is true, the statement following the keyword THEN is executed. If the expression is false, the statement immediately following the keyword THEN is not executed, and execution continues at the beginning of the next program statement. For example,

```
IF interest.rate ≥ .10 THEN PRINT "Too high!"
```

does not print anything if the variable *interest.rate* is less than 0.10 and prints "Too high!" if *interest.rate* is greater than or equal to 0.10.

Note that if the IF test is false and more than one statement is on the same line, Macintosh BASIC skips only the first statement after the keyword THEN. Execution of the line

```
IF a = 3 THEN b = 4: PRINT b
```

will always print the value of *b* in Macintosh BASIC, no matter whether *a* is 3 or not. The simplest way to avoid any confusion is to put only one statement on each program line.

PUTTING TWO ACTIONS IN ONE STATEMENT

■ IF/THEN/ELSE

In some situations, your program needs to take one action if a test is true and another action if the test is false. This kind of situation is best handled with an IF/THEN/ELSE statement. The statement following the keyword ELSE is executed only when the IF test is false.

If the variables *coin* and *heads* contain equal values when the statement

```
IF coin = heads THEN PRINT "Heads!" ELSE PRINT "Tails!"
```

is executed, the program prints "Heads!" and does not execute the statement after the keyword ELSE. If *coin* and *heads* are not equal, the program skips the statement after THEN and prints "Tails!". This statement is shorter and simpler than the two-statement sequence

```
IF coin = heads THEN PRINT "Heads!"
IF coin <> heads THEN PRINT "Tails!"
```

and the result is exactly the same.

Note that there can be no colons, commas, or other punctuation separating the parts of a simple IF/THEN/ELSE statement. Only one statement can appear between the keywords THEN and ELSE, and only one statement can appear after the keyword ELSE.

MULTIPLE-LINE TESTS

- IF/THEN/ELSE/ENDIF

The simple IF/THEN/ELSE statement is useful, but its limit of only one statement after each THEN and ELSE is very restrictive. The multiple-line IF/THEN/ELSE/ENDIF statement removes this restriction. The keywords IF, THEN, and ELSE are the same as before. The difference here is that the THEN and ELSE phrases can contain many statements.

The example from the previous section looks like this when it is rewritten in the form of a multiple-line statement:

```
IF coin = heads THEN
        PRINT "Heads!"
    ELSE
        PRINT "Tails!"
    ENDIF
```

The indication that this is a multiple-line IF statement is that nothing appears after the word THEN on the first line. If the test is true, Macintosh BASIC executes statements starting with the next line until it encounters either an ELSE or an ENDIF keyword. Any statements between ELSE and ENDIF are not executed. If the test is false, Macintosh BASIC skips all the statements between THEN and ELSE and executes any statements located between ELSE and ENDIF.

Figure 5-1 shows a two-dimensional diagram of the flow of program control during execution of a multiple-line IF statement. Such diagrams, called *flowcharts,* are often drawn as part of the written documentation for large programs. Figure 5-1 is two-dimensional,but a program listing has only one dimension, running from top to bottom. For this reason, programmers commonly use indentation to make the different portions of the multiple-line IF statement easy to identify. This is a good practice to follow in your programs. It becomes even more essential when your program contains a series of consecutive IF statements.

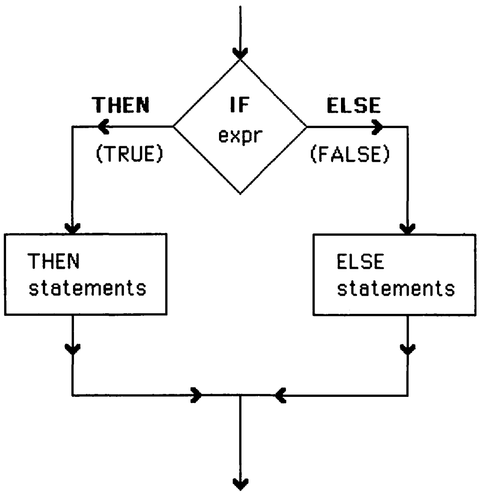


Figure 5-1. Flowchart of an IF statement

NESTING TESTS

IF statements can be nested, one inside another, as long as you follow the syntax rules. However, it is often very difficult to read a series of nested IF statements on a single line, particularly if any of the IF statements contain ELSE clauses. Nested IF statements should usually be written as multiple-line statements to improve readability. You must make sure the ELSE and ENDIF keywords match the correct IF statements. The statement

```
IF a=b THEN IF c=d THEN IF d=a THEN x=1 ELSE x=2 ELSE x=3
```

is much clearer when it is written as

```
IF a = b THEN
   IF c = d THEN
      IF d = a THEN
         x = 1
      ELSE
```

```
            x = 2
         ENDIF
      ELSE x = 3
      ENDIF
   ENDIF
```

Nested IF statements are often a desirable substitute for compound tests. Any time you have a compound test connected by the AND operator, you can replace it with nested IF statements—particularly when the second half of the test will produce nonsense or an error condition if the first half of the test is false. For example, the statement

```
IF a <> 0 AND b/a = 6 THEN d = c
```

tests whether two things are both true. If *a* is zero, the first test is false, and the compound test will always be false. In this situation it is unnecessary to have the computer spend time performing the second test. The second test involves a division by zero, an error condition that is handled gracefully by Macintosh BASIC (it returns the value *infinity*).

Using nested IF statements, the previous example can be rewritten as either

```
IF a <> 0 THEN IF b/a = 6 THEN d = c
```

or

```
IF a <> 0 THEN
   IF b/a = 6 THEN d = c
ENDIF
```

MULTIPLE TESTS IN A SINGLE COMMAND

■ SELECT/CASE/CASE ELSE/END SELECT

The IF/THEN/ELSE construct is really designed to handle situations with only one or two choices. Now we come to a statement that can handle multiple choices. In Figure 5-1 the flowchart shows that an IF/THEN/ELSE statement allows two alternate pathways from the beginning to the end of the structure. The

SELECT/CASE statement offers multiple pathways. Here is what a simple SELECT/CASE statement looks like:

```
! Days in a month
SELECT month
    CASE 1,3,5,7,8,10,12
        days = 31
    CASE 2: days = 28
    CASE 4, 6, 9, 11
        days = 30
    CASE > 12
        PRINT "You're kidding!"
    CASE ELSE
END SELECT
```

The first line of each SELECT/CASE statement contains the keyword SELECT followed by the expression that is to be evaluated to determine which path will be taken. The expression can be a single variable name or any legal BASIC expression. The optional word CASE may be added between SELECT and the expression, if you wish.

Each pathway begins with the CASE keyword followed by a description of the cases for which that pathway is to be taken. The descriptions can include actual values, called *constants* or *literal values,* but they must not use any variables or require any calculations during program execution. Each description can take the form of a single literal value, a range of values with the low and high values separated by the keyword TO, a range of values described by a relational operator followed by a literal value, or any combination of these with the individual items separated by commas. You can use the optional word IS in front of a relational operator if you wish. The following example uses all of the types of case descriptions:

```
SELECT w*8
    CASE < 0: x=1
    CASE IS < 5: x=2
    CASE 6: x=3
    CASE 7 TO 12: x=4
    CASE 13,15 TO 18,>20: x=5
    CASE 14,19: x=6
END SELECT
```

When a SELECT statement is executed, BASIC evaluates the expression after the word SELECT and then starts looking for a CASE statement that matches the value of the expression. When it finds the first matching CASE description, BASIC begins executing the instructions after that CASE statement. The instructions may begin on the next line after the CASE statement or on the same line if the colon is used at the end of the description to separate the two statements. You can use several statements for each case, as in the multiple-line IF statement, or you can leave a case empty by following it immediately with another CASE description.

Once a CASE description has been matched, execution of the statements associated with that description continues until another CASE statement or an END SELECT statement is reached. At that time, execution branches to the next statement after END SELECT. It is important to remember that the SELECT/CASE statement takes one—and only one—of the multiple paths. Even if there is a second CASE description that matches the SELECT expression, the statements associated with that second CASE description will not be executed.

CASE ELSE is an optional case description that can be included to trap all cases that are not matched by previous CASE descriptions. When it is used, CASE ELSE should always appear as the last CASE description because no descriptions after CASE ELSE will ever be reached during program execution. When a SELECT/CASE structure is executed, Macintosh BASIC displays an error message if the SELECT expression is not matched by any of the CASE descriptions. You can prevent this error message from occurring by using CASE ELSE to handle erroneous values.

EXAMPLE PROGRAM

The program in Figure 5-2 calculates the amount of paint you need to paint interior rooms. It calculates and prints the amount of paint required for each room and then prints the total amount of paint required at the end. The number of square feet of coverage per gallon, which is set in the third line of the program, can vary depending on the type of paint and the surface to be painted. An estimate of this number is usually printed on the paint can.

```
! Gallons of Paint
PRINT "Paint Estimator"
LET sqft.per.gallon = 400
total.walls = 0      ! Two variables for running totals
total.ceilings = 0

Next.room:
    ! Get Input
        PRINT
        PRINT "Please give room dimensions in feet"
        PRINT "or type 0 to quit:"
        PRINT
        INPUT "Length of room: "; length
        IF length = 0 THEN GOTO Finish:
        INPUT "Width of room: "; width
        INPUT "Height of room: "; height
    ! Do Calculations
        wall.area = 2 * length * height + 2 * width * height
        ceiling.area = length * width
        wall.paint = wall.area / sqft.per.gallon
        ceiling.paint = ceiling.area / sqft.per.gallon
        room.paint = wall.paint + ceiling.paint
        total.walls = total.walls + wall.paint
        total.ceilings = total.ceilings + ceiling.paint
    ! Display Answers
        PRINT
        PRINT "Walls require "; wall.paint; " gallons of paint."
        PRINT "Ceiling requires "; ceiling.paint; " gallons of paint."
        PRINT "Total for this room is "; room.paint; " gallons."
    GOTO Next.room

Finish:
    PRINT
    PRINT "Total wall paint, "; total.walls; " gallons."
    PRINT "Total ceiling paint, "; total.ceilings; " gallons."
    PRINT "Grand total, "; total.walls+total.ceilings; " gallons."
END PROGRAM
```

Figure 5-2. Gallons of paint

The first line of the program is a comment that contains a brief description of the program. The second line prints the title "Paint Estimator" at the top of the output window. Then a LET statement sets the variable *sqft.per.gallon* to a value of 400. The fourth and fifth lines set variables to zero to start the running totals.

The portion of the program labeled Next.room is indented slightly to set it apart from the rest of the program. This routine prints a blank line followed by the prompt to use feet as the units for the room dimensions and then another blank line. Finally, this routine requests and waits for you to enter values for the length, width, and height of the room.

As soon as the length has been entered, an IF statement tests whether its value is zero; if so, program execution branches to the line labeled Finish to print the totals.

The next five lines contain implied LET statements. They calculate the wall and ceiling area, the amount of paint required for the walls and ceiling, and the total amount of paint required to paint the room. The next two lines update the running totals. After printing a blank line to separate the answers from the input statements, the program prints the amount of paint needed to cover the walls, ceiling, and the whole room. The GOTO statement then branches back to the label Next.room. The Finish section of the program prints the running totals and ends the program.

PRACTICE EXERCISES

1. Does the following program print anything?

```
a: GOTO c
b: GOTO a
c: GOTO b
PRINT "Hello"
```

2. Evaluate the following expression to see whether *v~* is true or false:

```
v~ = 4 * 2 > 5 OR 3 ^ 2 + 3 ≤ 7 AND NOT (3 = 7 / 2)
```

3. This one-line IF statement is hard to read. Try rewriting it as a multiple-line IF statement:

```
IF a=6 THEN IF b=g THEN x=8 ELSE x=5 ELSE x=1
```

4. Rewrite the following series of IF statements as a SELECT CASE statement:

```
b = 10
IF i = 1 THEN b = 3
IF i = 2 THEN b = 5
IF i < 0 THEN b = 0
IF i = 4 OR i = 6 THEN b = 7
```

Chapter 6
Organizing Your Program

Statements:

- DO/EXIT DO/LOOP
- FOR/TO/STEP/EXIT FOR/NEXT
- GOSUB/RETURN

This chapter introduces loops and subroutines—methods of making your program take repetitive actions. *Loops* are a series of program statements that are repeatedly executed. *Subroutines* are groups of program statements you can execute from any location in your program. When a subroutine finishes executing, it returns control to the place from which your program called the subroutine.

USING LOOPS

Every loop has a definite beginning and a definite end. When a loop is encountered in a program, BASIC repeatedly executes the statements between the loop's beginning and end until an exit condition is met. When the program does exit from a loop, execution continues with the statement immediately following the end of the loop.

While the GOTO statement from the previous chapter can be used to make loops, the DO/LOOP and FOR/NEXT statements described in this chapter are much more efficient. The statements inside a DO loop are executed until a condition forces an exit from the loop. The statements inside a FOR/NEXT loop are executed for the number of times you specify when you define the loop.

Continuous Loops

■ DO/EXIT DO/LOOP

The DO/LOOP structure is the primary way to create an infinite loop in Macintosh BASIC. The program statements between DO and LOOP are executed repeatedly. The program segment

```
DO
    PRINT "This is a loop."
LOOP
```

is a complete DO/LOOP structure. The PRINT statement is executed repeatedly until you stop the program by closing the program's output window, selecting Halt from the Program menu, or selecting Quit from the File menu. (Quit will, of course, exit from Macintosh BASIC in addition to halting all programs.) The statement that initiates a continuous loop is simply DO. The statement that marks the end of the loop is LOOP.

The EXIT DO statement ends the DO loop by transferring control to the program statement after LOOP. EXIT DO is usually used in an IF statement that tests the condition at which you want to end the loop. Here is the previous loop with an EXIT DO statement added:

```
DO
    PRINT "This is a loop."
    count = count + 1
    IF count = 5 THEN EXIT DO
LOOP
END PROGRAM
```

The program prints "This is a loop" five times. The variable count is increased by one each time the loop is executed. When the IF statement finds that the *count* is equal to five, the EXIT DO statement is executed. The EXIT DO statement transfers control to the statement after the end of the loop, END PROGRAM.

BASIC allows you to use just EXIT instead of EXIT DO. As you will see later, however, there are several types of EXIT statements. To avoid confusion between EXIT DO and the other EXIT statements, you should always use the long version, EXIT DO.

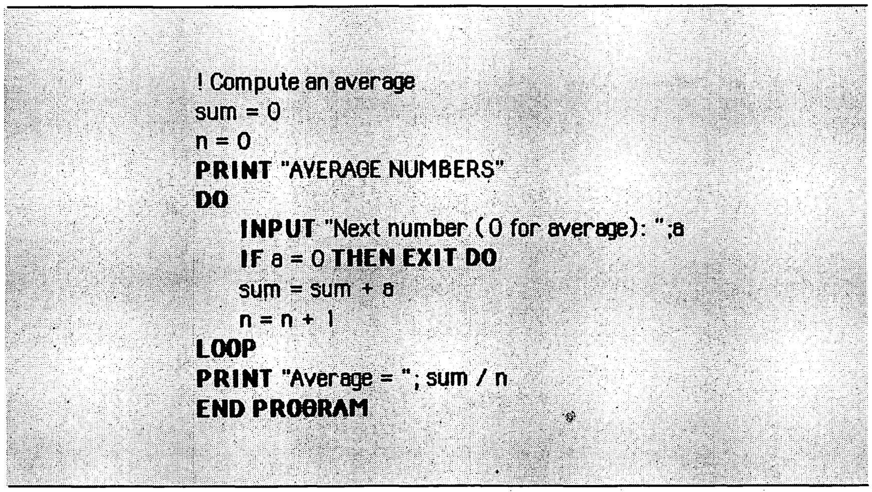
The exit condition can be anything that can be tested in an IF statement. Often it is some special input, such as a command from the person using the program. It could just as easily be the result of a calculation, an interval of time, or some other type of condition. The following loop works like an adding machine.

```
sum = 0
PRINT "ADD NUMBERS"
DO
    INPUT "Next number (0 for total): ";a
    IF a = 0 THEN EXIT DO
    sum = sum + a
LOOP
PRINT "Total = "; sum
END PROGRAM
```

This loop adds each typed number to a running total. When you type zero, the EXIT DO command transfers control to the statement after LOOP, which prints the total.

Now you can modify this short program to make it compute the average of a series of numbers. Since the average is the sum divided by the number of addends, this program must keep track of the number of addends in the sum. The program in Figure 6-1 uses a variable named *n* to store the number of addends.

Note that in this program, the location of the IF statement containing the EXIT DO is important. The test must be performed before *n* is incremented or the calculation of the average will be wrong. In general, put the exit test at the end of the loop if you want to execute the other statements in the loop at least once, and put the exit test first if you do not want to execute the entire loop at least once.



```
! Compute an average
sum = 0
n = 0
PRINT "AVERAGE NUMBERS"
DO
    INPUT "Next number (0 for average): ";a
    IF a = 0 THEN EXIT DO
    sum = sum + a
    n = n + 1
LOOP
PRINT "Average = "; sum / n
END PROGRAM
```

Figure 6-1. Computing an average

Nesting DO Loops

You can nest DO loops, as long as you remember that DO and LOOP statements must be paired with each other. You will receive an error message during program execution if a LOOP statement is encountered without a preceding DO statement. An error message also occurs if a LOOP statement is missing, but a missing LOOP statement usually causes other noticeable problems.

As with other control structures, indenting lines to mark the contents of a DO loop helps to make your program easier to read and understand, particularly if the loops are nested.

Each EXIT statement gets you out of only one DO loop. If you want to get out of an entire nest of DO loops, you must use a separate EXIT statement for each loop. The following example shows the averages program inserted inside a second DO loop:

```
DO
    sum = 0
    n = 0
    PRINT "AVERAGE NUMBERS"
    DO
```

```
        INPUT "Next number (0 for average): "; a
        IF a = 0 THEN EXIT DO
        sum = sum + a
        n = n + 1
    LOOP
    PRINT "Average = "; sum / n
LOOP
END PROGRAM
```

This program keeps computing new averages until you stop the program. The entire program is one large DO loop. Inside this loop is the averages program from Figure 6-1. The EXIT DO statement in the inner DO loop exits from only one DO loop. When you type zero, the EXIT DO statement transfers control to the statement that prints the average. Then BASIC executes the last LOOP statement, causing the entire program to repeat itself. The program does not contain an EXIT statement for the outer loop, so the program ends only when you stop it with a menu selection or by closing its output window.

Loops With Counters

- FOR/TO/STEP/EXIT FOR/NEXT

The FOR/NEXT loop is a very common control structure in BASIC programs. It is designed for situations in which you know (or can compute) how many times you want to repeat a block of program statements. Here is a FOR/NEXT loop that prints "This is a loop" five times:

```
FOR count = 1 TO 5 STEP 1
    PRINT "This is a loop."
NEXT count
```

The FOR statement specifies the name of a numeric variable (count in the previous example) that controls the number of times the loop is executed. This counter is called the loop's *index variable.* The FOR statement must also specify the index variable's starting value, its ending value, and the amount by which the index will change each time through the loop. The same variable name must be used in the matching NEXT statement that ends the loop. The amount by which the index variable will change each time through the loop is specified after the optional word STEP. If STEP is not specified, the index variable is incremented by 1 each time through the loop. Here is a FOR/NEXT loop that prints the numbers 1 through 10:

```
FOR i = 1 TO 10
   PRINT i
NEXT i
```

In this example, the FOR statement names *i* as the index variable and specifies the starting value of *i* as 1 and the ending value as 10. A STEP value of 1 is assumed. The index variable name must be a numeric variable name. The starting, ending, and step values, however, can be any legal BASIC numeric expression. When the FOR statement is encountered, Macintosh BASIC sets the index variable to the starting value. Then it compares the starting value to the ending value. If the starting value is already past the ending value (in the direction specified by STEP), control is immediately transferred to the statement after the end of the loop. Note that when this happens the statements inside the loop are not executed.

In the more normal case, the statements inside the loop are executed until the NEXT *i* statement is reached. Then the STEP amount is added to the value of *i*. The new value of *i* is compared to the prescribed ending value to determine whether the statements inside the loop are to be executed again. This process is repeated until the value of the index variable has passed the ending value. If the STEP value is negative, the index variable is decremented instead of incremented, as in the following example:

```
FOR i = 10 TO 0 STEP -1
   PRINT i
NEXT i
```

The index variable of a FOR/NEXT loop is an ordinary variable. It can be used inside the loop just like any other variable. However, any statement inside the loop that changes the value of the index variable will interfere with the operation of the loop.

Changing the value of the index variable from inside the loop is very risky and is not recommended as a good programming practice.

Once in a while you will have a situation in which you will need to provide for premature exit from a FOR/NEXT loop. The statement EXIT FOR transfers control to the statement just after the loop's NEXT statement. EXIT FOR is almost always used in an IF statement that tests for a condition that is to cause an early exit from the loop. The EXIT FOR statement in the following example is executed when i is equal to 3, transferring control to END PROGRAM.

```
j = 3
FOR i = 1 TO 5
    IF i=j THEN EXIT FOR
    PRINT i
NEXT i
END PROGRAM
```

The EXIT FOR statement is the approved way to leave a FOR/NEXT loop early. It is much safer than tampering with the index variable or using a GOTO statement. BASIC allows you to use EXIT instead of EXIT FOR, but you should always use the full wording to avoid confusion between EXIT FOR and EXIT DO.

After completion of a FOR/NEXT loop, your program should not rely upon the index variable to contain any specific value. In normal operation, the index will be one step past the ending value after a FOR/NEXT loop. This will not be the case, however, if an EXIT statement caused an early end to the loop or if the distance between the starting and ending values is not evenly divisible by the STEP value. The best programming habit is to reset the index variable to the new value if you want to use it again later in your program.

Nesting FOR/NEXT Loops

FOR/NEXT loops may be nested. One loop should be entirely contained in the other, as in this example:

```
FOR i = 1 TO 3
   FOR j = 1 TO 5
      PRINT i * j
   NEXT j
NEXT i
```

Failure to nest the loops properly is likely to lead to an error message and will interfere with your program's operation.

USING SUBROUTINES

■ GOSUB, RETURN

A *subroutine* is a block of program statements that is set up so that it can be executed from any point in your program. The way your program transfers control to the statements in the subroutine is referred to as *calling* the subroutine. Subroutines are often used to perform actions that need to be done several times during a program. They are also useful for breaking long programs into smaller, more manageable blocks of code.

The GOSUB statement calls a subroutine. You follow the command GOSUB with the name of the subroutine you want to execute. The name of the subroutine is the label or line number that marks the beginning of the subroutine in your program.

```
GOSUB help  ! Calls subroutine named 'help'
GOSUB 99  ! Calls subroutine starting at line number 99
```

When a GOSUB statement is executed, BASIC records the current location in your program and transfers control to the first statement of the subroutine. Once the statements in the subroutine are executed, a RETURN statement in the subroutine returns control to your program. Execution resumes with the statement following the GOSUB statement.

Each subroutine must begin with a label or a line number to identify it when it is called from other locations in the program. You are allowed to use line numbers, but labels make a program easier to follow. "GOSUB Get.input" conveys more meaning than "GOSUB 90." Each subroutine must end with a RETURN statement to send control back to the program that called it.

Since subroutines are blocks of code that should be executed only when called by GOSUB statements, they need to be protected from inadvertently being executed. One way to provide this protection is to put all of the subroutines at the end of a program after the END PROGRAM or END MAIN statement. That practice will be followed in the example programs in this book.

Here is an example of a main program and the subroutine it calls to handle input from the keyboard.

```
! Main program
FOR i = 1 TO 10
    GOSUB Get.input
    PRINT number
NEXT i
END PROGRAM

! Subroutine
Get.input:
    DO
    INPUT "Please type a number: "; number
    IF number > 0 AND number < 101 THEN EXIT DO
    PRINT "Sorry, it must be between 1 and 100"
    LOOP
RETURN
```

The main program in this example executes a FOR/NEXT loop that calls the subroutine Get.input. The subroutine gets a typed number between 1 and 100 and then prints the number. When the loop has been executed for the tenth time, the loop ends and the END PROGRAM statement stops execution. Without the END PROGRAM statement, the main program would not end and execution would continue into the subroutine.

The Get.input subroutine contains a DO loop that waits until a number between 1 and 100 is received. The INPUT statement receives the typed number, and the IF statement tests to see if the number is in the required range. If the number is acceptable, the EXIT DO command transfers control to the RETURN statement, which ends the subroutine and transfers control back to the PRINT statement in the main program. If the number is outside the prescribed range, the subroutine prints an error message and the LOOP statement causes the subroutine to be executed again. This type of DO loop is a good way to check for input errors.

A major benefit of using a subroutine is that the code needs to be written only once. After the subroutine is written and checked for proper operation, you can use it over and over again. You can keep "library" files containing the subroutines you use frequently. Whenever you need one of these subroutines in a new program, you can copy it from your library file into the Clipboard and from the Clipboard into your new program.

Nesting Subroutine Calls

Subroutines can be called from other subroutines in the same way they are called from main programs. Whenever a GOSUB statement is executed, Macintosh BASIC stores the location of the program statement following the GOSUB in a special place called the *stack.* The stack is what is called a "last in, first out" device. Each RETURN statement causes control to transfer to the last address placed in the stack. As long as each subroutine ends properly with a RETURN statement, control will eventually return to the program statement following the first GOSUB.

EXAMPLE PROGRAM

The program in Figure 6-2 uses a subroutine call inside nested FOR/NEXT loops to generate a multiplication table. The main program in this example consists of two FOR/NEXT loops. The outer loop, whose index is *i*, counts the rows of the multiplication table, and the inner loop with *j* counts the columns. Note that the inner loop is completed before the end of the outer loop. The END PROGRAM statement marks the end of the main program. Its main purpose is to separate the program from the subroutine.

Note that the PRINT statement inside the subroutine ends with a comma, which causes the next number to be printed on the same

```
! Make multiplication table
FOR i = 1 TO 5
    FOR j = 1 TO 5
        GOSUB Print.cell
    NEXT j
    PRINT
NEXT i
END PROGRAM
Print.cell:
    PRINT i * j,
    RETURN
```

Figure 6-2. Multiplication table

line at the next tab stop. The PRINT statement after NEXT *j* issues a carriage return, which causes the next entry to be printed at the beginning of the next row.

PRACTICE EXERCISES

1. When will the word "Hello" be printed in the following program?

```
DO
    INPUT "Enter a number: ";a
    IF a > 1 AND a < 5 THEN EXIT DO
LOOP
PRINT "Hello"
END PROGRAM
```

2. How many times will the statements inside the following loop be executed?

```
FOR i = 0 TO 100 STEP 10
    a = i
    k = 47
    PRINT i
NEXT i
```

3. Write a loop that prints all the odd numbers from 3 to 37.

4. Write a program that prints "This is a test." ten times. Use a subroutine to do the actual printing.

Chapter 7

Using Functions

Functions:

- ABS, SGN, SIGNNUM, COPYSIGN
- INT, TRUNC, RINT, ERR
- SQR, PI, SIN, COS, TAN, ATN
- LOG, EXP, LOGP1, EXPM1
- LOG2, EXP2, LOGB, SCALB
- RELATION, COMPOUND, ANNUITY
- TICKCOUNT, RND, RANDOMIZE, RANDOMX

A function is like a simple subroutine that returns a single result. All of the information needed to calculate the result is passed to the function as *arguments,* that is, values that appear in parentheses after the function's name. An argument can be any legal BASIC expression; however, the number and type (number or string) of arguments passed in a function call must match the number and type specified in the function's definition, or an error message will occur. Most functions require one or two arguments; some require no arguments at all.

Functions are very powerful parts of the BASIC language because you can use function calls as you use variables. In fact, you can even include a function call in the argument of another function call.

Macintosh BASIC provides a large selection of predefined functions. Many of them are described in this chapter. In addition to using the predefined functions, you can expand the language by defining your own functions. That is described in Chapter 11.

MANIPULATING THE SIGN OF A NUMBER

Macintosh BASIC has four functions that help you manipulate the sign of a number. These functions determine the absolute value of a number, record its sign, and copy a sign from one number to another.

Absolute Value

■ ABS

The absolute value of a number is the number with no sign. The ABS function takes one numeric argument and returns the absolute value of that number. The result of the ABS function is always a positive number. Here are some examples of the ABS function:

```
y = ABS(-3)  ! Puts +3 in y
j = ABS(3*4-14)  ! Puts +2 in j
e = ABS(4)  ! Puts +4 in e
s = ABS(0)  ! Puts 0 in s
```

Checking the Sign

■ SGN, SIGNNUM

The SGN and SIGNNUM functions help you use the sign of a number in a formula. Each function requires one numeric argument. The SGN function returns +1 if the sign of the argument is positive, −1 if the sign of the argument is negative, and 0 if the value of the argument is zero. Here are some examples of the SGN function:

```
a = SGN(99)  ! Puts +1 in a
PRINT SGN(-30)  ! Prints -1
b = SGN(0)   ! Puts 0 in b
```

The SIGNNUM function returns +1 if the sign of the argument is negative and 0 if the sign of the argument is positive or the value of the argument is zero.

```
a = SIGNNUM(99)  ! Puts 0 in a
PRINT SIGNNUM(-30)  ! Prints +1
b = SIGNNUM(0)   ! Puts 0 in b
```

The SGN function accepts a logical expression as well as a number for its argument. SGN returns +1 if the logical expression is true and 0 if the logical expression is false. The SIGNNUM function does not accept logical expressions.

```
a = SGN(TRUE)  ! Puts +1 in a
b = SGN(FALSE) ! Puts 0 in b
c = SGN(4>3)  ! Puts +1 in c
```

Copying the Sign

- COPYSIGN

The COPYSIGN function copies the sign of one numeric expression to a second numeric expression. COPYSIGN takes two arguments. The sign of the first argument is copied to the value in the second argument. COPYSIGN returns the value of the second argument with its new sign.

```
a = COPYSIGN(1,-9)  ! Puts +9 in a
b = COPYSIGN(-1,-9)  ! Puts -9 in b
c = COPYSIGN(0,-9)  ! Puts +9 in c
d = COPYSIGN(-3,9)  ! Puts -9 in d
```

ROUNDING AND TRUNCATING FRACTIONS

■ INT, TRUNC, RINT

Macintosh BASIC provides three functions that convert fractional numbers into whole numbers. The integer function, INT, returns the next integer lower than its argument if the argument contains a fractional part. The truncate function, TRUNC, returns the integer portion of the argument. The rounded integer function, RINT, rounds its argument to an integer.

The INT and TRUNC functions provide the same results when the arguments are positive numbers. However, when the argument is a negative number and contains a fractional part, the INT function returns the next integer lower than the argument, while the TRUNC function returns the next integer higher than the argument. These examples illustrate the difference:

```
a = INT(55.8)       ! Puts 55 in a
b = TRUNC(55.8)     ! Puts 55 in b
c = INT(-55.8)      ! Puts -56 in c
d = TRUNC(-55.8)    ! Puts -55 in d
```

The RINT function returns the value of its argument rounded to an integer. Unless you change the rounding direction with the SET ROUND statement described in Chapter 9, RINT follows normal rounding rules and rounds the value of its argument to the nearest integer. When the value is exactly halfway between two integers, RINT rounds to the even integer.

```
a = RINT(55.8)      ! Puts 56 in a
c = RINT(-55.8)     ! Puts -56 in c
e = RINT(55.3)      ! Puts 55 in e
f = RINT(-55.3)     ! Puts -55 in f
g = RINT(55.5)      ! Puts 56 in g
h = RINT(-55.5)     ! Puts -56 in h
```

IDENTIFYING ERRORS

■ ERR

Macintosh BASIC also has several functions that require no arguments. Instead of acting on data, these functions report on some aspect of the system's operation. Functions like these are called *system functions.*

The ERR function is a system function that returns the coded number of the most recent error that has been encountered in the program. If no error has been encountered since you first started the program running, ERR will return zero. Identifying errors enables you to take special actions, such as giving more detailed instructions. A list of error codes and their causes is included in Appendix B.

```
IF ERR = 182 THEN
    PRINT "Expected a Number"
    PRINT "Please type only numbers, not letters"
ENDIF
```

USING THE MATHEMATICAL FUNCTIONS

The mathematical functions in BASIC include square root; sine, cosine, and other trigonometric functions; and functions for handling logarithms and exponentials in complex formulas. In addition to the standard functions, Macintosh BASIC provides functions to manipulate powers of 2 and several unusual functions for logarithmic and exponential applications.

Square Root

- SQR

To obtain the square of a number, you multiply the number by itself. The square root function, SQR, works in just the opposite direction. It returns the number that when multiplied by itself results in the argument. For example, the square root of 9 is 3, and the square root of 16 is 4. If the argument of the SQR function is negative, the square root is not a real number. In this case the function returns the value NAN, which stands for Not A Number.

```
a = SQR(4)      ! Puts 2 in a
b = SQR(2)      ! Puts 1.414 in b
c = SQR(9*9)    ! Puts 9 in c
d = SQR(-100)! Puts 'NAN' in d
```

PI

■ PI

The constant pi (3.14159) is used in several common mathematical formulas, including the formulas for the area and circumference of a circle. Pi is usually represented by the Greek letter π in mathematical formulas. In Macintosh BASIC, PI is a system function that requires no arguments. Since the PI function always returns the same value, you can use it as if it were a constant. If you wish, you can use the Greek letter (OPTION-p or OPTION-SHIFT-P) instead of the word PI.

```
area = PI * radius ^ 2
circumference = 2 * PI * diameter
area = π * radius ^ 2  ! π is OPTION-p
area = Π * radius ^ 2  ! Π is OPTION-SHIFT-P
```

Trigonometric Functions

■ SIN, COS, TAN, ATN

Macintosh BASIC provides the sine, cosine, tangent, and arctangent trigonometric functions. The single numeric argument for the sine (SIN), cosine (COS), and tangent (TAN) functions is an angle, expressed in *radians*. Most people are used to working with angles in degrees instead of radians, so it will be necessary for your program to convert an angle from degrees to radians before using these functions. The formula for converting from degrees to radians is

$$\text{radians} = (\text{PI}/180) * \text{degrees}$$

The arctangent function (ATN), sometimes called the inverse tangent, takes the tangent of an angle as its argument and returns the size of the angle in radians. Here are some examples of the trigonometric functions:

```
degrees = 60
radians = (PI / 180) * degrees ! 1.047 radians
a = SIN (radians)   ! Puts .866 in a
b = COS (radians)   ! Puts .5 in b
c = TAN (radians)   ! Puts 1.732 in c
d = ATN (1.732)     ! Puts 1.047 in d
```

Logarithmic Calculations

■ LOG, EXP, LOGP1, EXPM1

In addition to the standard natural logarithm (LOG) and exponential (EXP) functions, Macintosh BASIC provides the logarithm of the argument plus one (LOGP1) and exponential minus one (EXPM1) functions. Natural logarithms and exponentials are to the base *e*, where *e* equals 2.718281828. If the LOG or LOGP1 function is used to take the logarithm of a negative number, the function will return NAN (Not A Number). LOGP1(x) and EXPM1(y) return more accurate results than LOG(x+1) and EXP(y)−1, respectively, when the values of x and EXP(y) are close to zero.

```
a = LOG(37)     ! Puts 3.61 in a
b = LOGP1(36)! Puts 3.61 in b
c = EXP(3.61)  ! Puts 37 in c
d = EXPM1(3.61)  ! Puts 36 in d
```

Powers of Two

■ LOG2, EXP2, LOGB, SCALB

The LOG2 function returns the base 2 logarithm of its argument. The base 2 logarithm is the power to which the number 2 should be raised to produce a result equal to the argument. The EXP2 function returns the value 2 raised to the power specified in the function's argument. Both functions require a numeric expression as an argument.

```
a = LOG2(8)    ! Puts 3 in a
b = EXP2(3)    ! Puts 8 in b
```

The LOGB function returns the exponent of the largest power of 2 that does not exceed the magnitude of its argument. The argument must be a numeric expression. The SCALB function requires two arguments, an integer expression and a numeric expression. SCALB returns the value of the second argument (the numeric expression) multiplied by 2 to the power of the first argument (the integer expression). Here are some examples:

```
a = LOGB(9)  ! Puts 3 in a
b = SCALB(3,7)  ! Puts 56 in b
```

Table 7-1. Results of the RELATION Function

Ordering Relation	Value	Constant
arg1 > arg2	0	GREATERTHAN
arg1 < arg2	1	LESSTHAN
arg1 = arg2	2	EQUALTO
arg1 and/or arg2 = NAN	3	UNORDERED

USING ORDERING RELATIONS IN CALCULATIONS

■ RELATION

The RELATION function provides a way to let your program take multiple branches depending on whether one number is greater than, less than, or equal to a second number. The function takes two arguments, both of which are numeric expressions. RELATION returns an integer that corresponds to the ordering relationship between the two arguments, as shown in Table 7-1.

If the value of the first argument is greater than the value of the second argument, RELATION returns the number 0. If the first argument is less than the second argument, RELATION returns the number 1. The function returns the number 2 if the two arguments are equal in value. RELATION returns the number 3 if one or both of the arguments has the value NAN (NAN usually results from trying an impossible operation such as taking the square root of a negative number).

BASIC recognizes the names in the rightmost column of Table 7-1 as constants. You can use the names instead of the numbers that RELATION returns to make your program more understandable. Here are some examples using RELATION:

```
a = RELATION( 4,5)  ! Puts 1 in a
SELECT CASE RELATION(a,b)
    CASE LESSTHAN: PRINT "Less"
    CASE GREATERTHAN: PRINT "Greater"
    CASE EQUALTO: PRINT "They're equal"
END SELECT
```

MAKING FINANCIAL CALCULATIONS

Macintosh BASIC has two functions that handle financial calculations relating to compound interest, annuities, and loans. These functions allow you to make common financial calculations by simply using the function name followed by a list of arguments in parentheses.

Compound Interest

■ COMPOUND

The COMPOUND function calculates compound interest. The function takes two numeric arguments. The first argument is the interest rate for each time period, and the second argument is the number of time periods over which interest is to be compounded.

The interest rate should be expressed as a fraction. If the rate is 11%, for instance, it should appear as 0.11 in the function's first argument. The result of the COMPOUND function can be multiplied by the principal to calculate the total value of the principal and compound interest at the end of the stated number of periods. Here are sample calls to COMPOUND:

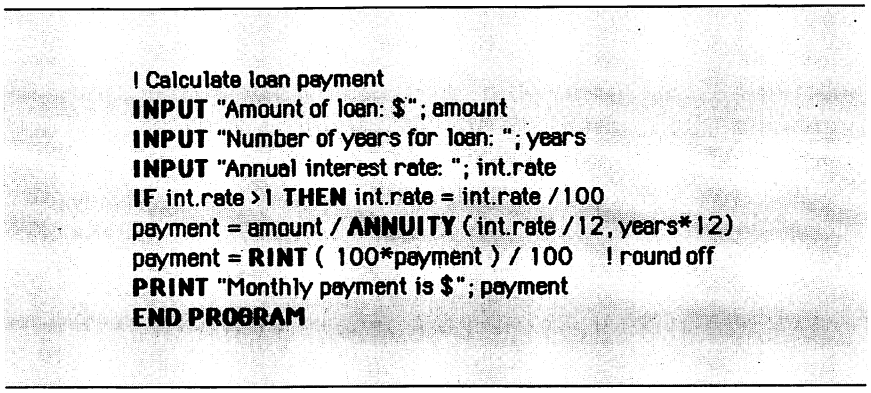
```
value = 2000 * COMPOUND ( .11,3)  ! Puts 2735.26 in value
value = 2000 * COMPOUND ( .11/12,3*12)  ! Puts 2777.76 in value
```

In the first example, COMPOUND calculates the value of three years of compound interest at an 11% annual interest rate. The value of the hypothetical investment of $2000 for three years at this rate is calculated as $2735.26 at the end of the three years. The second example uses the same dollar amount and annual interest rate, but the interest rate and the number of periods are modified to calculate the interest compounded monthly. With monthly compounding, the original investment is worth $2777.76 at the end of three years. How much more would it be worth if the interest were compounded daily?

Loans and Annuities

■ ANNUITY

The ANNUITY function can be used to calculate annuities, loans, and mortgage payments. This function also takes two arguments,



```
! Calculate loan payment
INPUT "Amount of loan: $"; amount
INPUT "Number of years for loan: "; years
INPUT "Annual interest rate: "; int.rate
IF int.rate >1 THEN int.rate = int.rate / 100
payment = amount / ANNUITY ( int.rate / 12, years* 12)
payment = RINT ( 100*payment ) / 100     ! round off
PRINT "Monthly payment is $"; payment
END PROGRAM
```

Figure 7-1. Calculate loan payment

the interest rate for each period and the number of periods. ANNUITY returns a number that, when divided into the amount of a loan or the capital amount of an annuity, gives the size of each payment. You can use it to calculate payments on a home mortgage or car loan that uses the normal compound interest rate formula. The program in Figure 7-1 uses the ANNUITY function to calculate monthly payments on a loan.

WATCHING TIME

■ TICKCOUNT

The TICKCOUNT function is used to measure small time intervals. It is a system function and requires no argument. The TICKCOUNT function returns a positive number. TICKCOUNT starts at zero when you switch on the Macintosh or use the programmer's switch to restart it. The value of TICKCOUNT increases by one every 1/60 of a second.

When you use TICKCOUNT, you need to remember that it is constantly changing at a rate of 60 times per second. This means that two successive references to TICKCOUNT, even in the same program statement, may return different answers. If you need to use the result of TICKCOUNT in several places, store it in a variable instead of calling TICKCOUNT more than once.

```
! Time something
time1 = TICKCOUNT
    ! Here I put whatever I'm timing.
time2 = TICKCOUNT
seconds.elapsed = (time2 - time1) / 60
PRINT seconds.elapsed
```

GENERATING RANDOM NUMBERS

■ RND, RANDOMIZE, RANDOMX

Random numbers can be used to create simulated data sets for complicated computer simulations or just to inject an element of chance into an activity like dealing a deck of cards. BASIC provides ways to generate both repeatable and non-repeatable sets of random numbers.

The RND function accepts one optional numeric argument. If you use it without an argument, RND returns a random number between 0 and 1. If you provide the optional argument, RND returns a random number between zero and that argument. The number RND returns will always have the same sign as the argument. If the argument is zero, RND returns zero. The number RND returns is a real number. If you want an integer, you will have to round it or truncate it.

Like almost all computerized random number generators, the RND function returns numbers that are not entirely random. They are random in the sense that any number in the requested range has an equal probability of occurring if enough random numbers are generated. However, the random numbers are calculated from a starting number, which is always the same every time a BASIC program starts to execute. This means that the same series of random numbers will be generated each time you run your program.

The RANDOMIZE statement calculates a new starting number for RND from the value of TICKCOUNT. Since the value of TICKCOUNT changes sixty times a second, you are not likely to see the same series of random numbers very frequently after using RANDOMIZE. If you do not want your program to generate the same series of random numbers each time it runs, you should use the RANDOMIZE statement to pick a random starting point.

```
RANDOMIZE       ! sets random seed
a = RND (9)     ! number between 0 and 9
a = RND (9) + 1     ! number between 1 and 10
a = INT( RND(9) ) + 1      ! integer between 1 and 10
a = RND         ! number between 0 and 1
a = RND (-4)    ! number between -4 and 0
```

The RANDOMX function is another random number generator. RANDOMX generates random integers ranging from 1 to $2^{31}-2$. RANDOMX requires one argument, which must be the name of a numeric variable (the variable must be double-precision, extended, or computational, as described in Chapter 9). The RANDOMX function puts its result in the numeric variable in addition to returning the result in the normal way.

```
x = 9
a = RANDOMX(x)  ! Puts new value in both a and x
```

EXAMPLE PROGRAM

The example program in Figure 7-2 uses random numbers as the basis of a simple guessing game. Your object in playing the game is to guess the random number from 1 to 10 that the program has generated. Each game consists of 25 guesses.

This program starts with a RANDOMIZE statement to initialize the random number generator. The rest of the program is a single DO loop that keeps playing new games until you stop the program by closing its output window or selecting the Halt or Quit command from the menu. After printing a blank line and setting the score to zero, a FOR/NEXT loop runs the game for 25 guesses.

The call to the RND function gets a number between 0 and 10. The INT function turns that number into an integer, and 1 is added to it. Note that one function call can be used as the argument for another. This property of functions allows efficient and concise program coding.

At this point in the program there is a very slight chance that the number might be 11. This happens only if the number returned by the RND function is exactly 10. The DO loop causes the number to be recalculated if it is in fact 11.

```
! Random Number Guessing Game
! Try to match the computer
RANDOMIZE
DO
   PRINT
   score = 0    ! start game with no score
   FOR try = 1 TO 25      ! 25 plays per game

      DO   ! get integer from 1 to 10
         number = 1 + INT (RND( 10))
         IF number <> 11 THEN EXIT DO
      LOOP

      DO   ! get guess from 1 to 10
         INPUT "Type number from 1 to 10: "; guess
         IF guess >= 1 AND guess <= 10 THEN EXIT DO
      LOOP

      IF guess = number THEN
         PRINT "You guessed it!!!"
         score = score + 1
        ELSE
         PRINT "Sorry, my number was "; number
      ENDIF

   NEXT try
   PRINT
   PRINT "You guessed "; score; " out of 25."
LOOP   ! Play another game

END PROGRAM
```

Figure 7-2. Random Number Guessing Game

The second DO loop receives the input from the keyboard and repeats the request for a guess if the number typed is not within the required range. A multi-line IF/THEN/ELSE statement is then used to test whether the answer is right or wrong and gives the appropriate response.

PRACTICE EXERCISES

1. What is the value of each of these expressions?
   a. ABS(3−8)
   b. SGN(−9) * ABS(−12)
2. What is the value of each of these expressions?
   a. INT(3.7)
   b. TRUNC(3.7)
   c. RINT(3.7)
   d. INT(−3.7)
   e. TRUNC(−3.7)
   f. RINT(−3.7)
3. Assume you want to borrow $5000 for 4 years. Can you write a program to print a table of your monthly payments at various interest rates? Print the values for interest rates from 9% to 12%, using increments of one half of a percent.
4. To get the performance you need from a program, it is sometimes desirable to have the program wait for a specified time before taking the next action. Can you write a loop that waits exactly one and a half seconds?
5. Write a subroutine that returns a random integer between −5 and −10.

Chapter 8

Manipulating Strings and Text

Commands:
- LINE INPUT
- OPTION COLLATE STANDARD,
  OPTION COLLATE NATIVE

Operator:
- &

Functions:
- LEN, LEFT$, RIGHT$, MID$
- VAL, STR$, ASC, CHR$
- KBD, INKEY$, DATE$, TIME$
- UPSHIFT$, DOWNSHIFT$

String variables can contain any sequence of characters. They most often contain letters, words, sentences, or other pieces of text. This chapter introduces commands and functions that allow your program to manipulate strings and the text they contain.

WORKING WITH STRINGS

Much of the work that you will be doing with strings will involve locating and replacing certain portions of a string. To allow you to accomplish this easily, Macintosh BASIC contains three functions (LEFT$, MID$, RIGHT$). You can use them to extract portions of a string as well as to locate and replace a particular sequence of characters within a string.

The ability to find the length of a string and an operator to allow you to add (concatenate) one string to the end of another are also essential in allowing you to easily manipulate text stored in strings.

Checking the Length of a String

■ LEN

The length of a string is the number of characters it contains, whether those characters are visible or not when printed. LEN is a numeric function that returns the length of its string argument. The LEN function is often used to calculate arguments for other string-related functions. A string that contains no characters is called an *empty string* or a *null string*. Since a null string contains no characters, its length is zero.

```
lgth = LEN ('test string')   ! Puts 11 in lgth
lgth = LEN ("Rah! Rah!")     ! Puts 9 in lgth
lgth = LEN ('32.9')          ! Puts 4 in lgth
a$ = 'testing'
lgth = LEN (a$)              ! Puts 7 in lgth
```

Selecting Part of a String

■ LEFT$, RIGHT$, MID$

The three functions LEFT$, RIGHT$, and MID$ return part of a string. LEFT$ and RIGHT$ each require two arguments. The first argument is the string from which the part is to be taken, and the second argument is the number of characters to be taken. LEFT$ returns the number of characters specified from the left end of the string, and RIGHT$ returns the number of characters specified from the right end of the string.

```
b$ = LEFT$ ('test string', 3)        ! Puts 'tes' in b$
b$ = RIGHT$ ('test string', 3)       ! Puts 'ing' in b$
a$ = '32.9'
b$ = LEFT$ (a$, LEN(a$)-1)           ! Puts '32.' in b$
b$ = RIGHT$ (a$, LEN(a$)-1)          ! Puts '2.9' in b$
```

The MID$ function returns any part of a string and can accept either two or three arguments. The first argument is the string from which the part is to be taken. The second argument specifies the character position from which the return string will be taken. The third argument, which is optional, is the number of characters to be taken. If you do not include the third argument, the MID$ function returns all the characters from the position specified by the second argument to the end of the string.

```
a$ = MID$('test',2,1)  ! Puts 'e' in a$
a$ = MID$('test',2,2)  ! Puts 'es' in a$
a$ = MID$('test',2)    ! Puts 'est' in a$
```

The LEFT$, RIGHT$, and MID$ functions expect their numeric arguments to be integers. Any fractional number passed as an argument is rounded to an integer. If the number of characters to be taken exceeds the number of characters available in the string argument, these functions return the remaining characters without adding any extra blanks. If the number of characters to be taken is zero, the null string will be returned. If the starting position you give to the MD$ function is 0 or a negative number, it starts with the first character instead.

Finding One String Inside Another

Often you need to know whether or not one string contains a specific character or another string. This example shows how to use the MID$ function to locate a decimal point:

```
a$ = "."
b$ = "This is a test."
PRINT b$
FOR i = 1 TO LEN(b$)
```

```
        IF MID$(b$,i,1) = a$ THEN
            PRINT """; a$; """ found at position "; i
        ENDIF
    NEXT i
```

The FOR/NEXT loop points the index variable *i* successively at each character of *b$*, the string being searched, from the first character to the last. The IF statement inside the loop takes the single character from string *b$* that is pointed to by the index variable and compares it to a period, the character being sought. If you were looking for a string longer than one character, you would replace the third MID$ parameter of 1 with the length of the string you were seeking.

Here is an example of a more general string search that will search for a string of any length:

```
    a$ = 'find'
    b$ = 'Will it find the string?'
    PRINT b$
    FOR i = 1 TO LEN(b$)+1-LEN(a$)
        IF MID$(b$, i, LEN(a$)) = a$ THEN
            PRINT """; a$; """ found at position "; i
        ENDIF
    NEXT i
```

This example differs in two important ways from the previous example. First, the third argument of the MID$ function is now set at the length of string *a$*. The part of string *b$* being compared to *a$* must equal the length of *a$*, or no match will be found. Second, the end of the FOR/NEXT loop is set at LEN(b$)+1−LEN(a$). The loop could go all the way to LEN(b$), but no match for *a$* could possibly be found at the end of that loop because the strings being extracted by the MID$ function would all be shorter than *a$*.

Adding One String to Another

- &

The operator that adds one string to the end of another is called the *concatenation operator.* Its symbol in Macintosh BASIC is the ampersand. The string following the concatenation operator is added to the end of the string before the concatenation operator. Thus, if *a$* contains the value "concat" and *b$* contains the value "enation," the statement

```
c$ = a$ & b$
```

puts the string "concatenation" in *c$*.

Replacing Part of a String

Sometimes finding one string inside another is not enough. Once you have found a string, you may need to replace it. To replace a substring, you need to handle three different pieces of the string: the beginning, the substring being replaced, and the end. This example replaces the word "true" with the word "fair":

```
a$ = 'This is a true test.'
PRINT a$
a$ = LEFT$(a$,10) & 'fair' & RIGHT$(a$,6)
PRINT a$
```

The LEFT$ function returns the first 10 characters, and the RIGHT$ function returns the last 6 characters. Both of these functions work on the original contents of *a$* because the implied LET statement evaluates the entire expression on the right of the equal sign before storing the result in *a$*.

This example demonstrates a more general way to replace a portion of a string:

```
a$ = 'find'   ! string to find and replace
c$ = 'eat'    ! the replacement
b$ = 'Will it find the string?'
PRINT b$
FOR i = 1 TO LEN(b$)+1-LEN(a$)
    IF MID$(b$, i, LEN(a$)) = a$ THEN
        b$ = LEFT$(b$, i-1) & c$ & MID$(b$, i+LEN(a$))
        EXIT FOR
    ENDIF
NEXT i
PRINT b$
```

In this example, the number of characters returned by the LEFT$ function is not specified directly but is calculated as one less than the position where the first character of the target string was found. The MID$ function is used instead of the RIGHT$ function to get the right end of the original string, because it is simpler in this case to calculate the starting location of the desired substring than it would be to calculate the number of characters desired. Only two arguments are passed to the MID$ function, so it returns everything from its starting position to the end of the string. The EXIT FOR statement stops execution of the FOR/NEXT loop after the replacement is completed.

CONVERTING BETWEEN STRINGS AND NUMBERS

■ VAL, STR$

The VAL function converts a string into a number, and the STR$ function converts a number into a string. VAL stops evaluating the number contained in its string argument if it encounters a non-numeric character. If the first character is non-numeric, VAL returns zero. Here are several examples:

```
a = VAL ('56.9')         ! Puts 56.9 in a
a = VAL ('43 years')     ! Puts 43 in a
b$ = STR$ (21.8)         ! Puts '21.8' in b$
b$ = '34.7'
a = VAL (b$)             ! Puts 34.7 in a
```

USING ASCII CHARACTERS

■ ASC, CHR$

Characters are stored inside the computer using numeric codes from 0 to 255. The coding system is called the American Standard Code for Information Interchange, abbreviated ASCII. The ASC function takes one string argument and returns the ASCII code for the first character of that argument. If the argument is the empty or null string, the ASC function returns −1.

The CHR$ function accepts one numeric argument, an ASCII code from 0 to 255, and returns the corresponding character. If the numeric argument contains a fraction, it will be rounded to the nearest integer. If it is less than 0 or greater than 255, an error message is presented. The CHR$ function is most often used in programs to introduce a character that cannot be typed from the keyboard. Here are some examples of ASC and CHR$:

```
a = ASC ('A')    ! Puts 65 in a
a = ASC ("a")    ! Puts 97 in a
a = ASC ('awesome')     ! Puts 97 in a
a = ASC ("")          ! Puts -1 in a
a$ = CHR$ (65)    ! Puts 'A' in a$
a$ = CHR$ (97)    ! Puts 'a' in a$
```

Appendix C contains a list of all the ASCII codes and their corresponding characters.

On the Macintosh, each type font includes a separate character set that can be printed. This makes unusual characters possible. In fact, each Macintosh font and size prints a different pictorial character for ASCII value 217 (OPTION-SHIFT-~). The program in Figure 8-1 prints a reference chart for the type font currently in effect, so you can identify any unusual or non-standard characters in the font. Select the font and size you want from the Fonts menu when the program pauses, and then press RETURN to display the font's characters.

```
! Print ASCII numbers and characters
! Use to display the characters in a font
PRINT "Select font and size."
INPUT "Then press RETURN to start."; a$
FOR i = 0 TO 255
    PRINT i, CHR$ (i)
NEXT i
```

Figure 8-1. Print ASCII numbers and characters

READING KEYBOARD CHARACTERS

Characters that you type at the keyboard can be received by your program one character at a time using the string function INKEY$. If you need to know the ASCII value of a typed character, you can use the KBD function.

You will often need to read an entire line of keyboard input, that is, a line of characters followed by a carriage return character. In this case, you can use the LINE INPUT statement. LINE INPUT allows you to read punctuation marks like the comma and quotation marks, which you would not be able to read with a standard INPUT statement.

Getting Single Characters

- KBD, INKEY$

The KBD and INKEY$ functions report on keyboard activity in different ways. KBD returns the ASCII value of the most recently typed character. INKEY$ returns the next character typed from the keyboard. Neither function requires an argument.

The KBD function is initialized to zero when your program first starts running and thereafter returns the ASCII value of the last character typed. KBD keeps returning the same value until another character is typed, no matter how many times the KBD function is called. The KBD function does not respond to the modifier keys (COMMAND, OPTION, SHIFT, CAPS LOCK) unless another key is pressed at the same time.

The INKEY$ function returns the next available character from the keyboard. If no character is available, INKEY$ returns the null character ("") instead of waiting. INKEY$ does not print the typed character.

INKEY$ actually gets its characters from a temporary storage area called the *keyboard buffer.* BASIC maintains the keyboard buffer to make certain that no characters are lost when you type very fast. As soon as you press a key on the keyboard, BASIC puts the corresponding ASCII value into the keyboard buffer. The buffer stores 29 characters in the order they were typed. When BASIC needs a character from the keyboard, it takes the character that was typed first from the keyboard buffer. If you manage to type fast enough to fill the keyboard buffer, BASIC discards the first character that was placed in the keyboard buffer to make room for each new character you type.

When your program calls INKEY$, INKEY$ gets the next character from the keyboard buffer. If the buffer is empty, INKEY$ returns the null character. You can observe the differences between KBD and INKEY$ for yourself by running this short program:

```
DO
   PRINT CHR$(KBD), INKEY$
LOOP
```

Getting a Whole Line of Input

■ LINE INPUT

The INPUT command described in Chapter 3 places the values you type at the keyboard into variables whose names are listed in the INPUT statement. In order to accept several values from one line of input, the INPUT command interprets commas and quotation marks as delimiters separating one value from another. If you are typing strings that contain commas or quotation marks, however, you want BASIC to treat the entire line as a single string value.

The LINE INPUT command treats everything you type on an input line as a single value, even if your typing contains commas and quotation marks. Each LINE INPUT statement places a value in only one variable. You can use a prompt string in the LINE INPUT statement just as you can in an INPUT statement. A comma after the prompt string moves the insertion point to the next BASIC tab stop, and a semicolon leaves the insertion point at the end of your prompt string. The variable used to receive the input should, of course, be a string variable. The statement

```
LINE INPUT a$
```

puts whatever is typed into the variable a$.

You will receive an error message if you type a non-numeric character when the INPUT statement contains a numeric variable. Here is an example that uses LINE INPUT instead to check for an incorrect character when your program is expecting a number from the keyboard:

```
DO
LINE INPUT "Please type a number: "; a$
number = VAL (a$)
IF STR$(number) = a$ THEN EXIT DO
PRINT "Type numeric characters only"
LOOP
```

This block of code uses a LINE INPUT statement to get a typed line in a$ and then uses the VAL function to convert that string to a number. The STR$ function converts the number back to a string and compares that string with the original. If they are equal, the typed input was a number. If the typed line included any non-numeric characters, the two strings will not be equal, the loop will continue, and the program will ask you to retype the number.

UPPER- AND LOWERCASE

■ UPSHIFT$, DOWNSHIFT$

The UPSHIFT$ and DOWNSHIFT$ functions each take one string argument. UPSHIFT$ returns the argument with every character in uppercase letters, and DOWNSHIFT$ returns the argument with every character in lowercase letters. Here are some examples:

```
a$ = UPSHIFT$ ('Yes')          ! Puts 'YES' in a$
a$ = DOWNSHIFT$ ('Yes')        ! Puts 'yes' in a$
name$ = 'rICHARD'
a$ = UPSHIFT$(LEFT$(name$,1)) & DOWNSHIFT$(MID$(name$,2))
                               ! Puts 'Richard' in a$
```

These functions are handy for formatting output from strings. They also save program code when checking input. The fact that lowercase and uppercase letters are not equal leads to complications when strings are being compared. A simple task such as checking to see whether the word "yes" was typed could be very complicated without these functions, because the program would have to look for all eight combinations of upper- and lowercase letters: YES, YEs, YeS, Yes, yES, yEs, yeS, and yes. By using the UPSHIFT$ or DOWNSHIFT$ function, you can check all of these possibilities at once:

```
IF UPSHIFT$ (answer$) = 'YES' THEN PRINT "Yes, OK."
```

CHANGING THE STRING SORT ORDER

■ OPTION COLLATE STANDARD, OPTION COLLATE NATIVE

Macintosh BASIC allows you to change the rules that govern comparisons between strings. Standard comparisons between strings compare the ASCII codes of the characters in the two strings. But comparing ASCII codes is not very useful when alphabetizing strings that contain both capital and lowercase letters. The ASCII codes for all of the capital letters are less than the ASCII codes for the lowercase letters; thus, in a standard string comparison, "Z" is less than "a".

The OPTION COLLATE NATIVE statement tells Macintosh BASIC to make its string comparisons using normal alphabetical order. After BASIC executes the OPTION COLLATE NATIVE statement, it uses alphabetical ordering until you execute an OPTION COLLATE STANDARD statement.

```
a~ = "baby" < "Jane"  ! Puts FALSE in a~
OPTION COLLATE NATIVE
b~ = "baby" < "Jane"  ! Puts TRUE in b~
OPTION COLLATE STANDARD
c~ = "baby" < "Jane"  ! Puts FALSE in c~
```

HANDLING THE DATE AND TIME

The Macintosh automatically maintains the date and time of day. The DATE$ and TIME$ functions gives Macintosh BASIC programs access to this information. Neither of these functions uses an argument, and both return strings. If the date or time returned by these functions is incorrect, you should use the Alarm Clock or Control Panel desk accessory to reset it.

If you are going to manipulate the DATE$ or TIME$ strings in your program, you should store the returned string in a variable of your own and manipulate that variable instead of repeatedly calling the DATE$ or TIME$ function during a calculation. This will avoid the possibility of errors caused by the date or time changing during the calculation.

Date

- DATE$

The DATE$ function returns a string representing the current date. The format of the string varies according to the country setting in the System file on your start-up disk. Table 8-1 shows some of the formats used to portray the date in different countries.

In the United States, the date takes the form of the month, day, and last two digits of the year separated by slashes. The month and day can each contain one or two digits. This variability in length makes it more difficult to extract the three individual parts of the date if you need them in your program. Here is one way to extract them from the DATE$ string:

```
! Get month, day, and year
da$ = DATE$          ! copy so it can't change
month = VAL (da$)
year = 1900 + VAL (RIGHT$(da$,2))
dd$ = RIGHT$(da$,5)   ! dd/yy
IF LEFT$(dd$,1) = '/' THEN dd$ = RIGHT$(dd$,4)  ! d/yy
day = VAL (dd$)
```

Table 8-1. Formats for DATE$

Country	January 23, 1985
United States	1/23/85
France	23.1.85
Germany	23.1.1985
Great Britain	23/01/1985
Italy	23-01-1985

Time

■ TIME$

The TIME$ function returns a string representing the current time. The TIME$ string changes once each second. For measuring more precisely, use the TICKCOUNT function described in Chapter 7. Like the date, the format of the TIME$ string also varies according to the setting in the System file on your start-up disk. Table 8-2 shows some of the different formats used to portray the time in different countries.

In the United States, the time takes the format of the hour, minute, and second, separated by colons, and followed by a space and either AM or PM. The minute and second always contain two digits, but the hour may be either one or two digits. As with the DATE$ function, this variability in length makes it harder to extract the three individual parts of the time if you need them in your program. Here is one way to extract them from the TIME$ string:

```
! Get hour, minute, and second
ti$ = TIME$          ! copy so it can't change
hour = VAL (ti$)
tm$ = RIGHT$ (ti$, 8)   ! mm:ss ?M
minute = VAL (tm$)
second = VAL (RIGHT$(tm$, 5))
IF MID$ (tm$, 7, 2) = 'PM' THEN hour = hour + 12
```

Table 8-2. Formats for TIME$

Country	Time
United States	11:27:00 PM
France	23:27:00
Germany	23:27:00 Uhr
Great Britain	23:27:00
Italy	23:27:00

EXAMPLE PROGRAM

The example in Figure 8-2 is a program for a timer that uses the TIME$ function to measure any length of time up to 24 hours.

```
! Timer program
DO ! main program
    INPUT a$
    IF UPSHIFT$(a$) = "B" THEN GOSUB TStart
    IF UPSHIFT$(a$) = "E" THEN GOSUB TStop:
LOOP
END PROGRAM    ! protect subroutines

TStart:  ! remember starting value
    t$ = TIME$
    RETURN
TStop:          ! stop and calculate time elapsed
    te$ = TIME$
    GOSUB Convert    ! convert starting time to seconds
    st1 = timeC
    t$ = te$  ! ready to convert ending time
    GOSUB Convert    ! convert ending time to seconds
    IF timeC < st1 THEN timeC = timeC + 24*60*60
    time = timeC - st1
    hours = INT(time / 3600)    ! number of whole hours
    minutes = INT( (time-hours*3600) / 60)  ! number of minutes
    seconds = time - hours*3600 - minutes*60
    PRINT "The elapsed time was:"
    IF hours > 0 THEN PRINT hours; " hours"
    IF hours+minutes > 0 THEN PRINT minutes; " minutes"
    PRINT seconds; " seconds"
    RETURN
Convert:
    ! Get TIME$ from t$
    ! Return number of seconds since midnight in timeC
    timeC = 60*60*VAL(t$)  ! convert hours
    timeC = timeC + 60*VAL(MID$(t$, LEN(t$)-7))  ! minutes
    timeC = timeC + VAL(MID$(t$, LEN(t$)-4))  ! seconds
    IF RIGHT$(t$,2) = "PM" THEN timeC = timeC + 12*60*60
    RETURN
```

Figure 8-2. Timer program

The program uses three subroutines: one to start the timer, one to stop it, and one to convert the time into seconds.

The main timer program consists of a single DO loop that gets input from the keyboard. If the input is the single letter "b" in either upper- or lowercase, the program executes a GOSUB to the TStart subroutine. If the input is either an upper- or lowercase "e," the program executes a GOSUB to the TStop subroutine.

All the TStart subroutine does is record the value of TIME$ in the variable *t$* and return. When the TStop subroutine is called to stop timing, it records the value of TIME$ in a different variable, *te$,* and then uses the subroutine Convert to change the two time strings into seconds.

Note that the Convert subroutine takes its starting value from *t$* and returns its result in *timeC*. The value in *timeC* after the first GOSUB Convert had to be saved in another variable, or it would have been changed during the second execution of the subroutine. Before the second GOSUB, *t$* is set to the new value to be converted. The Convert subroutine uses several of the string functions to read the components of the time from a TIME$ string. The result in seconds is placed in *timeC.*

Once the program has converted the starting and ending times into numbers, it compares the two numbers. If the ending time is less than the starting time, the clock must have moved past midnight, so 24 hours' worth of seconds is added to the ending time. Once the elapsed time is stored in the variable time, the program uses the INT function to break the time down into hours, minutes, and seconds. When the results are printed, the IF statements prevent a zero from being printed as the first unit of time.

PRACTICE EXERCISES

1. What results are returned by the following function calls?
   a. LEFT$('abcdefg', 4)
   b. RIGHT$('yes, there are bananas', 6)
   c. MID$("public policy", 8, 3)
   d. MID$('banana split', 2)
2. Can you evaluate these expressions?
   a. LEFT$('Police',4) & RIGHT$('Attics',4)
   b. MID$('aspects',2,5) & MID$('trumpets',2,3)
3. Try writing a loop that uses LINE INPUT to read a typed number into a string variable, turns the string into an integer, and checks whether the input contains any extra characters.
4. Often you want your program to wait until the person using the program has pressed a key on the keyboard. INKEY$ returns a null string instead of waiting for the keypress if no key has been pressed. Can you write a subroutine that waits for a keypress and then returns the typed character in a variable named *c$?*
5. Write an "alarm clock" that asks for a time setting and prints "RING" when that time arrives. Allow the setting to be to the nearest minute.

Part two
Intermediate Techniques

Chapter 9

Variables, Data, and Arrays

Commands:

- DIM, UNDIM
- DATA, READ, RESTORE, FREE
- SET/ASK EXCEPTION, SET/ASK HALT
- SET/ASK PRECISION, SET/ASK ROUND

System Function:

- FREE

This chapter describes the types of variables you use in Macintosh BASIC and the types of data each variable can hold. The chapter then introduces arrays and data statements.

The concluding section contains information that will be primarily of interest to programmers writing sophisticated numerical programs. It provides a brief introduction to some of the technical aspects of the Macintosh's numeric computation environment.

VARIABLE TYPES

Macintosh BASIC has a rich variety of variable types. Each type holds a specific kind of data. There are ten variable types in all—five for numbers and five for different kinds of non-numeric data. The type of a variable is determined by the last character of the variable's name. Because the character that specifies the type of data is part of the variable's name, you can use the same word for the names of variables of different types without confusing the BASIC compiler.

Numeric Variables

Table 9-1 lists the five types of numeric variables in Macintosh BASIC. Real numbers, or reals, can have numeric values that include fractions. Integers can have only whole number values. Integers can have plus and minus signs, but they can never have decimal points or fractions. When you create a numeric variable, it contains the value zero until you store a different value in it.

Numbers that are very large or very small are often expressed as a number times a power of ten. The number 4000, for instance, can be expressed as $4*10^3$. The format is often shortened a bit by

Table 9-1. Numeric Variable Types

Type	Symbol	Digits of Accuracy	Range	Example
REALS				
Double precision	(none)	15	±1E308	name
Single precision	|	7	±1E38	name|
Extended precision	\	19	±1E4932	name\
INTEGERS				
Short integer	%	5	±32767	name%
Computational	#	18	±1E18	name#

omitting the multiplication sign and replacing the 10 with E (for exponent). With those changes, 4000 is expressed as 4E3. The first part of the number (4 in our example) is called the *mantissa;* the part of the number after the E is called the *exponent.*

When a real number is stored inside a computer, the mantissa and the exponent are stored separately. The amount of space reserved for the mantissa determines how many digits can be stored and retrieved accurately. The amount of space reserved for the exponent determines the maximum (and minimum) size of the value that can be stored in the variable. A double-precision variable, for example, has enough space to have 15 digits of accuracy in the mantissa, and the space reserved for its exponent is enough to handle exponents as large as 308.

A variable whose name does not end in a special character is a double-precision real. The 15-digit accuracy of a double-precision real variable is sufficient for most purposes. If you need greater precision or need to store numbers greater than 1E308 (1 with 308 zeros after it), you can use an extended-precision variable, which provides 19 digits of accuracy. An extended-precision variable's name ends with the \ character, which is located on the key between BACKSPACE and RETURN at the right of the keyboard. With SHIFT held down, the same key produces the | character that signifies the name of a single-precision real variable. BASIC performs internal calculations in extended precision and then rounds to the precision of the variable that is to receive the result.

Short integers hold values between −32767 and +32767. Their names end with the % sign, SHIFT-5 on the keyboard. If your program tries to store a larger or smaller value than the short integer variable can hold, you will receive an Integer Overflow error message.

Computational variables are sometimes called *comp variables* or *long integers.* They have names ending with the # sign, SHIFT-3 on the keyboard, and hold integer values up to 18 digits long.

Here are some examples of statements that use numeric variables:

```
LET a\ = 78E100  ! extended precision
LET a| = .07  ! single precision
a = 8800      ! double precision
a% = 77       ! short integer
a# = 77000  ! comp ( long integer)
```

Special Numeric Values

■ INFINITY, NAN

Several mathematical operations—dividing a number by zero or taking the square root of a negative number, for example—give an answer that cannot be expressed as a real number. When one of these operations is performed, Macintosh BASIC returns either INFINITY or NAN (Not A Number) as the result. INFINITY represents a number larger than any definable number. You get INFINITY when you divide a positive number by zero. −INFINITY is smaller than any definable number. You get −INFINITY when you divide any negative number by zero. NAN is the answer if you take the square root of a negative number or do some other invalid operation.

```
PRINT SQR(-1)  ! Displays 'NAN(1)'
PRINT 5/0      ! Displays 'INFINITY'
PRINT -5/0     ! Displays '-INFINITY'
```

You can set a numeric variable equal to one of these values if you wish. For infinity, you can use either the word INFINITY or the symbol ∞ (OPTION-5 on the keyboard). If your program tries to store a number into a real variable that is too large or too small to fit, BASIC changes the number to plus or minus INFINITY. If your program tries to store a number into a normal integer variable that is too large or too small to fit, BASIC gives you an Integer Overflow error message. If your program tries to store too large a number into a computational (long integer) variable, BASIC changes the number to NAN.

When BASIC prints a NAN value, it includes a number in parentheses. The number tells you what kind of invalid operation caused the NAN result. Table 9-2 lists the types of NANs you are likely to encounter and their causes.

Non-Numeric Variables

Table 9-3 lists the five non-numeric variable types: string, character, Boolean, pointer, and handle. A string is a series of characters

Table 9-2. Types of NANs

NAN	Operation Causing the NAN	Example
NAN(1)	Square root	SQR(−1)
NAN(2)	Addition or subtraction	−INFINITY + INFINITY
NAN(4)	Division	0/0
NAN(8)	Multiplication	0 * INFINITY
NAN(9)	MOD or REMAINDER	REMAINDER(v,0)
NAN(20)	Comp type value out of range	v# = 8E100
NAN(21)	NAN typed from keyboard	a = NAN
NAN(33)	Trigonometric function	SIN(INFINITY)
NAN(36)	Logarithmic function	LOG(−5)
NAN(37)	Exponentiation	(−1) ^ 0.5
NAN(38)	ANNUITY or COMPOUND	COMPOUND(0,INFINITY)

(letters, digits, punctuation marks, and special characters). When a string value is entered directly in a program, it is called a *string literal* and is enclosed in either single or double quotation marks. In Macintosh BASIC a string may contain as many as 65,535 characters. The name of a string variable ends with the $ symbol,

Table 9-3. Non-Numeric Variable Types

Type	Symbol	Example
String	$	name$
Character	©	name©
Boolean	~	name~
Pointer	]	name]
Handle	}	name}

SHIFT-4 on the keyboard. When you first create a string variable, it contains a null or empty string.

```
a$ = "test string"  ! string variable
b$ = a$ & ' *2'
PRINT b$
```

A character variable holds the ASCII value of a character. ASCII values range from 0 to 255. The name of a character variable ends with the © symbol, OPTION-g on the keyboard.

If you try to store a number larger than 255 into a character variable, the variable will hold the value of the number MOD 256. If you try to store a negative number into a character variable, the variable will hold the value of the number MOD 256 + 256. When you first create a character variable, it contains the value zero.

```
a© = ASC('*')
b© = 65  ! ASCII value of 'A'
PRINT CHR$(a©); CHR$(b©)
```

Boolean variables can hold only the two logical values—TRUE and FALSE. A Boolean variable's name ends with the ~ character (tilde). You can type the tilde by pressing the leftmost key in the top row while the SHIFT key is held down. When you create a Boolean variable, it contains the value FALSE until you store a different value in it.

```
a~ = TRUE
b~ = (index > 4)
PRINT a~, b~
```

Pointers and handles are special types of variables. They are used primarily when calling Macintosh toolbox procedures and functions. A *pointer* holds the address of a location in the Macintosh's memory, and a *handle* holds the address of a pointer. The name of a pointer variable ends with the character], and the name of a handle variable ends with the character }. These two types of variables are discussed more fully in Chapter 19.

Mixing Variables of Different Types

If you attempt to use a variable of an inappropriate type in a program statement, BASIC gives you a Type Mismatch error message. The Type Mismatch message is usually caused by trying to store a value from one variable into another variable that cannot accept that type of value. You cannot, for example, store a number into a Boolean variable, because a Boolean variable can only hold the value TRUE or FALSE.

All of the numeric variable types are compatible with each other and can be used in the same program statement without causing a Type Mismatch error. Because a character variable holds a number, it can also be used in calculations with numeric variables without causing a Type Mismatch error. The other four variable types—string, Boolean, pointer, and handle—do not mix with each other or with numeric variables. You can store a string, Boolean, pointer, or handle value only in its own type of variable.

USING ARRAYS

An *array* is a way of grouping several variables under a single name so you can access them efficiently. If you had a list of three numbers that you wanted to multiply by 10 and you put each number in a separate variable, your program might look something like Figure 9-1. You would get very tired before you finished typing this kind of program if your list had several hundred numbers instead of only three.

A better way to write this program would be to organize the values in a list. Then you could refer to each number as the first number in the list, second number in the list, and so forth. If you went a step further, you could use a variable—called an *index variable*—to keep track of your place in the list. That way you could use a FOR/NEXT loop and an index variable to reference each value in the list. Your program would require only one statement to multiply and one statement to print.

The kind of indexed list just described is an example of an array. You can refer to an individual value in the array by using the array name and the index that tells where the value is located in the

```
a = 3
b = 7
c = 9
a = a * 10
b = b * 10
c = c * 10
PRINT a
PRINT b
PRINT c
END PROGRAM
```

Figure 9-1. Using separate variables

array. If you used an array, your program might look something like Figure 9-2.

Creating Arrays

■ DIM

When you create an array, you need to specify the maximum dimension of the array so BASIC will know how much space to set aside in your machine's memory to store each of the values. The

```
DIM a(3)
a(1) = 3
a(2) = 7
a(3) = 9
FOR index = 1 TO 3
    a(index) = a(index) * 10
    PRINT a(index)
NEXT index
END PROGRAM
```

Figure 9-2. Using an array

DIM or dimension statement serves this purpose. The keyword DIM is followed by the array name, with the maximum size of the dimension in parentheses. The dimension can be a number or any legal numeric expression. It can have any value from 0 to 32767. An array you dimension at 9 contains 10 elements because Macintosh BASIC always allocates the 0th element.

You can dimension more than one array in the same statement if you separate the array names with commas.

```
DIM array1(15)
DIM array2(60), array3(30)
```

Array names follow the same rules as the names of simple variables. The same ten variable types shown in Tables 9-1 and 9-3 also apply to arrays. The last letter of an array name specifies the type of variables that can be stored in the array.

```
DIM array(70),array~(30),array$(50)
DIM array%(40),array#(80)
```

You can have an array with the same name as a simple variable, if you are willing to risk getting them mixed up in your own mind.

The DIM statement for an array must be executed before you attempt to reference any element of the array, or BASIC will give you an error message. Because of this, DIM statements are often grouped together near the beginning of a program. If the names of your arrays are not self-explanatory, it is a good idea to put a comment near the beginning of your program describing the values each array will hold.

If you attempt to reference an element of an array with an index that is negative or an index that is larger than the size of the array, BASIC will give you an error message.

Arrays With More Than One Dimension

You can define an array with as many dimensions as you like. All you do is use commas to separate each dimension inside the parentheses in the DIM statement. Two-dimensional arrays are often used for data that can be displayed in tables with rows and columns.

An address list can be displayed on paper with each horizontal line made up of first name, last name, street address, city, state, and ZIP code columns, as shown in Figure 9-3. If the list contains 50 addresses, the statement

```
DIM address.list$(50,6)
```

creates an array that can hold the address list. It dimensions an array with 50 rows and 6 columns.

To access a particular element of an array, you use the array name followed by indexes in parentheses to indicate which element you want. You need to supply one index for each of the array's dimensions. In the previous example, the first dimension represents the row that holds a particular address, and the second dimension represents a specific column or part of the address. If you wanted to sort the list into alphabetical order by names, you would first sort on the last name in the second column. In the event of a tie in the second column, you would sort on the first name in column 1.

Whenever you refer to an element of an array in your program, you must use the correct number of dimensions. If you do not, you will receive an error message. You cannot use the same name for

```
First Name Last Name  Address        City     State Zip

Jane       Doe        123 Z Street   Anytown  HI    90000
John       Smith      456 EZ Lane    Anytown  CA    90099
.
.
.
```

Figure 9-3. Address list

two arrays of the same data type, even if they have a different number of dimensions.

Copying Arrays

You can copy values from one array into another by using index variables in FOR/NEXT loops. If you want to copy an entire array, however, Macintosh BASIC allows you to do it with a single program statement. You can use the normal LET statement to assign the values of one array to another, without using any values inside the parentheses:

```
array1() = array2()   ! copy entire array
```

If the arrays have more than one dimension, the appropriate number of commas must be included inside the parentheses, as in the following example of two-dimensional arrays:

```
array1(,) = array2(,)   ! copy 2-dimensional array
```

If the two arrays do not have the same number of dimensions, you will receive an error message. The size of each dimension does not, however, have to be identical in the two arrays. As long as the number of dimensions is the same, BASIC changes the dimensions of the destination array to match the dimensions of the array being copied.

Removing an Array

■ UNDIM

The UNDIM command "undimensions" an array. UNDIM frees all the memory occupied by the array. You lose the values stored in the array unless you save them somewhere else before you use UNDIM. The word UNDIM must be followed by the array name and a set of parentheses. The parentheses must contain the appropriate number of commas if the array has more than one dimension. If you have more than one array to undimension, you can separate the array names with commas in a single UNDIM statement.

```
DIM array1(300),array2(50,3),array3(4,2,2)
    ! Program statements using the arrays
UNDIM array3(,,)
UNDIM array1,array2(,)
```

When you use UNDIM, you remove the array from memory. It no longer exists, and any later references to the array in your program will cause error messages.

PUTTING DATA IN YOUR PROGRAM

■ DATA, READ

The program in Figure 9-2 used an array to handle the multiplication and printing of a series of numbers more efficiently than the program in Figure 9-1. However, the program in Figure 9-2 still contained a separate program statement to set the value of each element of the array. Those separate statements can be replaced with the more efficient DATA and READ statements, as shown in the following program:

```
DIM a(3)
DATA 3,7,9
FOR index = 1 TO 3
    READ a(index)
    a(index) = a(index) * 10
    PRINT a(index)
NEXT index
END PROGRAM
```

READ works very much like INPUT, but READ takes the input values from DATA statements embedded in the program instead of from the keyboard. After the keyword READ, you list the variables whose values are to be taken from the DATA statements. If you have more than one variable name in the same READ statement, separate them with commas. Each variable in the READ command must be compatible with the type of data in the DATA statement, or you will receive a Type Mismatch error message. Here are some examples:

```
DATA 8,test,TRUE,9
READ a,b$,c~,d      ! Yes, types all match
READ a,b% ,c~,d     ! No, b% cannot receive a string
```

Values in a DATA statement are separated by commas. Each value must be an actual, or literal, value, not an expression. DATA statements can be located anywhere in your program. If a DATA statement is encountered during program execution, BASIC skips the DATA statement and resumes execution with the next program line.

String values in DATA statements do not have to be surrounded by quotation marks unless they contain a comma or quotation marks. If you want to include either single or double quotation marks inside a string in a DATA statement, use the opposite kind of quotation mark at both ends of the string, as in these examples:

```
DATA "This a fine day, isn't it?"
DATA 'The " marks are in this string.'
```

The READ statement ignores spaces at the beginning of an unquoted string in the DATA statement, but includes spaces at the end of the string. If you want it to include one or more spaces at the beginning of a string, you should enclose the string in quotation marks.

READ and DATA statements are often used to initialize the values of an array, as in this example:

```
DIM days$(7)
DATA Monday, Tuesday, Wednesday
DATA Thursday, Friday, Saturday, Sunday
FOR day = 1 TO 7
    READ days$(day)
NEXT day
```

MOVING THE DATA POINTER

- RESTORE

BASIC maintains a pointer to the next DATA item to be read. When execution of your program begins, the pointer points to the first item in the first DATA statement in your program. Each time a data item is read by a READ statement, BASIC advances the pointer to the next data item. You can change the pointer with the RESTORE command.

RESTORE by itself moves the data pointer back to the first DATA item in your program. RESTORE followed by a line number or label moves the data pointer to the first DATA item in your program following the occurrence of the line number or label. The RESTORE command comes in handy when you need to use the same data more than once in your program.

```
DATA 1,2,3
label: DATA 5,8,9
RESTORE
READ a  ! reads 1
RESTORE label
READ a  ! reads 5
```

UTILIZING THE AVAILABLE MEMORY

- FREE

Many things can occupy space in Macintosh's random access memory: the operating system, Macintosh BASIC itself, desk accessories, open windows, pictures, and other programs. Your program cannot control all of these, but it can control its own use of the machine's memory. FREE is a system function that takes no arguments. It returns the number of bytes that are not being used. You can print the value of FREE to find out how much room is left, or you can use FREE in a program that needs to know the amount of room left in memory.

```
IF FREE < 1000 THEN
    PRINT "Warning: "
    PRINT "Running out of memory."
    PRINT FREE; " bytes left."
ENDIF
```

Of all the things that can take up large amounts of memory, the one over which you have the most control is the array. An array dimensioned (9,9) has ten times ten (don't forget, space is reserved for element 0), or 100, elements. You can multiply the number of elements by the number of bytes each element occupies to calculate the minimum amount of memory space required for the array. Table 9-4 lists the number of bytes required to store a single element of each variable type.

An array with 100 double-precision elements needs 100 times 8, or 800, bytes of memory in which to store the elements.When storing a string value, Macintosh BASIC uses two bytes to store the length of the string and then uses one additional byte for each character in the string. You can, if you wish, base your program's dimension statements on the amount of free memory. If you do, be sure to leave plenty of extra bytes free for other uses. Here is one way you could do this:

```
! Make array of double precision reals
number = (FREE-10000) / 8  ! 8 bytes per element
DIM array(number)
PRINT "Array has "; number; " elements."
```

Table 9-4. Variable Storage Requirements

Variable Type	Bytes per Value
Double Precision	8
Single Precision	4
Extended Precision	10
Short Integer	2
Computational (long integer)	8
String	2 + length of string
Character	1
Boolean	1
Pointer	4
Handle	4

THE NUMERIC ENVIRONMENT

The Macintosh has a numeric calculation environment that exceeds the precision used in calculations on many large mainframe computers. The Macintosh's numeric environment is called the Standard Apple Numeric Environment (SANE). It meets the standards promulgated by The Institute of Electrical and Electronics Engineers, Inc., which is an industry organization that sets standards for things related to computing.

The commands described in this section allow you to change several features of the SANE environment. Most programmers do not need to use these commands; they are useful for persons doing advanced numeric programming. If you want to delve into these matters more deeply than the very brief discussion here, you can find more information in the *Apple Numerics Manual* published by Apple Computer, Inc.

When Calculations Don't Work

- SET/ASK EXCEPTION, SET/ASK HALT

The numerics environment allows you to find out if any unusual events happened during numeric calculations. Table 9-5 lists the five conditions for which you can check. These conditions are called *exceptions.*

To check for an exception, use the command ASK EXCEPTION followed by the name or number of the exception, a space, and the

Table 9-5. Exception Conditions

Exception	Value	Cause
INVALID	0	Invalid operation, result is NAN
UNDERFLOW	1	Result so small it rounded to zero
OVERFLOW	2	Result so large it became INFINITY
DIVBYZERO	3	Division by zero, result INFINITY
INEXACT	4	Calculation result had to be rounded

name of a Boolean variable. To check whether a division by zero has occurred, you could use the statement

```
ASK EXCEPTION DIVBYZERO variable~
```

which would set your Boolean variable to TRUE if the condition has occurred and to FALSE if it has not. Once it becomes TRUE, the exception will always return TRUE until you use a SET EXCEPTION statement to reset it to FALSE. You follow the command SET EXCEPTION with the exception name or number and the Boolean value TRUE or FALSE.

HALT determines whether or not your program stops when an exception occurs. Follow the command SET HALT with the name or number of an exception, a space, and a Boolean expression that evaluates to TRUE or FALSE. To find out whether a halt is already set, use ASK HALT followed by the name or number of the exception, a space, and a Boolean variable to receive the answer.

The statement

```
SET HALT 2 TRUE
```

causes your program to halt with an error message if a calculation causes an overflow condition. All of the exceptions and halts are initialized as FALSE when you begin to run your program.

Controlling Precision and Rounding

■ SET/ASK PRECISION, SET/ASK ROUND

The numerics environment also allows you to set or ask the degree of precision being used in BASIC's internal calculations and the direction in which rounding of decimal fractions will occur. Table 9-6 lists the choices of precision. You can change the precision by using SET PRECISION followed by the appropriate word or number from the table. If you follow ASK PRECISION with the name of a numeric variable, BASIC puts the precision number in the variable. When your program begins, PRECISION is set to EXTPRECISION.

Table 9-6. Calculation Precision

Constant	Value	Precision
EXTPRECISION	0	Extended precision, 80 bits
DBLPRECISION	1	Double precision, 64 bits
SGLPRECISION	2	Single precision, 32 bits

You use SET ROUND and ASK ROUND to change the rounding direction for the RINT function and all other internal calculations that require rounding a real number to a certain number of decimal places or to an integer. Table 9-7 lists the possible rounding directions. Follow ASK ROUND with the name of a numeric variable to learn the current rounding direction. To change the rounding direction, use SET ROUND followed by the name or number of the direction you want. ROUND is set to TONEAREST when your program begins.

Here are a few short examples using PRECISION and ROUND:

```
ASK PRECISION what%   ! puts 0 in what% if extended
SET ROUND TowardZero  ! sets rounding toward zero
SET ROUND 0   ! sets rounding back to ToNearest
ASK ROUND where  ! puts 0 in where if ToNearest
```

Table 9-7. Rounding Directions

Direction	Value
TONEAREST	0
UPWARD	1
DOWNWARD	2
TOWARDZERO	3

Commands for Numerics Experts

Macintosh BASIC provides twelve additional commands that give you access to the most sophisticated features of the Apple numerics environment. You are not likely to use these commands unless you are involved in extremely sophisticated numerical programming. They are listed here so you will know they exist. Table 9-8 summarizes these additional numerics environment commands. You will need to refer to the *Apple Numerics Manual* for the details for these commands.

Table 9-8. Additional Numerics Environment Commands

ASK ENVIRONMENT	Followed by a numeric variable name, saves one number that describes the entire numerics environment.
SET ENVIRONMENT	Followed by a number obtained from ASK, restores that numerics environment. 0 restores the default numeric environment.
PROCENTRY	Followed by a numeric variable name, saves the current numeric environment in that variable and sets the default numeric environment.
PROCEXIT	Followed by an environment number saved by PROCENTRY or ASK ENVIRONMENT, resets that environment. Use SET EXCEPTION INVALID FALSE after using PROCEXIT.
REMAINDER	Function takes two numeric arguments and returns an integer remainder derived from the result of the first argument divided by the second argument, as explained in the *Apple Numerics Manual.*
CLASSCOMP	Function takes one numeric argument and returns the class number of the argument as if it were converted to type comp. Classes are 0 for SNAN, 1 for QNAN, 2 for INFINITE, 3 for ZeroNum, 4 for NormalNum, 5 for DenormalNum. For an explanation of these types, see the *Apple Numerics Manual.*

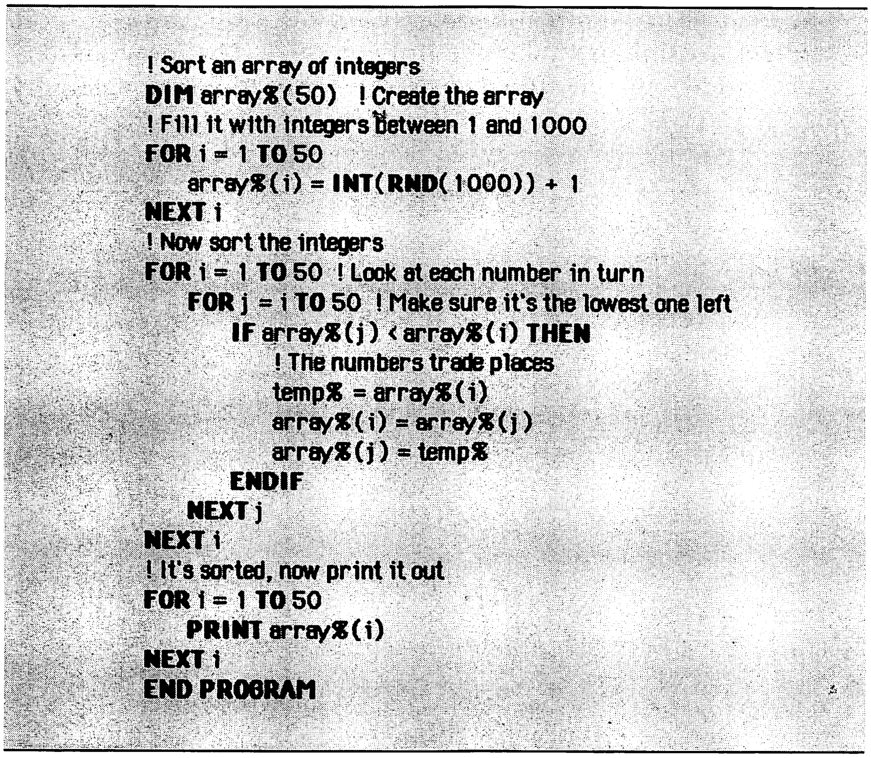
Table 9-8. Additional Numerics Environment Commands (*continued*)

CLASSDOUBLE	Function takes one numeric argument and returns the class number of the argument as if it were converted to type double. Classes are listed under Classcomp.
CLASSEXTENDED	Function takes one numeric argument and returns the number class of the argument as if it were converted to type extended. Classes are listed under Classcomp.
CLASSSINGLE	Function takes one numeric argument and returns the number class of the argument as if it were converted to type single. Classes are listed under Classcomp.
NEXTDOUBLE	Function takes two numeric arguments and returns the next representable value after the first argument in the direction of the second argument with all numbers treated as double-precision.
NEXTEXTENDED	Function takes two numeric arguments and returns the next representable value after the first argument in the direction of the second argument with all numbers treated as extended-precision.
NEXTSINGLE	Function takes two numeric arguments and returns the next representable value after the first argument in the direction of the second argument with all numbers treated as single-precision.

EXAMPLE PROGRAMS

The two example programs in this section use many of the features introduced in this chapter. The first program creates, sorts, and displays an array of 50 integers. The second example program prints a list of names and addresses in alphabetical order.

The sorting example in Figure 9-4 uses a DIM statement to dimension an integer array. The array will hold 51 elements (0 through 50), but the program does not use the 0th element. A FOR/NEXT loop stores a random integer between 1 and 1000 into each element of the array.



```
! Sort an array of integers
DIM array%(50)  ! Create the array
! Fill it with integers between 1 and 1000
FOR i = 1 TO 50
    array%(i) = INT(RND(1000)) + 1
NEXT i
! Now sort the integers
FOR i = 1 TO 50  ! Look at each number in turn
    FOR j = i TO 50  ! Make sure it's the lowest one left
        IF array%(j) < array%(i) THEN
            ! The numbers trade places
            temp% = array%(i)
            array%(i) = array%(j)
            array%(j) = temp%
        ENDIF
    NEXT j
NEXT i
! It's sorted, now print it out
FOR i = 1 TO 50
    PRINT array%(i)
NEXT i
END PROGRAM
```

Figure 9-4. Sort an array of integers

The sort routine begins with a FOR/NEXT loop that points in turn to each position in the array, from lowest to highest. While the first FOR/NEXT loop points to one element of the array, a second FOR/NEXT loop searches from that element to the end of the array. An IF statement inside the second loop tests whether the element pointed to by the second loop is smaller than the element pointed to by the first. If so, the elements are exchanged. After the second loop is finished, the element pointed to by the first loop is now the smallest one in the array, so the program can look for the next larger element.

This sorting method is not always the fastest, but it is easier to understand than many of the other methods. You can scroll through the listing in the output window to confirm that the array has been sorted correctly.

The example in Figure 9-5 uses a two-dimensional array to store an address list similar to the one in Figure 9-3. This program displays the address list in alphabetical order, but it does not actually

```
! Display address list in name order
DIM address.list$(50,6)
! 2nd dimension is first name,last name,street,city,state,ZIP
DATA Jane,Doe,123 Z Street,Anytown,HI,90000
DATA John,Smith,456 EZ Lane,Anytown,CA,90099
DATA Shawn,OReilly,"2 O'Hara Street",Midtown,IA,88888
DATA Bobbie,Oregon,1 Evrystreet,Bigcity,NY,10000
DATA Sam,Spade,3 Nowhere Lane,Somewhere,MD,20000
! Read the data into the array
FOR address = 1 TO 5
    FOR item = 1 TO 6
        READ address.list$(address,item)
    NEXT item
NEXT address
! Initialize for the display routine
OPTION COLLATE NATIVE
LName.done$ = ""   ! Last name done so we don't repeat
address.list$(0,2) = CHR$(255)   ! Larger than any name's first letter
FOR times = 1 TO 5   ! 5 names to print
    ! Select the next name in order
    next.to.print = 0   ! Start by pointing to something larger than any name
    FOR address = 1 TO 5
        IF address.list$(address,2) > LName.done$ THEN
            IF address.list$(address,2) < address.list$(next.to.print,2) THEN
                next.to.print = address
            ENDIF
        ENDIF
    NEXT address
    ! We have the next one, so print it
    FOR i = 1 TO 6
        PRINT address.list$(next.to.print,i);' ';
    NEXT i
    PRINT   ! End the display line
    LName.done$ = address.list$(next.to.print,2)
NEXT times
END PROGRAM
```

Figure 9-5. Display address list in name order

move the elements of the array. The program uses a DIM statement to dimension the string array *address.list$* for 50 rows and 6 columns. Two nested FOR/NEXT loops control a READ statement that reads strings from the DATA statements into elements of the array. The DATA statements could have been located anywhere in the program.

An OPTION COLLATE NATIVE statement tells BASIC to sort strings in normal alphabetical order (see Chapter 8 for a review of this statement). A variable that will be used to store the most recently printed last name is initialized with a null string, and the last name position in the 0th row of the array is filled with CHR$(255), a value larger than any alphabetic character. The program points to this element when it begins searching for the next last name to print.

A FOR/NEXT loop finds and prints the next name and address five times. The variable named *next.to.print* is set to 0, so the expression *address.list$(next.to.print,2)* will return the CHR$(255) value stored earlier. A FOR/NEXT loop looks at each entry to see if the last name of that person is greater than the previous name printed (if it is not greater, the name being examined has already been printed) and also less than the name pointed to by the variable *next.to.print.* If the name meets both of these conditions, the row number containing that name is stored in *next.to.print.* Initializing *next.to.print* to point to CHR$(255) ensures that the row number of the first name not already printed will be stored in *next.to.print.* When the loop ends, *next.to.print* contains the row number of the name lowest in alphabetical order that has not yet been printed. A final FOR/NEXT loop prints the six strings that make up the full name and address.

If you examine Figure 9-5 closely, you may notice that the program as written will only print one address for each last name, even if several people in the list have the same last name. You can correct this by adding statements to compare the first names if the entry in the array has the same last name as the one just printed.

PRACTICE EXERCISES

1. How many elements of what data type can be stored in each of the following arrays?

   a. Variable(44)

   b. Iftest~(8)

   c. Bigtime \(900,2)

   d. Whoknows%(99)

   e. Name$(9,2)

2. What is the dimension of the array named q$ when this program reaches the END PROGRAM statement:

```
DIM a$(3),q$(9)
q$(9) = "how do you like this?"
PRINT q$
q$() = a$()
END PROGRAM
```

3. Which of the following statements will cause error messages?

   a. a$ = 8

   b. a() = b(,)

   c. test~ = 0

   d. string = 'value'

   e. a% = 50000

4. What is wrong with this READ/DATA combination:

```
DATA 4,test,5,test
READ a,a$,b,c
```

Chapter 10
Formatting Program Output

Commands:

- PRINT, SET/ASK VPOS, SET/ASK HPOS
- GPRINT, SET/ASK PENPOS
- SET/ASK FONT, SET/ASK FONTSIZE
- SET/ASK GTEXTFACE, SET/ASK GTEXTMODE
- GTEXTNORMAL, SET/ASK SHOWDIGITS
- CLEARWINDOW, SET/ASK TABWIDTH
- DOCUMENT PRINT

Functions:

- TAB FORMAT$

This chapter describes the commands that affect the arrangement and appearance of the text printed in the output window. First some of the fine points about the PRINT command, which you have been using since your first BASIC program, are covered. Then the chapter discusses GPRINT, the second print command in Macintosh BASIC. The GPRINT command controls the font, size, and appearance of text in the output window. Next, this chapter describes the FORMAT$ function and the ways it can help you arrange information. Lastly, this chapter describes how to print a copy of the ouptput window on paper.

PRINTING NORMAL TEXT

Normal text is printed with the PRINT command. The output window displays only a portion of the output document. You can think of the output document under the window as a sheet of paper 8 1/2 by 11 inches in size.With a 12 point font size, the output document will hold 48 lines of text.

The original size of the output window allows you to see a portion of the output document that is approximately 15 lines high by 30 characters wide if you are using a 12 point font size (the exact number of characters varies because most fonts are proportional). If your program prints more lines of text than fit in the window, the text scrolls upward so that the most recently printed text is visible. You can change the font and size of normal text output by using the Fonts menu as described in Chapter 2, but you cannot change these attributes from your program.

A PRINT Command Refresher

- PRINT

You follow the PRINT command with a list of numbers and strings separated by semicolons or commas. The numbers and strings can be literal values or expressions. If the separator is a semicolon, PRINT displays one value immediately after another, with no intervening spaces. If the separator is a comma, PRINT moves to the next tab stop before displaying the next value. The position where the next character is printed is called the *text insertion point.*

BASIC issues a carriage return character at the end of each PRINT statement so the next PRINT statement will begin printing at the start of the next line. You can suppress the carriage return character by ending your PRINT statement with a semicolon or a comma.

Before the PRINT command starts printing on a new line, it erases everything else on that line. That action, of course, destroys any graphics you may have drawn there. You can use PRINT with graphics in the same window if you are careful to print before you draw the graphics. However, you may get a surprise if you try to use the Copy Picture command to save a copy of your completed output window. You will get a copy of all your graphics information, but none of the text displayed with the PRINT command will appear on the copy. This is because output from the PRINT command is kept in text format and is never converted to graphics format. You will get much better results if you use the GPRINT command whenever you have any graphics in the output window and restrict your use of the PRINT command to those times when you only want to display text.

Positioning Normal Text

- SET/ASK VPOS, SET/ASK HPOS

VPOS and HPOS are special variables that control the location where your next PRINT or INPUT statement will begin. VPOS is the vertical position or line of text in the output document, and HPOS is the horizontal character position within the line of text.

You can change the vertical position by using the SET VPOS command followed by a number or numerical expression. You can find out the current setting by following the ASK VPOS command with the name of a numeric variable. BASIC will store the value of VPOS in that variable.

```
SET VPOS 3  ! Sets to PRINT on line 3
ASK VPOS vOld  ! Gets VPOS in vOld
SET VPOS vOld+3  ! Sets to line vOld+3
```

BASIC accepts values for VPOS ranging from 1 to the number of the last line of the document. With a 12 point font size, 48 lines fit in a normal size output document. You can get more lines in your output document by using a smaller font size or enlarging the document with the SET DOCUMENT command described in Chapter 16.

ASK HPOS followed by a numeric variable name returns the number of characters between the text insertion point and the left edge of the output document. You use SET HPOS to specify the position where you want the next character printed. BASIC accepts values in the range 1 to 256 for SET HPOS. If you set the number too high, however, the insertion point may be past the right edge of the document, where you will not be able to see it.

```
PRINT "The story ";
ASK VPOS vOld  ! Gets VPOS in vOld
ASK HPOS hOld  ! Gets HPOS in hOld
SET VPOS 10
PRINT "At a new location"
SET VPOS vOld  ! Restore old line
SET HPOS hOld  ! Restore character position
PRINT "continues."
```

When your SET HPOS value requires BASIC to count character positions on an empty line or on part of a line where text has not been printed, BASIC uses the width of a numeric digit in the current type font for the width of each character position. Digits are eight pixels (screen dots) wide in most 12 point fonts (seven pixels for 12 point Monaco), and six pixels wide in most 9 point fonts. If you set VPOS or HPOS to a location that is beyond the area visible in the output window, BASIC scrolls the window's contents to display the insertion point.

Using the Tab Function

■ TAB

The TAB function is used within a PRINT statement to move the text insertion point to a specific character position to the right. This makes it possible for you to move the insertion point between each item you print and still print several items with one PRINT statement. You follow the word TAB with parentheses containing your new HPOS setting and then with a semicolon.

```
PRINT "testing";TAB(10);"1, 2, 3."   ! TAB skips 2 characters
PRINT "testing";TAB(5);"1, 2, 3."    ! TAB does not move
```

The TAB function moves the text insertion point only to the right, never to the left. If you have already printed past the new HPOS setting, TAB has no effect. If you have not printed as far as the new HPOS setting, TAB skips enough blank character positions to move to your new setting. You can use TAB as often as you wish in the same PRINT statement.

PRINTING GRAPHICS TEXT

The GPRINT command prints graphics text. Graphics text is merged with any information that is already in the output window and text may appear on top of whatever was already in the document. Nothing is erased unless you use a separate command to do so. The location of graphics text in the window is measured in pixels (the dots on the screen) instead of in lines and characters. The original size of the output window is 240 pixels high and 240 pixels wide. The full Macintosh 9-inch screen is 342 pixels high by 512 pixels wide.

The GPRINT Command

- GPRINT

You should use the GPRINT, or graphics print, command instead of PRINT when you have both text and graphics to print in the same output document. You can write GPRINT statements just like PRINT statements. Commas and semicolons have the same effects in GPRINT statements as they do in PRINT statements.

GPRINT does not mix well with either the PRINT or INPUT command. Both PRINT and INPUT erase graphics from the line to be printed, which makes it difficult to use them with any graphics already in the window. If you are using any fancy type, you will probably find it much easier to do all your printing with GPRINT and to put all necessary INPUT statements before your graphics output routine.

```
GPRINT a, b  ! Prints values of a and b
GPRINT "a";"b"  ! Prints 'b' immediately after 'a'
GPRINT "a","b"  ! Prints 'a', tabs, then prints 'b'
```

It is important to remember to set the printing location for the first GPRINT command in your program. If you do not set it, the first GPRINT statement will print your text outside the visible part of the output document.

Positioning Graphics Text

■ SET/ASK PENPOS

GPRINT has its own commands to set the position of the text in the output document. VPOS and HPOS do not affect the location of the text GPRINT displays. BASIC allows you to use TAB in GPRINT statements, but the results are not guaranteed.

While GPRINT appears to print text, it is really a graphics command. You learned earlier about the insertion point BASIC maintains to keep its place in text output. BASIC also maintains a separate pointer, called the *graphics pen,* to keep its place in graphics output. Each time the graphics pen is used by a graphics command (including GPRINT), BASIC updates the location of the graphics pen. The GPRINT command always begins its next display at the current location of the graphics pen.

You express the location of the graphics pen as a pair of numbers separated by a comma. The first number is the number of pixels to the right of the left edge of the window. The second number is the number of pixels down from the top of the window. When your program starts, the graphics pen is located at pixel 0,0—the top left corner of the output window. The coordinates become larger the farther you move down or to the right. For example, the pen coordinate 10,20 is 10 pixels to the right and 20 pixels down from the top left corner of the output window.

To move the graphics pen to a new location, you can use the SET PENPOS command. A SET PENPOS statement looks like

```
SET PENPOS a,b
```

where a and b are the horizontal and vertical coordinates of the location where you are setting the graphics pen. Both a and b are numeric expressions, and they are separated by a comma. SET

PENPOS 7,12 sets the pen so GPRINT will begin printing in the same place as PRINT would.

ASK PENPOS gets the current graphics pen location. Follow the command ASK PENPOS with two numeric variable names separated by commas. BASIC puts the horizontal position into the first variable and the vertical position into the second variable. Here are a few examples of SET PENPOS and ASK PENPOS:

```
SET PENPOS 7,12 ! Sets pen for first line
ASK PENPOS h, v  ! Puts 7 in h, 12 in v
```

You can also set the graphics pen position within the GPRINT statement. To do that, use GPRINT AT followed by the coordinates, a semicolon, and the items you want GPRINT to display. You can use AT only at the very beginning of a GPRINT statement, not after any item has been printed.

```
GPRINT AT x,y; "Hello"  ! Sets pen to x,y and prints "Hello"
GPRINT AT 7,12; a$  ! Prints a$ on first line
ASK PENPOS h, v  ! Save pen position
SET PENPOS h, v  ! Sets pen back for GPRINT
```

When you print graphics text, the text is displayed to the right of the graphics pen position with the base of the characters even with the pen position. Thus, the text appears above and to the right of the graphics pen position. This means that when the pen starts at location 0,0, any text printed will be above the output window. That is why you have to use SET PENPOS or GPRINT AT to set the location for the first GPRINT command in your program.

GPRINT remembers the most recent horizontal position specified by SET PENPOS or GPRINT AT and uses that position as its left margin until another SET PENPOS or GPRINT AT command changes it. GPRINT ignores carriage return characters (CHR$(13)) when they are embedded in strings longer than one character, but it prints them in one-character strings. You will need to use either SET PENPOS or GPRINT AT to reset the location for your GPRINT commands whenever you have used PLOT, CLEARWINDOW, or any other commands that might cause the graphics pen to be moved.

Controlling the Graphics Text Font

■ SET/ASK FONT, SET/ASK FONTSIZE

With GPRINT, you control the font, size, and appearance of graphics text. You cannot change these attributes later with the Fonts menu. SET FONT changes the font used by GPRINT to the font whose number you specify. When you use the command ASK FONT followed by the name of a numeric variable, BASIC puts the number of the current font into the variable. The SET FONT and ASK FONT statements use the font numbers shown in Table 10-1.

```
SET FONT 0          ! Sets Chicago font
ASK FONT oldfont%   ! Gets font number
PRINT               ! Sets to application font
SET FONT oldfont%   ! Resets font for GPRINT
```

Table 10-1 shows the most generally available fonts, their font numbers, and the sizes in which each font is available. You can

Table 10-1. Generally Available Fonts and Sizes

Font Number	Name	Sizes
0	System (Chicago)	12
1	Application (Geneva)	12
2	New York	9, 10, 12, 14, 18, 20, 24, 36
3	Geneva	9, 10, 12, 14, 18, 20, 24, 36
4	Monaco	9, 12
5	Venice	14
6	London	18
7	Athens	18
8	San Francisco	18
9	Toronto	9, 12, 14, 18, 24
10	Seattle	10, 20
11	Cairo	18
12	Los Angeles	12, 24

learn whether a font is present on your BASIC disk by looking to see if it is listed in the Fonts menu.

Unless you change them, the font and size used by GPRINT are set to the standard Application font, Geneva 12. If your program requests a font that is not available, BASIC uses the Geneva font. The PRINT and INPUT commands always use the Geneva 12 font unless you have made a selection from the Fonts menu to change them.

Setting the font size is done separately from setting the font itself. SET FONTSIZE followed by a number sets the font to that size, and ASK FONTSIZE puts the current font size in the numeric variable whose name you include in the ASK FONTSIZE statement. You use the actual font size in SET FONTSIZE and ASK FONTSIZE statements.

```
SET FONTSIZE 9  ! Sets for 9-point type
ASK FONTSIZE size  ! Gets font size
SET FONTSIZE size+3  ! Sets to a larger size
```

The most common sizes are 12 point and 9 point. In the 12 point size, the base of each line of text is 16 pixels below the base of the line above it; in the 9 point size, the distance is 12 pixels. Your type fonts look best if you display them in sizes that are present on your disk. If you request a size that is not available, BASIC attempts to scale the font to your requested size. The result is likely to be unappealing, or even unreadable, if the requested size is not an even multiple of an available size.

Controlling the Appearance of Graphics Text

■ SET/ASK GTEXTFACE, SET/ASK GTEXTMODE

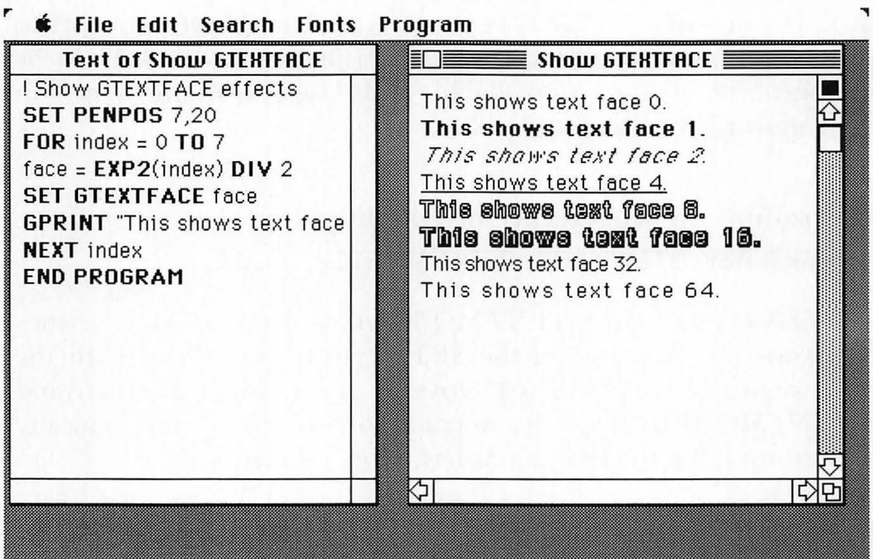
GTEXTFACE and GTEXTMODE are two more characteristics that you can change with the SET keyword and query with the ASK keyword. GTEXTFACE governs the style of the text, and GTEXTMODE governs the interaction between the text and any background design. Here are a few short examples:

```
SET GTEXTFACE 9     ! Set for boldface outline type
SET GTEXTMODE 10    ! Make text visible on any background
ASK GTEXTFACE face  ! Get GTEXTFACE in face
ASK GTEXTMODE mode! Get GTEXTMODE in mode
```

Table 10-2. GTEXTFACE Settings

Style	Value
Plain	0
Boldface	1
Italic	2
Underline	4
Outline	8
Shadow	16
Condense	32
Extend	64

Table 10-2 shows the possible GTEXTFACE style characteristics and their values. You calculate the GTEXTFACE setting by adding together the values of the style characteristics you want to use. SET GTEXTFACE 3, for example, produces boldface italic type. If the



```
! Show GTEXTFACE effects
SET PENPOS 7,20
FOR index = 0 TO 7
face = EXP2(index) DIV 2
SET GTEXTFACE face
GPRINT "This shows text face
NEXT index
END PROGRAM
```

```
This shows text face 0.
This shows text face 1.
This shows text face 2.
This shows text face 4.
This shows text face 8.
This shows text face 16.
This shows text face 32.
This shows text face 64.
```

Figure 10-1. GTEXTFACE samples

condense and extend options are both selected, they cancel each other out and you get normal spacing between letters.

Figure 10-1 shows a short program that displays each of the GTEXTFACE style characteristics in the Geneva 12 font. Some of the style characteristics look better when combined with other characteristics than they do alone (italics tend to look better in boldface, for instance). Experiment with various combinations of characteristics until you find some combinations that you like.

The GTEXTMODE settings range from 8 to 11, as shown in Table 10-3. The normal graphics text mode is 9, in which letters are displayed in black and no changes are made to the background of the letters. GTEXTMODE 8 erases the background before displaying black letters. Text mode 10 is handy when you want to see text on any color background, because it prints each letter in the opposite color from the background. GTEXTMODE 11 displays the letters in white, so it is useful only when you know the background is already black.

The program in Figure 10-2 demonstrates the basic differences between the four GTEXTMODE settings. The GTEXTMODE settings are very similar to the PENMODE settings that govern the way the graphics pen draws designs. If you want to explore GTEXTMODE in more detail, you might want to refer to the discussion of SET/ASK PENMODE in Chapter 16.

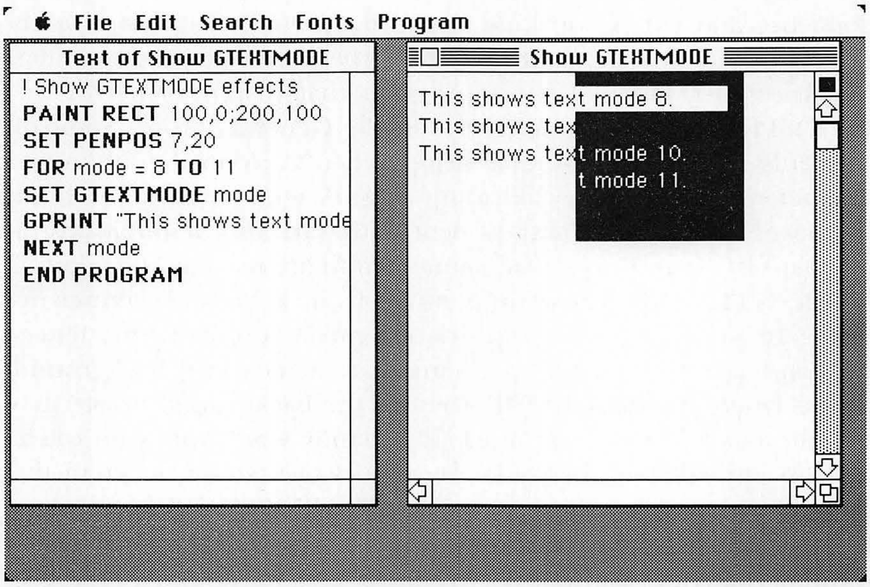
Clearing the Graphics Text Settings

■ GTEXTNORMAL

GTEXTNORMAL provides a quick way to reset all four graphics text characteristics to their original settings. It sets the text font to

Table 10-3. GTEXTMODE Settings

Setting	Name	Effect
8	Copy	Black text with white around it
9	OR	Black text without disturbing background
10	XOR	Text color is opposite of background color
11	Clear	White text without disturbing background



```
! Show GTEXTMODE effects
PAINT RECT 100,0;200,100
SET PENPOS 7,20
FOR mode = 8 TO 11
SET GTEXTMODE mode
GPRINT "This shows text mode
NEXT mode
END PROGRAM
```

```
This shows text mode 8.
This shows tex
This shows text mode 10.
              t mode 11.
```

Figure 10-2. GTEXTMODE samples

the Application font (Geneva), size 12. It also resets the GTEXT-FACE setting to 0 (plain text) and the GTEXTMODE setting to 9. GTEXTNORMAL does not have any effect on PENPOS or any of the other settings that directly affect the graphics pen. To reset those settings, you can use the PENNORMAL command that is described in Chapter 16.

```
GTEXTNORMAL ! Reset GPRINT font, size, face, and mode
```

PRINTING NUMBERS

■ SET/ASK SHOWDIGITS

When you display a number with a PRINT or GPRINT statement, Macintosh BASIC normally prints up to ten digits, with the last digit rounded if necessary. The count includes digits both to the left and to the right of the decimal point.

If a number is too large to display in ten digits, BASIC switches to scientific notation, where it displays up to ten significant digits plus an exponent. The number ten billion (10,000,000,000) has a total of eleven digits (a one and ten zeros). Because the number has more than ten digits, BASIC displays it as 1E10.

You can change the number of significant digits BASIC displays by using the SET SHOWDIGITS command followed by a number or numeric expression. You can set SHOWDIGITS as low as 1 and as high as 19. BASIC sets SHOWDIGITS to ten at the beginning of each program. You can find the current value of SHOWDIGITS with the statement ASK SHOWDIGITS followed by the name of a numeric variable.

```
SET SHOWDIGITS 5     ! Set to show only 5 digits
ASK SHOWDIGITS digs     ! Get SHOWDIGITS in digs
SET SHOWDIGITS digs+2  ! Show 2 more digits
```

SPECIFYING FORMATS

■ FORMAT$

FORMAT$ is a string function that is most frequently used in PRINT and GPRINT statements to change the format of numbers and strings before they are printed. You give FORMAT$ two arguments:

- A *format image* or "picture" of the way you want the number or string to look
- The number or string that is to be made to fit the format image.

In the statement

```
PRINT FORMAT$("###.##"; num)
```

the string "###.##" is a format image, and *num* is a variable whose value is to be formatted and printed. Notice that the arguments to the FORMAT$ function are separated by a semicolon, not a comma.

The format image may be either a string literal or a string expression. Every character in the format image represents the location of a character in the output string. If the format image is seven characters long, the output string will be seven characters

Table 10-4. Summary of FORMAT$ Characters

For numbers and strings	
#	Placeholder for one character
For numbers	
$	At beginning of image string, causes leading $ sign
+	At beginning of image string, causes leading + to be printed
.	Decimal point
,	Used before decimal point (if any), inserts a , every three positions
−	At end of image string, causes trailing minus signs
^	Holds place for digit of exponent in scientific notation
For strings	
|	Causes string to be centered in field
>	Causes string to be right-justified in field

long. With FORMAT$, any necessary extra blank spaces are added to the left end of the output string unless you specify otherwise.

The symbol # (SHIFT-3) is used to occupy a character position in the format image without conveying any other special meaning. Table 10-4 is a quick reference to the characters that have special meaning when included in a format image. The next few pages describe how to use these characters.

Formatting Numbers

The format image for a number starts with the # sign, a dollar sign, or a plus sign. It should, of course, include enough positions to allow the number, its sign, and any other characters specified in your format to be displayed. If a decimal point is included in the image, the number will be displayed with a decimal point in that exact position. Formatting 123.4 with the image "###.###" results in the string "123.400".

FORMAT$ fills any extra positions to the left of the number with blank spaces and any extra positions to the right of the decimal point with zeros. If you use a dollar sign in the field's first position, a dollar sign will appear just before the beginning of the number. If you use a plus sign in the first character position, a plus sign will appear at the beginning of positive numbers (minus signs, of course, are always printed). Using the image "$####.##" when formatting 55.2 results in the string " $55.20."

FORMAT$ will insert commas at every third position to the left of the decimal point if you put a comma in your format image. The comma must be located after the first position of the image. If your image includes a decimal point, the comma must be located before the decimal point. A single comma in the format image will cause insertion of several commas in a large number, so be sure you allow enough character positions in the field for them. Formatting 1343900 with the image "#,######.#" results in "1,343,900.0".

You can get FORMAT$ to put minus signs at the end instead of the beginning of negative numbers by using a minus sign in the last position of your format image. Scientific notation, with an exponent, is specified by using several ^ (SHIFT-6) symbols to mark the location of the exponent in the field.

The number being formatted is rounded, if necessary, to the number of decimal places your format specifies. If the number does not fill the entire field, it is right-justified so your decimal points will be lined up if you are printing a column of numbers. If the number or its exponent do not fit in the number of character positions you specify, the entire field is filled with question marks. Here are some more examples:

```
PRINT FORMAT$("##.##"; 3)  ! Prints 3.00
@PRINT FORMAT$("$#.##"; 1.888)  ! Prints $1.89
PRINT FORMAT$('+#,###.#'; 4567)  ! Prints +4,567.0
@PRINT FORMAT$('##.##^^^^'; 9888)  ! Prints 9.89E+03
PRINT FORMAT$("$##.##-"; -3.2)  ! Prints $3.20-
PRINT FORMAT$("##.#"; 478)  ! Prints ????
```

Formatting Strings

FORMAT$ gives you three choices for strings. They can start at the left end of the format image field (left-justified), be centered, or be all the way to the right with leading blanks (right-justified). If you give no special instruction, FORMAT$ returns your string left-justified.

To center the string in the field, include the symbol | in your format. To type the | symbol, hold down the SHIFT key while you press the key at the right edge of the keyboard just above the RETURN key. To right-justify the string, include the symbol > (SHIFT-.) in your format. If your string is too long to fit in the field you specify, FORMAT$ drops the right end of the string without issuing an error message.

```
PRINT FORMAT$("##|#####"; "Test")  ! Centers 'Test'
@PRINT FORMAT$('#>####'; "Test")    ! Right-justified
PRINT FORMAT$("###"; "Test string")  ! Prints 'Tes'
```

Using the Same Format for Both Numbers and Strings

You can format strings with the same format images you use for numbers. For example, you can use the same format image for a column of numbers and the title above it. When the value to be formatted is a string, FORMAT$ treats each special number-formatting character ($, +, −, ^, comma, or decimal point) as if it were a # sign. When the value being formatted is a number, FORMAT$ treats each special string-formatting symbol (| or >) as if it were a # sign. If your format includes a decimal point, the | or > symbol must be in one of the positions to the left of the decimal point.

```
@PRINT FORMAT$('$|#.##'; "Test")
    ! Prints 'Test' centered in 6-character field
@PRINT FORMAT$('$|#.##'; 37)      ! Prints '$37.00'
```

Using Multiple Format Fields

You can specify more than one field in the same format string, and you can include as many numbers and strings after the semicolon in a FORMAT$ call as you wish. The first item to be formatted is matched with the first format field, and the second item is matched with the second field. If there are more numbers and strings than format fields, the function cycles through the existing format fields until it runs out of numbers and strings to format.

Each field in a format image starts with #, $, or +. A field ends with a space or any other character that is not a format character. Non-format characters are taken as literals and are included "as is" in the string returned by FORMAT$. Formatting the number 12 with the image "T-##X" results in the string "T-12X".

A comma is only valid as a special formatting character when it is to the left of the decimal point. If a comma is encountered to the right of the decimal point, it is taken to be a literal, and it ends the field. Plus signs and dollar signs are only used as the first character of a format field, so a new field begins whenever one is encountered in your format image.

```
PRINT FORMAT$("##* "; 11,12,13)  ! Prints '11* 12* 13* '
PRINT FORMAT$("## $#.## "; 20,3,30,4)
                             ! Prints '20 $3.00 30 $4.00 '
PRINT FORMAT$("ZIP #####"; 99999)  ! Prints 'ZIP 99999'
PRINT FORMAT$('$##,>## ##'; "Amount", 1299.95)
                             ! Prints '   Amount $1,299.95'
```

ERASING THE OUTPUT DOCUMENT

- CLEARWINDOW

The CLEARWINDOW command erases the entire document in the output window. No distinction is made between text and graphics; everything is erased. If you want to erase only a part of the document, you can use the ERASE command that is described in Chapter 16. In addition to erasing the output document, CLEARWINDOW resets HPOS to 1, VPOS to 1, and PENPOS to 0,0. Thus, CLEAR WINDOW resets the page so new text will start at the top in addition to erasing the page.

```
CLEARWINDOW  ! Erases window and resets
```

CHANGING THE WIDTH OF THE TAB FIELD

- SET/ASK TABWIDTH

When you use either a comma or the TAB key (CHR$(9)) in a PRINT or GPRINT statement, the printed output resumes at the beginning of the next tab field. When your program starts running, those tab fields occur at every 100 pixels, beginning at the left edge of the output window. You can change the width of the tab fields by using the SET TABWIDTH command followed by the number or numeric expression that gives the new width. For example, SET TABWIDTH 50 sets the tab fields at every 50 pixels. The command ASK TABWIDTH, followed by the name of a numeric variable, sets the numeric variable to the current setting of TABWIDTH.

```
SET TABWIDTH 33  ! Sets tabs every 33 pixels
ASK TABWIDTH tabs%  ! Puts TABWIDTH in tabs%
```

While your program is running, each line of text is displayed according to the setting of TABWIDTH at that point in your program. But if you use the scroll bars to look through the output document, any text displayed with the PRINT command is reformatted to match the last setting of TABWIDTH. Text displayed with the GPRINT command always stays where you put it and is not rearranged later no matter how many times you change TABWIDTH. The tab fields in your program's listing window are set at every 20 pixels and are not affected by the TABWIDTH setting.

Printing on Paper

- DOCUMENT PRINT

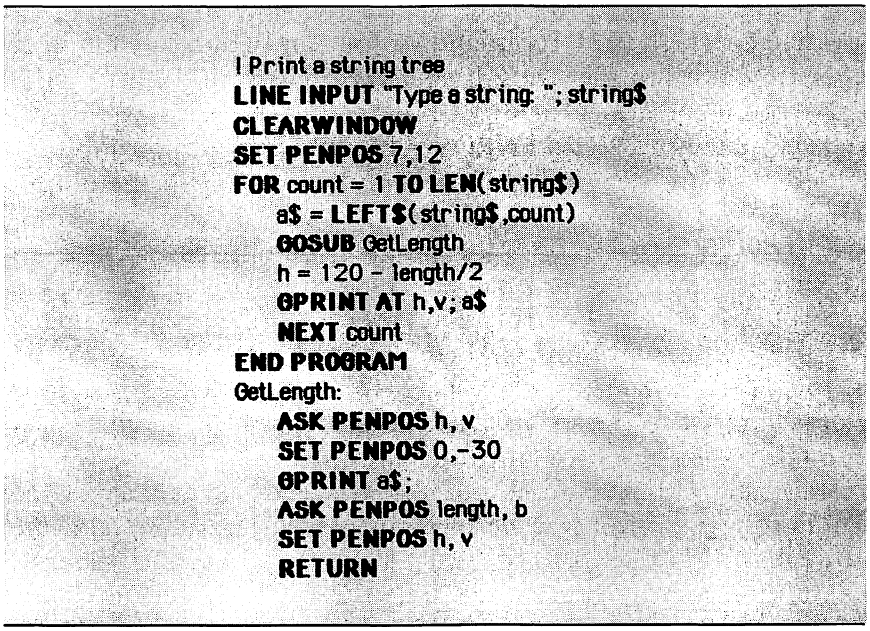
The DOCUMENT PRINT statement copies the output window to a piece of paper—assuming you have an Imagewriter or LaserWriter printer attached to your Macintosh. Everything displayed in the output window appears on the printed copy, no matter whether the information was put in the output window by a PRINT or GPRINT command or by one of the graphics commands described in Chapter 16.

Technically, DOCUMENT PRINT copies the entire document behind the output window, not just the visible area, to the printer. If you can see additional information by scrolling the output window, that information will be copied to the printer along with the information displayed in the visible area of the window.

```
PRINT "This is a test."  ! Prints in output window
DOCUMENT PRINT  ! Copies output document to printer
```

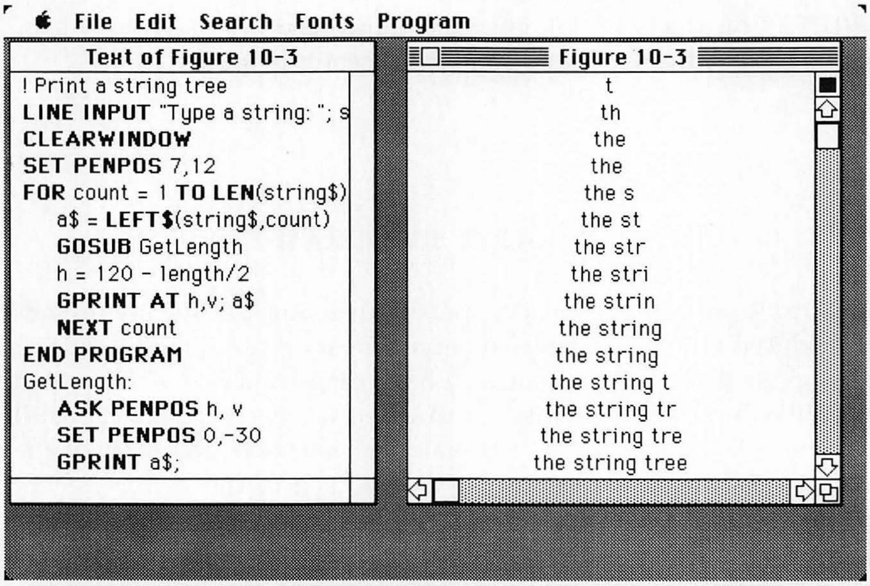
EXAMPLE PROGRAM

The example program in Figure 10-3 uses many of the commands introduced in this chapter to demonstrate a procedure for centering strings in the output window. The program takes a string typed from the keyboard and uses it to construct a string "tree," with each successive printed line containing one more character of the string until the entire string is printed.Each of the strings is centered in the output window, giving the overall look of a tree composed of the different-length strings. Figure 10-4 shows a sample of the program's output using the input string "the string tree."



```
! Print a string tree
LINE INPUT "Type a string: "; string$
CLEARWINDOW
SET PENPOS 7,12
FOR count = 1 TO LEN(string$)
    a$ = LEFT$(string$,count)
    GOSUB GetLength
    h = 120 - length/2
    GPRINT AT h,v; a$
    NEXT count
END PROGRAM
GetLength:
    ASK PENPOS h, v
    SET PENPOS 0,-30
    GPRINT a$;
    ASK PENPOS length, b
    SET PENPOS h, v
    RETURN
```

Figure 10-3. Print a string tree



```
! Print a string tree
LINE INPUT "Type a string: "; s
CLEARWINDOW
SET PENPOS 7,12
FOR count = 1 TO LEN(string$)
   a$ = LEFT$(string$,count)
   GOSUB GetLength
   h = 120 - length/2
   GPRINT AT h,v; a$
   NEXT count
END PROGRAM
GetLength:
   ASK PENPOS h, v
   SET PENPOS 0,-30
   GPRINT a$;
```

```
        t
        th
       the
       the
      the s
      the st
     the str
     the stri
    the strin
   the string
   the string
  the string t
  the string tr
 the string tre
the string tree
```

Figure 10-4. String tree output

First the program asks you to type a string from the keyboard. It uses the LINE INPUT command so you can include a comma or quotation mark in the string if you wish. Then the CLEARWINDOW command erases the input prompt and the string you typed, and the statement SET PENPOS 7,12 sets the initial location for GPRINT to the beginning of the first line of text in the output window.

A FOR/NEXT loop with the variable count as its index prints the string "tree." The loop starts by setting the variable *a$* equal to the first character of the string you typed. Next the program does a GOSUB to the subroutine GetLength, which puts the length of *a$* in the variable named *length*. Each time through the loop, one more character of the string is added to *a$* until, in the final pass through the loop, *a$* contains the entire string.

Since the output window is 240 pixels wide, the center of the window is at pixel 120. The program sets the variable *h* to the value that will cause a$ to be centered horizontally in the window, 120 minus half the length of *a$*. The GPRINT AT statement sets the pen to the position just calculated and prints *a$*. The FOR/NEXT loop is repeated until the program is finished.

The key to this program is the subroutine GetLength, which obtains the length of the string *a$*. Because different characters are not the same width in any type font except Monaco (and even in Monaco the widths can vary depending on the type size chosen), the width of the string cannot be calculated directly from the number of characters in the string.

The method used is to print the string using GPRINT at an invisible location and then check the pen location to see how long the string is. SET PENPOS 0,−30 puts the pen above the output window, and ASK PENPOS length,b gets the length of the string after printing. The ASK PENPOS h,v statement at the start of the subroutine saves the original pen location, and the SET PENPOS h,v statement at the end restores the pen. Saving and restoring the pen location ensures that the subroutine can be used safely from any point in your program without causing any change in the location of your printed output.

PRACTICE EXERCISES

1. What statement would you use with the PRINT command to cause the next output to be at the beginning of line 3? What statement would you use with the GPRINT command (assume 12 point type)?
2. Can you write a routine to print the integers from 1 to 5 on a single line spaced 20 pixels apart? Have your routine save and restore the values of any system parameters that are changed.
3. Write a short program that prints "Hello" in 24 point Outline type in the New York font.
4. Can you specify a format that prints dollar amounts up to $10 million? The format should provide for insertion of commas where they are appropriate and should also center any text if it is used to print a string.

Chapter 11

Defining Your Own Functions

Commands:

- DEF, FUNCTION
- EXIT FUNCTION, END FUNCTION

In addition to the large variety of predefined functions described throughout this book, Macintosh BASIC has an extremely powerful capacity to handle user-defined functions. This is your chance to redefine the BASIC language to include all the special functions you always thought a language should have. With a little imagination, you can even use defined functions to create your own language.

Macintosh BASIC provides a simple way to define functions in a single line and a slightly more complicated way to define functions as miniature programs using more than one line. Most of the rules about defined functions apply equally to both types.

SINGLE-LINE FUNCTIONS

■ DEF

Any formula that can be written in one line and that gives a single result can be defined as a single-line function. The definition of a single-line function begins with the keyword DEF followed by a space, the name of the function, an equal sign, and the formula that determines the value of the function. If the function takes arguments, variable names representing the data types of the arguments are listed in parentheses just after the function name. The names in parentheses are called *parameters*. The following function named Celsius takes one argument, represented by the parameter *x*:

```
DEF Celsius(x) = (x-32) * 5 / 9
```

You can use a defined function in your program by using its name in a program statement, just as you use the functions that are already a part of the BASIC language. If the function definition has parameters, you replace each parameter with the value you want the function to use. Each time you use the function in a program statement, BASIC executes your function definition and then returns to your program. In Figure 11-1, the main program calls the function Celsius with the variable *degrees* as the argument. When BASIC executes the function Celsius, it substitutes the value of the degrees in the formula for the parameter *x*.

```
! Convert degrees Fahrenheit to Celsius
DO
    INPUT "Degrees Fahrenheit: "; degrees
    PRINT "That's "; Celsius(degrees); " Celsius."
LOOP
END PROGRAM
DEF Celsius(x) = (x-32) * 5 / 9
```

Figure 11-1. Convert degrees Fahrenheit to Celsius

It doesn't matter where you put the function definitions in your program. If a function definition is encountered in the flow of program execution, BASIC jumps around the function definition and continues executing the next valid program statement. Function definitions are processed when the program is compiled and do not need to be executed to become operative.

FUNCTION NAMES

Function names are governed by the same rules as variable names, including the use of special characters at the end of the name to indicate the type of the function's value. The function Celsius, for example, returns a double-precision real value. If you want it to return a short integer value, you could add % to make its name Celsius%. You cannot use the identical name for both a variable and a function. For safety, you should not use the name of a function for either a simple variable or an array variable.

When a function or variable name is encountered while executing your program, BASIC checks whether the name matches a function. If the name does match, BASIC executes the function. If there is a variable with the same name as the function, BASIC may execute the function when you want to access the variable, or it may store the function result in the variable. In either case, you will not get the correct result.

PARAMETER PASSING

A function receives information from the rest of your program through the parameters that are listed in parentheses after the function's name. Only one piece of information, the function result, is passed from the function back to the program that called it.

When you are writing a function definition, you can use the value of any variable from the calling program, with the exception of variables with the same name as those appearing in the parameter list. If you use variables that are not parameters, you should place a comment just after the function definition to remind yourself that the function depends on variables that are not in the parameter list. One of the significant benefits of defining your own functions is that you can use them in other programs. However, if your function uses variables that are not parameters, the function may not work in another program unless that program also sets the same variables.

Defined functions in Macintosh BASIC can accept any number of parameters. You specify the data type of each parameter in the function's definition by using the appropriate last character for the parameter's name. You can, of course, define a function that uses no parameters. When you use a function in your program, you will receive an error message if you use an incorrect number of arguments or if any of the arguments is of a data type that is incompatible with the corresponding parameter in the function's definition.

When BASIC executes a defined function, the values of all the arguments being passed to the function are copied. Any references to the parameters from inside the function are to these copies, not to the original variables in the main program. This ensures that no program statement inside the function can inadvertently affect the value of any variables outside the function. The copies are erased when execution returns to the main program.

MULTIPLE-LINE FUNCTIONS

■ FUNCTION, END FUNCTION

Single-line functions can do little more than compute a value and return. Multiple-line functions, however, can use all of the BASIC commands. If you redefined the Celsius function from Figure 11-1 as a multiple-line function, it would look like this:

```
FUNCTION Celsius(x)
Celsius = (x-32) * 5 / 9
END FUNCTION
```

The definition of a multiple-line function begins with the keyword FUNCTION, and the first line of the definition contains nothing else except for the function's name and parameters. The last line of a multiple-line function definition is the statement END FUNCTION. Between these two lines, you can use as many program statements as you wish. One of those statements must set the function's name equal to a value. If no statement gives a value to the function, a numeric function will return 0, a string function will return the null string, and a Boolean function will return false.

Figure 11-2 shows a program that uses a multiple-line function. This function finds the maximum of two numbers. The name of the function, Max, does not end in a special character so the function returns a double-precision real number.

The mechanics of the function definition are relatively simple. The function has two numeric parameters. An IF/THEN/ELSE/ ENDIF statement tests whether the first argument is larger than the second and sets the value of the function equal to the larger argument. Note that when you set the value of the function, you do not include any parentheses or parameters with the function name on the left of the equal sign.

Multiple-line functions communicate with programs as single-line functions do. Programs can pass information to the function through the parameters, and the only value passed back to the calling program is the function result. The function has access to the calling program's variables, but not to any variable with the same name as one of the parameters in the function definition statement.

```
! Find the maximum of two numbers
DO
    INPUT "Please type a number: ";x
    INPUT "Please type another: ";y
    PRINT "The maximum is "; Max(x,y)
LOOP
END PROGRAM
FUNCTION Max(a,b)
IF a > b THEN
        Max = a
    ELSE
        Max = b
    ENDIF
END FUNCTION
```

Figure 11-2. Find the maximum of two numbers

You can include any legal BASIC statement as part of a multiple-line function. If the statements in a function definition print, use sound, store a value in a variable, or write to a file, those actions will occur every time the function name is mentioned in the main program.

Getting Out Early

■ EXIT FUNCTION

Macintosh BASIC includes an EXIT FUNCTION statement. When executed, this statement causes an immediate exit back to the program that called the function. Be sure you set the value of the function before executing the EXIT FUNCTION statement. If you do not, the function will return the same value it returned the last time you called it.

Like the EXIT DO and EXIT FOR statements, EXIT FUNCTION can be abbreviated as just EXIT. However, if you use the short form EXIT inside a DO loop or FOR/NEXT loop, it will cause an exit from only that loop instead of from the entire function. To eliminate a source of potential confusion and program errors, you should always use the long form, EXIT FUNCTION. Here is an example of an early exit from a function that calculates the number of hours remaining until midnight:

```
FUNCTION time.left( hour )
time.left = 0
IF hour > 24 THEN EXIT FUNCTION
time.left = 24 - hour
END FUNCTION
```

Using Variables in Functions

You can use LET statements and implied LET statements to assign values to variables inside multiple-line functions. However, you should be careful. The variables that receive values in the function affect your main program unless they are listed as parameters in your function definition statement. Parameters are *local* to the function; all other variables are *global,* that is, they affect other parts of your program.

If your main program uses a variable named x and a function stores a value in *x,* the next time the main program uses *x* it will be using the value stored by the function. To minimize the likelihood that programs could have variable names identical to those used by a function, the names of variables used in the functions in this book will all start with the letter "z."

Do not make the mistake of using the function name as if it were a variable name. It is not. Even though you store a value in the function name (just as you do into a variable name), you should not try to read that value while still in the function. If you need to save a value, use a parameter or variable name.

Writing a Recursive Function

A *recursive function* is a function that calls itself. Sometimes this is the most concise or clear way to define a particular function. For instance, the factorial of a number is defined in mathematics as the number multiplied by all integers smaller than itself down to 1. The factorial of the number N is N times the factorial of (N−1). The factorial of three can be expressed as

```
3! = 3*2
   = 3*2*1
   = 6
```

This is the kind of relationship that you can easily define with a recursive function. Figure 11-3 shows a program that uses a factorial function that is defined recursively. The function and its parameter are declared as extended-precision because factorials can be very large numbers.

Note that the definition of a recursive function always contains actions to be taken in two cases: the recursive case, which is usually executed, and the non-recursive case, which is executed when the function reaches its simplest value. If the ending case is not included in its definition, a recursive function will keep calling itself and never return to the program that called it. The function

```
INPUT a
PRINT factorial\(a)
END PROGRAM
FUNCTION factorial\(x\)
IF x\ > 1 THEN factorial\ = x\ * factorial\(x\-1)
IF x\ < 2 THEN factorial\ = x\
END FUNCTION
```

Figure 11-3. Factorial function defined recursively

will keep calling itself until it uses all of the computer's available memory or until you grow tired of waiting and stop the program.

Recursive functions provide an excellent way to learn about the concept of recursion and can provide a reasonably neat and understandable way to define some functions. However, recursive functions often require more memory and execution time than simpler functions.

In order to be recursive, a function must allocate space for copies of its parameters each time it is called. If the function is called a hundred times, as the function in the factorial example in Figure 11-3 would be if you typed "101" in response to the prompt, the recursive function calls would use enough space in the computer's memory for 100 copies of the function's parameters. A function that calls itself hundreds of times can fill up the available memory in a hurry.

In addition to causing an Out of Memory condition if the recursive function calls itself too many times, the copying of the function's parameters and the overhead involved in repetitively entering and exiting the function take time. The program in Figure 11-4 uses a FOR/NEXT loop instead of a recursive statement to define a factorial function. It uses extended precision, just like the program in Figure 11-3. The recursive function in Figure 11-3 takes well over five seconds to calculate and print the factorial of 400, while the non-recursive function in Figure 11-4 takes less than one second to print the same value.

```
INPUT a
PRINT factorial\(a)
END PROGRAM
FUNCTION factorial\(x\)
y\ = x\
FOR zindex = x\-1 TO 1 STEP -1
y\ = y\ * zindex
NEXT zindex
factorial\ = y\
END FUNCTION
```

Figure 11-4. Factorial function defined with FOR/NEXT loop

SOME EXAMPLE FUNCTIONS

The rest of this chapter provides a series of example functions that you can use in your own programs. You can keep these and other functions you write in a special program file on your disk. When you need one of the functions in a program you are writing, you can copy it from that "function library" file and paste it into your program. Let the functions defined here spark your imagination. You can write a defined function for almost any formula or series of actions you may need in your programs.

String Functions

You often need to search for one string inside another. Instead of writing code to do each individual search, you can use the function InStr, shown in Figure 11-5. This function is similar to the string search function found in several other versions of BASIC.

The first parameter, *startpos%,* is an integer that indicates the character position in the longer string where you will start searching. A value of 1 for *startpos%* will tell the function to search the entire string. The second parameter is the string to be searched,

```
FUNCTION InStr(startpos% ,string$,lookfor$)
InStr =0
IF LEN(lookfor$) < 1 THEN InStr = startpos%
FOR zposn = startpos% TO LEN(string$) + 1 -LEN(lookfor$)
    IF lookfor$ = MID$(string$,zposn,LEN(lookfor$)) THEN
        InStr = zposn
        EXIT FUNCTION
    ENDIF
NEXT zposn
END FUNCTION
```

Figure 11-5. String search function

and the third parameter is the string you want to find. The function returns the character position at which the string being sought begins, or zero if that string was not found. The function call InStr(1,"test.please",".") returns 5, the character position of the period in the string "test.please."

Another common function is one that generates a string of identical characters such as spaces or asterisks. The function String$ in Figure 11-6 is a function that fulfills this need. Its arguments are the length of the string you want and the character it is to contain. The call String$(5,' ') returns a string containing five spaces.

```
FUNCTION String$(length% ,char$)
zc$ = LEFT$(char$,1)  ! Make sure only one character
zstr$ = ""   ! Start with empty string
IF length% > 0 THEN
    FOR zcounter = 1 TO length%
        zstr$ = zstr$ & zc$
    NEXT zcounter
ENDIF
String$ = zstr$
END FUNCTION
```

Figure 11-6. String$ function

```
DEF area.circle(radius) = PI*radius*radius
DEF circumference(radius) = 2*PI*radius
DEF area.triangle(width,height) = width*height/2
```

Figure 11-7. Three numeric functions

Numeric Functions

Many short formulas are easily converted into functions. Using the function name instead of a formula in the main program makes the logic of the program easier to follow. Figure 11-7 shows three

```
  ! Conversion from degrees to radians
DEF radians(degrees) = (PI/180)*degrees
  ! Normal trigonometric functions
DEF secant(x) = 1/COS(x)
DEF cosecant(x) = 1/SIN(x)
DEF cotangent(x) = 1/TAN(x)
DEF arcSin(x) = ATN(x/SQR(1-x*x))
DEF arcCos(x) = 1.5708-ATN(x/SQR(1-x*x))
DEF arcSecant(x) = ATN(x/SQR(x*x-1))+SGN(SGN(x)-1)*1.5708
DEF arcCosecant(x) = ATN(x/SQR(x*x-1))+(SGN(x)-1)*1.5708
DEF arcCotangent(x) = ATN(x)+1.5708
  ! Hyperbolic functions
DEF sinh(x) = (EXP(x)-EXP(-x))/2
DEF cosh(x) = (EXP(x)+EXP(-x))/2
DEF tanh(x) = EXP(x)/(EXP(x)+EXP(-x))*2+1
DEF secanth(x) = 2/(EXP(x)+EXP(-x))
DEF cosecanth(x) = 2/(EXP(x)-EXP(-x))
DEF cotangenth(x) = EXP(x)/(EXP(x)-EXP(-x))*2+1
DEF arcSinh(x) = LOG(x+SQR(x*x+1))
DEF arcCosh(x) = LOG(x+SQR(x*x-1))
DEF arcTanh(x) = LOG((1+x)/(1-x))/2
DEF arcSecanth(x) = LOG((SQR(1-x*x)+1)/x)
DEF arcCosecanth(x) = LOG((SGN(x)*SQR(x*x+1)+1)/x)
DEF arcCotangenth(x) = LOG((x+1)/(x-1))/2
```

Figure 11-8. Useful trigonometric functions

```
FUNCTION LeapYear~(year)
IF year MOD 4 = 0 THEN
        LeapYear~ = TRUE
    ELSE
        LeapYear~ = FALSE
ENDIF
END FUNCTION
```

Figure 11-9. Leapyear~ function

functions that return the area and circumference of a circle and the area of a triangle.

The only trigonometric functions built into the BASIC language are sine, cosine, tangent, and arctangent. All of the other trigonometric functions can be defined from these four functions and other predefined BASIC functions. Figure 11-8 gives definitions for these additional trigonometric functions.

Boolean Functions

When your program needs to take different actions depending on the result of some test, a Boolean function is often appropriate. For example, your main program would be much easier to understand if a test read IF LeapYear~(1983) than if it read IF 1983 MOD 4 = 0. The function LeapYear~ as defined in Figure 11-9 makes the first phrasing possible.

PRACTICE EXERCISES

1. Figure 11-1 defines a function that converts the temperature from degrees Fahrenheit to degrees Celsius. Can you define a function that converts from degrees Celsius to degrees Fahrenheit?
2. Figure 11-2 defines a function that returns the maximum of two numbers. Can you write an integer function that returns the smaller of two integers?
3. Write a string function that has one string parameter and replaces all periods in the string with commas.
4. Can you write a recursive function that takes one integer argument and returns a string of asterisks the length specified by the argument?

Chapter 12
Using Files

Statements:
- OPEN #, CREATE #
- INPUT #, LINE INPUT #, PRINT #
- READ #, WRITE #, REWRITE #
- SET/ASK CURPOS #, SET/ASK HPOS #
- SET/ASK EOF #
- CLOSE #, CLOSE

Functions:
- TYP (#), ATEOF (#)

Positions:
- BEGIN, END, RECORD, SAME, NEXT

Contingencies:
- IF MISSING~, IF THERE~, IF EOR~, IF EOF~

This chapter describes the BASIC statements that your program uses to store data in disk files. Before you start to use the commands presented in this or the next chapter, be sure to copy all of your important files to a backup disk. Never put a disk with your only copy of an important file into the disk drive while you are experimenting with commands that write on disks.

WHAT IS A FILE?

A *file* is a collection of information arranged in a preestablished format. Files usually reside on a disk or some other device outside your machine. The name of each file is stored in a directory on the disk and is displayed under the file's icon in the directory window when you are in the Finder.

The data in some files is subdivided into smaller units called *records*. If records are divided into even smaller units, those units are called *fields*.

Organization of a File

Macintosh BASIC allows you to organize each file in one of three ways: as a series of data that is ordered from beginning to end (sequential), as a sequence of numbered records (random-access), or as a continuous stream of data (stream). This chapter describes sequential and random-access files, which are used to store data. Stream files are most often used to communicate with another computer or an external device like a modem or printer. They are discussed in the next chapter.

A sequential file, sometimes called a serial file, contains information stored one byte after another. It is the most common type of file because its structure is simple. A sequential file structure contains no gaps in the middle of the stored information, so it is relatively compact. Figure 12-1 shows the organization of fields and records in a typical sequential file.

Macintosh BASIC assumes that a file is organized as a sequential file unless you tell it otherwise. You should use sequential file organization whenever you plan to read the information in the same order in which you wrote the information to the file.

A random-access (RECSIZE) file is a sequence of fixed-length, numbered records. It is called a random-access or relative file because you can get to any record directly by using its index

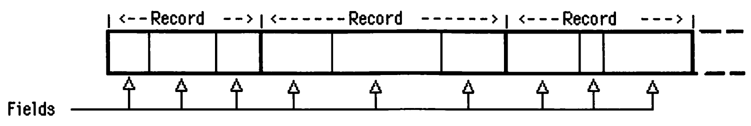


Figure 12-1. Fields and records in a sequential file

number. You declare the size of the fixed-length records when you open the file. The length of the file on disk is always the fixed length times the number of records in the file, even if all the records are empty. Figure 12-2 shows the organization of records and fields in a typical random-access file.

Random-access files are less compact than sequential files. When you write information to a record, Macintosh BASIC always writes the full length of the record, padding it with ASCII 0 characters if necessary. In addition, if you write a record with an index number past the end of the file, Macintosh BASIC creates empty records to fill the gaps in the file. For example, if the last record in your file is record 3 and you write record 50, Macintosh BASIC creates empty records for records 4 through 49. Random-access files are appropriate when you need instant access to information in any part of your file.

Macintosh BASIC uses a file pointer to keep its place in each sequential or random-access file. The pointer tells BASIC where in the file the next read or write operation is to take place. Macintosh BASIC updates the pointer after each operation; it also provides

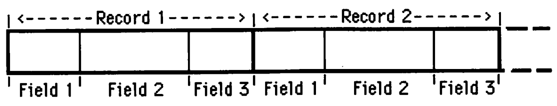


Figure 12-2. Fields and records in a RECSIZE file

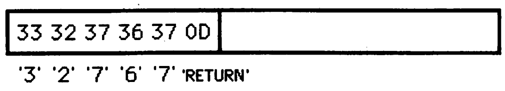


Figure 12-3. TEXT data format

commands that let you change the pointer if you want to move to a different place in the file for your next operation.

Types of Data in Files

Files differ from each other in the type of data they hold. The two major types of file data are text (TEXT) and binary (BINY). Macintosh BASIC assumes you want a file to hold text data unless you tell it otherwise. Figure 12-3 shows a short text file that contains the integer 32767.

Text data in a file is made up of any series of characters. Each character is stored in the file as an ASCII code, so each character takes one byte of storage space. A text file may be subdivided into records and fields. Records are separated from each other by carriage return characters, ASCII 13. Fields within records are separated from each other by commas or Tab characters, ASCII 9. When calculating the length of a file or record, you need to remember that the Tab and carriage return characters occupy a byte each time they occur.

Binary (BINY) data is a series of bytes. When BASIC writes to a binary file, data from the Macintosh's internal memory is copied to the file without any field or record markers. You are responsible for keeping track of the types of the variables from which the information comes. You can read information from the file into any type of variable, but it won't make any sense unless you read it into the same type of variable from which it came.

Figure 12-4 shows a short binary file that contains the integer 32767. Binary data is usually more compact than text data, so it

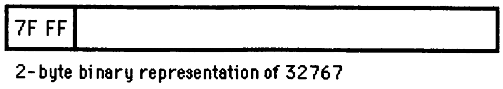


Figure 12-4. BINY data format

usually takes less time to read or write a binary file than a text file with the same amount of information. When you write a variable to a binary file, BASIC copies the variable exactly as it is stored in memory. In text format, the number 32767 occupies five bytes, one for each character, as shown in Figure 12-3. In binary format, the same number occupies only two bytes if it is stored in an integer variable. You can use a binary file whenever you have information of a fixed length that you want to store in a compact fashion.

A DATA type file is a special kind of binary file with a one-byte code before each item. The code is called a *type tag,* and its value identifies the type of variable for the next item in the file. Except for the fact that you can examine the type tag to learn what type of value is stored next, you handle DATA files just like BINY files. You can use a DATA type file any time you can use a BINY file. Figure 12-5 shows a DATA file that contains the integer 32767.

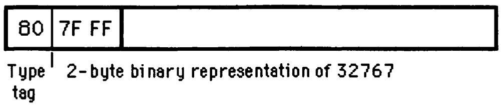


Figure 12-5. DATA data format

CREATING A FILE

Macintosh BASIC lets you create a file or using an existing file. In order for you to be able to write or read information to and from the file, the file must be open. Opening a file is analogous to opening the drawer of a file cabinet and pulling out a folder. In this section you will see how to create a file and open it for use by your program.

Communications Channels

When you open or create a file, Macintosh BASIC establishes a communications channel between your program and the file. Each communications channel has a number, which you assign when you open the channel. You use the channel number to identify the file in every program statement that gets information from the file or sends information to it.

You can have as many as seven channels to different files open at the same time. You give each channel a number from 1 to 32767. Channel 0 is reserved for input from the Macintosh keyboard and output to your program's output window. It is always open and does not count as one of your seven channels.

File Names

File names can be 63 characters long. You can use any printable character except the colon (:) in the name of a file. When looking for a file name, Macintosh BASIC ignores any differences between upper- and lowercase letters.

Files are organized into groupings called *volumes*. Each diskette contains one volume. The volume's name is the title that appears under the diskette's icon on the desktop. The files on larger storage devices, such as hard disks, may be divided among several volumes. A volume name can be 27 characters long and can contain any printable character except the colon. You can include the name of the disk volume on which the file is located as the first part of a file name if you wish. If you do include the volume name, use a colon to separate the volume name from the file name.

Here are some example file names:

```
theFile
Volume1:theFile
master file
```

Opening a File

■ OPEN #, CREATE #

The OPEN# and CREATE# commands open files. OPEN# opens a file that already exists, and CREATE# opens a new file. The two commands use the same syntax:

```
OPEN #channel : "filename"     ! Opens existing file
CREATE #channel : "filename"   ! Opens new file
```

Follow the word OPEN or CREATE with a space, the # sign, and a channel number. The channel number can be a number, a numeric variable, or a more complicated numeric expression. You use a colon after the channel number and then supply the file's name. The file name can be a string literal or a string expression.

The OPEN# and CREATE# statements establish a communications channel to a file and set the characteristics of both the channel and the file. After the file name you may include optional keywords to set the file's organization, the type of data, and the direction of information flow in the channel. Each optional keyword that you use is preceded by a comma. You can use optional keywords in any order.

The optional keywords for the file's organization are SEQUENTIAL, RECSIZE, and STREAM, which correspond to the three ways of organizing a file that were discussed earlier in this chapter. (Stream files are discussed in the next chapter.) When you specify a random-access file, you declare the size of the fixed-length records by putting the length just after the keyword RECSIZE. The optional keywords for type of data are TEXT, BINY, and DATA, which correspond to the types of data described earlier in this chapter. If you do not specify the type of data, Macintosh BASIC opens the file as a sequential file holding text data.

```
OPEN #1: "theFile", SEQUENTIAL, TEXT
OPEN #33: "old file", RECSIZE 40, BINY
CREATE #9: "new file"       ! Assumes sequential text
OPEN #8: "reading file"     ! Assumes input sequential text
```

The direction of information flow in each communications channel is set when you open the channel. You can set the channel to allow your program to read or to both read and write. The keyword INPUT sets the channel to read only. The keywords OUTIN and APPEND set the channel to handle information flow in both directions. If you do not use an access keyword, Macintosh BASIC opens the file with access INPUT.

The access keywords also determine the initial setting of the file pointer that BASIC uses to keep its place in each file. INPUT and OUTIN set the file pointer to the beginning of the file. APPEND sets the file pointer to the end of the file, which is where you want it if you are going to add more information to an existing sequential file.

```
OPEN #1: "theFile", SEQUENTIAL, TEXT, APPEND
OPEN #33: "old file", RECSIZE 40, BINY, OUTIN
CREATE #9: "new file", APPEND  ! Assumes sequential text
OPEN #8: "reading file"  ! Assumes input sequential text
```

READING AND WRITING FILES

Macintosh BASIC provides two sets of commands that transfer information between a file and your program. You use INPUT#, LINE INPUT#, and PRINT# with text data (of type TEXT). You use READ#, WRITE#, and REWRITE# with binary data (either BINY or DATA type).

Reading Information From a Text File

- INPUT #, LINE INPUT #

INPUT# works like the INPUT command that handles the text you type on the keyboard. After the word INPUT, a space, and the # sign you supply the channel number that you assigned to the file in the OPEN# statement, followed by a colon and then the name of the variable that is to receive a value. You can list more than one variable in a single INPUT# statement if you separate the variable names by commas.

```
INPUT #8: item1
INPUT #8: item1, item2, item3$
```

As with the regular INPUT statement, the types of variables you list must match the data being input or BASIC will generate an error message. The INPUT# statement fills one variable from each field in the file. Fields end with commas, Tab characters, or Return characters.

As it fills each variable, Macintosh BASIC advances the file pointer to the beginning of the next field in the file. When it has filled all the variables listed in an INPUT# statement, Macintosh BASIC checks whether the file pointer is at the beginning of a record. If the pointer is not already at the beginning of a record, Macintosh BASIC moves the file pointer to the beginning of the next record in the file unless the input list ends with a comma. (If the list ends with a comma, Macintosh BASIC leaves the file pointer in its current position.) If an INPUT# statement tries to read values past the end of a record or past the end of a file, Macintosh BASIC generates an error message.

LINE INPUT# is very similar to INPUT#. After the # sign you supply the channel number you assigned to the file in the OPEN# statement, followed by a colon and then the name of the variable that is to receive a value. The LINE INPUT# statement can contain only one variable name. Each LINE INPUT# statement reads an entire record, including any commas and Tab characters it may contain.

```
LINE INPUT #8: line1$
```

Putting Information in a Text File

- PRINT #

To add information to a text file, use the keyword PRINT followed by a space, the # sign, the file's channel number, and a colon. The rest of the PRINT# statement is the same as the PRINT statement that displays information in your program's output window. In fact, the PRINT statement is just a PRINT# statement that always sends its information to channel 0.

```
PRINT #1: item1
PRINT #1: item1,item2,item3$
PRINT #1: "Input sees this";" as one string."
```

In the standard PRINT statement, you use a comma to move across the screen to the next tab setting. When the PRINT# statement encounters a comma, a Tab character (ASCII 9) is stored in the file and becomes a field separator. The field separator is used by subsequent INPUT# statements to separate the items being read from the file. Just as an extra comma in a PRINT statement skips an extra tab field in the output window, an extra comma in a PRINT# statement creates an extra empty field in the file.

You use semicolons to separate items in a PRINT statement list. Just as a semicolon in a regular PRINT statement does not write any characters to the output window, a semicolon in a PRINT# statement does not send any characters to the file. Thus, if you print two items separated by semicolons, there will be nothing in the file to separate them. A subsequent INPUT# statement will attempt to read these two items as a single item.

If you end a PRINT# statement with a comma or a semicolon, the current record will remain open, and the next PRINT# statement will add to the same record. If the PRINT# statement does not end with a comma or semicolon, Macintosh BASIC sends a carriage return character to end the record. If your file was opened as a RECSIZE file, BASIC generates an error message if a PRINT# statement tries to output more text than will fit in a single fixed-length record.

When you send information to a file that already contains information, you need to be aware of the location of the file pointer. When you open a file with access OUTIN, the file pointer is at the beginning of the file. If you print text without moving the file pointer from the beginning of the file, you will overwrite the existing information. It is much safer to open the file with access APPEND, which puts the pointer at the end of the file where it is safe to add information.

Reading and Writing Binary Information

- READ #, WRITE #, REWRITE #

READ# is used instead of INPUT# to read binary information. WRITE# and REWRITE# are used instead of PRINT# to write binary information. You can use WRITE# to send BINY or DATA information to a sequential file or to write a new random-access record. You use REWRITE# to write to a previously written random-access record. If you have several values to read or write, you separate the items in the value list with commas. The number of bytes taken from or sent to the file matches the number of bytes occupied by the type of variable used.

```
READ #33: item1   ! Reads 8 bytes (double precision)
READ #33: item1%,item2%   ! Writes 4 bytes (2 for each integer)
WRITE #33: item1\,item2%   ! Writes 12 bytes (10 extended, 2 integer)
REWRITE #33: item1  ! Writes 8 bytes (double precision)
```

When you are working with a random-access file, you must use WRITE# only for new records and REWRITE# to replace the information in existing records. WRITE# gives an error if you try to use it on an existing record, and REWRITE# gives an error if the record is not already there. Macintosh BASIC makes this distinction to protect you from inadvertently overwriting existing information. Each WRITE# or REWRITE# statement in your program writes a separate record to the file unless you end the statement with a comma. When you end your WRITE# or REWRITE# statement with a comma, you keep the current record open, and the next WRITE# or REWRITE# statement will add to the same record.

DATA Type Binary Files

- TYP(#)

The DATA type file contains binary data with a special one-byte type tag before each value to indicate the type of variable from which it came. If you use the DATA format in a random-access file, remember to allow an extra byte for each value's type tag. The

Table 12-1. DATA File Type Tags

Type Tag	Type of Data	Symbol	Number of Bytes
0	Integer	%	2
2	String	$	length+2
4	Extended-precision real	\	10
5	Single-precision real	|	4
6	Double-precision real	(none)	8
7	Computational (long integer)	#	8
12	Boolean	~	1
13	Character	©	1

possible values for type tags and the corresponding variable types are listed in Table 12-1. Each record in a DATA type file ends with an ASCII 0.

You can use a DATA file in the same way you use a BINY file. Use the READ# command to read data from the file and the WRITE# and REWRITE# commands to write data to the file. The only difference is that with a DATA file Macintosh BASIC checks the type tags against the variable names when reading data and generates a type mismatch error if the types do not match. You can use this type checking to help find subtle programming errors.

DATA type files can also be used in a more sophisticated way to allow different types of data to be mixed together in the same file without a predefined ordering. You could store each different type of data in a different type of variable. When you write to the file, each type of data would have its own unique type tag.

When you read data from such a file, you need to look at the type tag before reading each item to find out which kind of data it contains. You use the TYP function to do this. TYP is followed by parentheses containing the # sign and the channel number. It returns the type tag number for the next value in the file. The TYP function returns a value of −1 if the file pointer is not pointing to a type tag (this happens at the end of a random-access record or at the end of the file). You can use an IF or CASE statement to check the type tag and read the value into a variable of the appropriate type.

```
IF TYP(#33) = -1 THEN PRINT "Out of data"
IF TYP(#33) = 12 THEN READ #33: Boolean~
SELECT CASE TYP(#33)
    CASE 0: READ #33: integer%
    CASE 2: READ #33: string$
END SELECT
```

CHANGING POSITION IN A FILE

You can change the position of the file pointer at the beginning of any of the input or output statements. As an example, the format of an INPUT# statement that positions the file pointer looks like

INPUT #1, *position* : *variable$*

where *position* is one of the positioning keywords described in the following sections. Macintosh BASIC repositions the file pointer before it executes the input or output command.You can move the file pointer in any sequential or random-access file.

Moving to the Beginning or End

■ BEGIN, END

BEGIN and END can be used with any command that sends or receives information through a channel. BEGIN moves the file pointer to the beginning of a file, and END moves the pointer to the end of a file. BASIC moves the file pointer as specified before it executes the rest of the program statement.

```
INPUT #8, BEGIN: item1
PRINT #1, END: item1, item2
```

BEGIN is most commonly used in INPUT#, LINE INPUT#, or READ# statements to reread a file from the beginning. END is most commonly used in PRINT# or WRITE# statements to add to a file after reading part of its contents.

Choosing a Record in a Random-Access File

■ RECORD

RECORD is used only with random-access (RECSIZE) files. You use it in file-handling statements the same way you use BEGIN and END. RECORD, followed by the index number of a record, positions the file pointer to the beginning of that record. Calculating a record number and using it with RECORD is the normal way to find a record in a random-access file.

```
READ #33, RECORD 17: item1
REWRITE #33, RECORD 17: item3
```

Choosing a Specific Record

■ SAME, NEXT, SET/ASK CURPOS #

SAME moves the file pointer back to the start of the most recently referenced record. If the file pointer is not already at the beginning of a record, NEXT moves it to the beginning of the next record. NEXT does not move the file pointer if it is already at the beginning of a record. You use SAME and NEXT in file-handling statements the same way you use BEGIN and END.

```
READ #33, SAME: item1,item2
REWRITE #33, SAME: item1,item3
READ #33, NEXT: item1
```

You can find out the present location of the file pointer by using the ASK CURPOS followed by a space, the # sign, the file's channel number, a comma, and a numeric variable name. The SET CURPOS# command provides an alternative way to move the file pointer. Both of these commands are used in separate program statements, not in the file-handling statement. CURPOS is short for CURrent POSition. CURPOS for a random-access file is the index number of the current record; for a sequential file, it is the byte position in the file, with the first byte located at position 0. You can use SET CURPOS# only if the file's channel is set for access OUTIN or APPEND.

```
ASK CURPOS #33, variable
SET CURPOS #1, byte
```

Locating Data Within a Record

- SET/ASK HPOS #

HPOS# is the character position within a file record. The first character in a record is in position 0. SET HPOS# is only useful when you have fields of known length inside the records of a random-access file and you want to access a field without reading all the fields that precede it. Follow SET HPOS with a space, the # sign, the channel number, a comma, and the number of the character position you want to set. You can use SET HPOS# only if the file's channel is set to allow output (access OUTIN or APPEND). ASK HPOS is followed by a space, the # sign, the file's channel number, a comma, and the name of the numeric variable in which you want Macintosh BASIC to store the HPOS value.

```
ASK HPOS #33, byte
SET HPOS #1, byte
```

Length of the File

- SET/ASK EOF #

You can use EOF# to obtain or set the length of a file. The Keyword EOF is followed by a space, the # sign, and the number of an open channel. With ASK EOF#, you follow the channel number with a comma and a numeric variable name. With SET EOF, you follow the channel number with a comma and a numeric expression for the new file length. When you want to change the length of a file, you will first need to use ASK EOF# to get the current length of the file and then use SET EOF# to set the new length.

```
ASK EOF #33, records
SET EOF #33, records-1 ! Cuts off last record
```

The length of a sequential file is the number of characters or bytes in the file. The length of a random-access file is the number of records in the file. If you use a number smaller than the current file length with SET EOF#, you will truncate the file and lose all the information after the new file ending. If you use a higher number, you lengthen the file. You can only use SET EOF# if the channel to the file was opened with access OUTIN or APPEND.

CHECKING FOR SPECIAL CONDITIONS

You can specify an action for Macintosh BASIC to take if certain special conditions, called *contingencies,* arise during file operations. This allows your program to retain control instead of having BASIC generate an error message and stop the program.

To specify a contingent action in a file-handling statement, insert a comma, the test for the contingency, and the statement to be executed if the test is true just before the colon. The format of the contingency test in READ# and WRITE# statements looks like this:

READ #1, IF *contingency* THEN *statement*: income$
WRITE #3, *position,* IF *contingency* THEN *statement*: outgo$

The statement to be executed if the condition is true must be a single Macintosh BASIC statement. The only colon in the program line should be the colon before the beginning of the values list.

If you are setting the file pointer and testing for a contingency, the contingency should be second, since that is the order in which they are executed. Macintosh BASIC first executes any positioning keyword, then checks the contingency, and finally performs the input or output operation. If you have more than one value or variable listed after the colon, BASIC checks the contingency before each individual input or output operation. If the contingency test is true, BASIC executes your contingency statement, skips the rest of the input/output statement, and continues with the next program statement. None of the contingencies are used with PRINT#.

Conditions Related to Records

■ IF MISSING~, IF THERE~

IF MISSING~ and IF THERE~ are contingencies that test whether a value is present in the field or record pointed to by the file pointer. You use IF MISSING~ with INPUT#, LINE INPUT#, READ#, and REWRITE# to check whether the field or record required by the file statement is missing. You use IF THERE~

with WRITE# to prevent WRITE# from overwriting existing data. These two tests cannot be used with a BINY sequential file because that kind of file does not contain identifiable fields or records.

```
READ #33, IF MISSING~ THEN PRINT "Help!": item
WRITE #33, IF THERE~ THEN CALL DoRewriteInstead: item
! Read a whole text file
DIM item$(size)
DO
    INPUT #1, IF MISSING~ THEN EXIT DO: item$(i)
    i = i + 1
LOOP
```

Checking for End of Record

- IF EOR~

IF EOR~ tests whether the file pointer is located at the end of a record. You can use it with INPUT# or READ#. You cannot use IF EOR~ with LINE INPUT#. LINE INPUT# always reads an entire record and always leaves the file pointer at the beginning of the next record.

```
READ #33, IF EOR~ THEN GOSUB HandleError: item1
```

Checking for End of File

- IF EOF~, ATEOF~(#)

IF EOF~ is used with INPUT#, LINE INPUT#, READ#, and REWRITE# to check for an end of file error before performing the input or output operation. If the file pointer is at the end of the file, your contingency statement is executed.

```
LINE INPUT #1, IF EOF~ THEN GOSUB FileEnd: line$
```

ATEOF~ is a Boolean system function. Unlike all the other contingencies described in this section, ATEOF~ can be used in any program statement except file-handling statements. ATEOF~ returns TRUE if the file pointer is at the end of the file and

FALSE if it is not. ATEOF~ takes one argument, the channel number of the file to be tested, in parentheses with the # sign.

```
IF ATEOF~(#1) THEN PRINT "Done."
```

PUTTING FILES AWAY

■ CLOSE #, CLOSE

CLOSE, followed by a space, the # sign and a channel number, closes the file, updates the file on the disk if it has been changed, updates the disk directory, and releases the channel. You can close all your open files and channels at once by using CLOSE with no channel number. BASIC does not close channel 0 (input from the keyboard and output to the output window). To reuse a closed file, open it again with another OPEN statement.

```
CLOSE #1  ! Closes channel 1
CLOSE     ! Closes all open files
```

EXAMPLE PROGRAM

The Note Pad desk accessory is handy for keeping lists. Sometimes it would be handier, however, if you could read the entire contents of the Note Pad into another program all at once instead of cutting and pasting it one page at a time through the Clipboard. The example program in Figure 12-6 makes this possible.

The Note Pad desk accessory stores the information you type in a separate file named the Note Pad File. The icon for this file is usually located in the System Folder in your directory window.

The Note Pad File has a slightly unusual organization, which does not exactly match either the SEQUENTIAL or the RECSIZE formats used by Macintosh BASIC. The Note Pad File contains a record 256 bytes long for each of the Note Pad's eight pages. Each record contains everything you have typed on the corresponding page of the Note Pad, including any Returns or other unusual characters. A byte containing the ASCII code 0 is stored in each record just after the end of the valid information.

```
! Note Pad Copier
! Copies Note Pad File into another file
PRINT "NOTE PAD COPIER"
PRINT "Please give a name for the copy"
INPUT "file: "; file$
OPEN #1: "Note Pad File", TEXT, RECSIZE 256
CREATE #2: file$, OUTIN
FOR page = 1 TO 8
    LINE INPUT #1, RECORD page-1: line$
    FOR i = 1 TO LEN(line$)
        IF ASC(MID$(line$,i,1)) = 0 THEN
            line$ = LEFT$(line$,i-1)
            EXIT FOR
        ENDIF
    NEXT i
    PRINT #2: line$
NEXT page
CLOSE
PRINT "The file has been copied."
END PROGRAM
```

Figure 12-6. Note Pad copier

The program in Figure 12-6 copies the Note Pad File to a regular sequential text file. When it starts, the program asks you to type the name of the file you want to create to hold the copy of the Note Pad File. The program then opens the Note Pad File as a random-access file with 256-byte records containing TEXT data. The program uses a CREATE statement to open your output file so you will not destroy any existing file with the same name. It uses access OUTIN because you will put information into the file.

The statements inside the FOR/NEXT loop are executed once for each of the Note Pad's eight pages. The LINE INPUT#1 statement positions the file at each page's record (the record numbers start at 0, so the record is one less than the page number) and then reads the entire record into the string variable *line$*.

The program then uses a FOR/NEXT loop to look at each character starting at the beginning of the string in *line$*. When the

0 that indicates the end of the valid information is found, the program truncates the string in *line$* at that point and executes the PRINT#2 statement.

The PRINT#2 statement puts the entire string in *line$* into your copy file. Note that the LINE INPUT# statement reads the entire 256-byte RECSIZE record into the string variable *line$*, no matter what that record contains. In fact, the variable will include everything you typed on that page, including the carriage return characters that mark the end of a line. The PRINT# statement writes all of these characters to the sequential text file. When you read the sequential text file later with LINE INPUT# statements, the carriage return characters will cause Macintosh BASIC to put each line from the Note Pad into a separate variable.

PRACTICE EXERCISES

1. What statement would you use to open a channel to add information at the end of an existing sequential text file named "gift list"?
2. How would you open a new random-access file named "new-file" with text records 30 characters long?
3. How would you get the value of an integer named *integer%* and a string named *string$* from record 23 of a binary REC-SIZE file named "datafile"?
4. Can you write a loop that searches for and reads the first record that begins with a string in a DATA RECSIZE type file? Assume the file is connected to channel 12 and was opened for access OUTIN.

Chapter 13

Files, Volumes, and Devices

Commands:

- RENAME, DELETE, LOCK, UNLOCK
- GETFILEINFO, SETFILEINFO
- SETVOL, EJECT, GETVOLINFO
- DEVCONTROL #, DEVSTATUS #

Functions:

- GETFILENAME$, GETVOLNAME

Devices:

- .AIN, .AOUT, .BIN, .BOUT, .SOUND, .PRINTER

This chapter describes how you use Macintosh BASIC to manipulate whole files and the volumes on which files are stored. It also shows how you can use the file-handling commands to open channels and communicate with physical devices like printers and modems attached to the ports on the back of your Macintosh.

OPERATIONS ON FILES

Macintosh BASIC makes it very easy to perform operations like renaming and deleting files from a BASIC program. In addition to those operations, you can get a list of all the files on a disk volume, move a file from one Macintosh application program to another, and change some of the file information that the Finder keeps.

Renaming a File

- RENAME

You can change the name of a file by using the RENAME command followed by the name of the existing file, a comma, and the new name for the file. Both names can be string literals, string variables, or string expressions. If the new file name is already being used, BASIC generates an error and does not rename the file.

```
RENAME "old", "new"  ! Changes the name of file "old" to "new"
```

Deleting a File

- DELETE

The DELETE command followed by a file name removes the file with that name from the disk or volume directory. The file name can be a string literal, variable, or expression. BASIC generates an error if it cannot find the file.

```
DELETE "oldfile"  ! Throws oldfile away
```

When it executes the DELETE statement, Macintosh BASIC deletes the file immediately. You do not get a chance to change your mind, as you do in the Finder. BASIC's DELETE command is the same as putting the file in the trash and emptying the trash all in a single statement. BASIC does not provide any way to recover a deleted file. The only files you are likely to delete from within a program are temporary files that the program creates.

Locking and Unlocking Files

■ LOCK, UNLOCK

Locking a file is a way to protect it from being destroyed inadvertently. You can lock a file from the Finder by selecting the file's icon, choosing Get Info from the File menu, and clicking on the box labeled Locked. Once you lock a file that way, the Finder will not let you put the file in the trash unless you unlock the file first. However, that method of locking a file does not prevent you from changing the file by writing in it.

BASIC provides an even better way to lock a file. When you lock a file from BASIC, you cannot throw the file in the trash from the Finder, delete the file from a BASIC program, or write anything into the file from a BASIC program. Locking a file prevents you from executing operations that might destroy or change the information in the file. You can still open, read, and copy a locked file just as you would any other file.

To lock a file from BASIC, just use the keyword LOCK followed by the name of the file. To unlock the file, use the word UNLOCK followed by the file name. The file name can be a string literal, string variable, or string expression.

```
LOCK "file1"   ! Locks the file named file1
LOCK a$        ! Locks the file whose name is in a$
UNLOCK "file1" ! Unlocks the file named file1
UNLOCK a$      ! Unlocks the file whose name is in a$
```

If you want to change the status of a file from the keyboard while you are running Macintosh BASIC, you can type the LOCK or UNLOCK command in an untitled program window and run it without affecting anything else.

If you open a locked BASIC program file and make changes in the program, you cannot save the changes in the same file until you unlock it. This allows you to protect the master copy of your program. You can, of course, select Save a Copy In to save the changed version of the program in another file.

Listing Existing Files

■ GETFILENAME$

GETFILENAME$ is a string function that returns the name of a file. The most common uses of GETFILENAME$ are to list the files on a volume and to search for a file with a particular name. GETFILENAME$ takes one argument, which is a number ranging from 1 to the number of files in the volume directory. GETFILENAME$ returns an empty string if you give it an argument that is larger than the number of files.

```
filename$ = GetFileName$(count)
PRINT GetFileName$(1)
```

The matching between numbers and file names is quite arbitrary and does not always stay the same. The order of the files can change whenever your program executes a CREATE, RENAME, or DELETE statement.

Since GETFILENAME$ returns an empty string when its argument exceeds the number of file names, you can obtain the names of all the files on a volume by starting with GETFILENAME$(1) and incrementing the argument by 1 until the function returns an empty string. When you get the empty string, you have obtained the names of all files on that volume. The program in Figure 13-1 lists all files on the current volume.

```
! Display List of Files on Disk
count = 1
DO
    file$ = GetFileName$(count)
    IF file$ = "" THEN EXIT DO
    PRINT file$
    count = count + 1
LOOP
END PROGRAM
```

Figure 13-1. Display list of files on disk

Getting Information on a File

■ GETFILEINFO

GETFILEINFO obtains information that the Macintosh operating system keeps about a file and places that information in 48 consecutive bytes in memory. You reserve space for the information by dimensioning an array large enough to hold the 48 bytes. Integers occupy two bytes each, so an integer array with a dimension of 23 is large enough if you use the 0th element.

You follow GETFILEINFO with the name of the file, a comma, the @ sign, and the name of an array element. The 0th element of the array is frequently used as the beginning of the storage area, but that is not required. You must, however, have at least 48 bytes in the array.

The @ sign creates a pointer that gives the GETFILEINFO routine the address of the array element. A thorough discussion of pointers is contained in Chapter 19. For now, you can think of the @ sign followed by the array element as meaning "put the information into memory starting with the address of the specified array element."

```
DIM a%(23)  ! Integer array with 48 bytes
GETFILEINFO "whatfile", @a%(0)
```

The block of data stored in the 48-byte area by GETFILEINFO actually contains several different data types. GETFILEINFO puts information there as a single block of binary data, and you have to do the calculations necessary to get each part of the data into a usable form. The array you dimension to reserve space can actually be of any variable type, but you can retrieve and interpret the data most easily if you use an integer array. Table 13-1 lists the contents of the data block with the number of bytes occupied by each item.

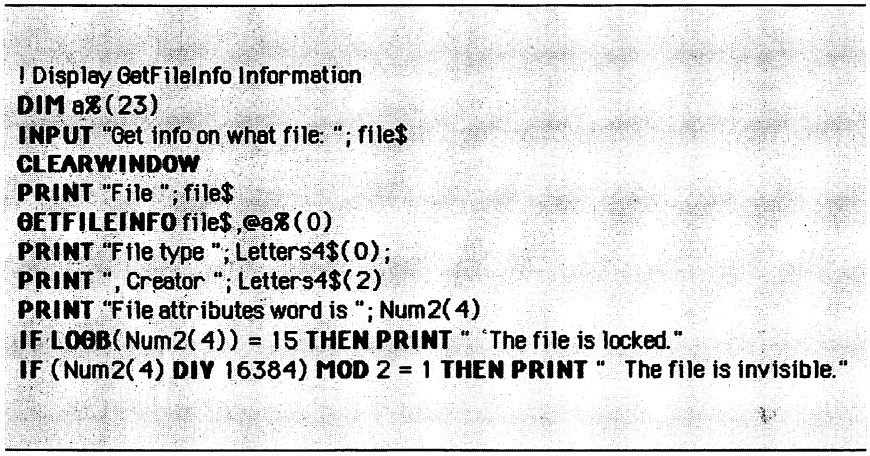
The program in Figure 13-2 decodes and displays the information in the GETFILEINFO data block. In addition to displaying the information, the program in Figure 13-2 serves as an example of a way to extract information from the data block so you can use it in your BASIC programs.

The first two items of information about each file are its file type and creator. These are codes made up of four letters. File types and creator codes are discussed in detail later in this chapter.

Table 13-1. GETFILEINFO Information

Byte Offset	Length (bytes)	Description
0	4	File type
4	4	Creator
8	2	File attribute flags
10	2	Icon's vertical location
12	2	Icon's horizontal location
14	2	Folder class code
16	4	* File number in directory
20	2	* Block number on disk where data starts
22	4	* Data logical EOF—(same as ASK EOF(#)
26	4	* Data physical EOF (actual file length)
30	2	* First block in resource fork
32	4	* Logical EOF of resource fork
36	4	* Physical EOF of resource fork
40	4	Date and time file created
44	4	Date and time file modified

* Items marked with asterisks cannot be changed with SETFILEINFO.



```
! Display GetFileInfo Information
DIM a%(23)
INPUT "Get info on what file: "; file$
CLEARWINDOW
PRINT "File "; file$
GETFILEINFO file$,@a%(0)
PRINT "File type "; Letters4$(0);
PRINT ", Creator "; Letters4$(2)
PRINT "File attributes word is "; Num2(4)
IF LOGB(Num2(4)) = 15 THEN PRINT "  The file is locked."
IF (Num2(4) DIV 16384) MOD 2 = 1 THEN PRINT "  The file is invisible."
```

Figure 13-2. Display GetFileInfo information

```
PRINT "Icon location:"
PRINT "   Vertical "; a%(5);
PRINT ", Horizontal "; a%(6)
PRINT "Folder code "; a%(7);
SELECT CASE a%(7)
    CASE < 0: PRINT " (on the desktop)"
    CASE = 0: PRINT " (in directory window)"
    CASE > 0: PRINT " (in a folder)"
    END SELECT
PRINT "File number "; Num4(8)
PRINT "Data starts at block "; Num2(10)
PRINT "Data logical EOF is "; Num4(11)
PRINT "Data actual EOF is "; Num4(13)
PRINT "Resources start at block "; Num2(15)
PRINT "Resources logical EOF is "; Num4(16)
PRINT "Resources physical EOF is "; Num4(18)
PRINT "File Creation "; Num4(20)
PRINT "File Modification "; Num4(22);
END PROGRAM
FUNCTION Letters2$(first)
Letters2$ = CHR$(a%(first) DIV 256) & CHR$(a%(first) MOD 256)
END FUNCTION
FUNCTION Letters4$(first)
Letters4$ = Letters2$(first) & Letters2$(first+1)
END FUNCTION
FUNCTION Num4(first)
Num4 = Num2(first) * 65536 + Num2(first+1)
END FUNCTION
FUNCTION Num2(first)
    IF a%(first) < 0 THEN
        Num2 = a%(first) + 65536
    ELSE Num2 = a%(first)
    ENDIF
END FUNCTION
```

Figure 13-2. Display GetFileInfo information (*continued*)

After the file type and creator code, the data block contains two bytes that contain coded information about the file. This coded information is technically referred to as the *file attribute flags.*

Only two pieces of this information are very useful to the BASIC programmer. One piece tells you whether or not the file was locked from the Finder. The other useful piece of coded information tells you whether the file is an invisible one. When a file is marked as invisible, the Finder does not display any information about it and will not open it. The DeskTop file, whose name often appears as the result of the function call GETFILENAME$(1), is an example of an invisible file.

The next two fields of the data block are the vertical and horizontal coordinates of the file's icon in the directory window. The top left corner of the directory window is (0,0), and the coordinates increase as the icon moves down and to the right. If the file is located in a folder, the coordinates correspond to the icon's location in the folder's directory window.

The sixth field is the file's folder class code. If the folder class code is 0, the file is located in the main directory window. If the folder class code is less than zero, the file's icon is located on the desktop, not in any window. If the folder class code is greater than zero, the file is in a folder, and the code is the number of the folder in which the file is located. Folder numbers seem to be assigned at random; however, all the files in a single folder have the same folder number.

The next field in the data block is the file's number in the volume directory. This number corresponds to the argument for GETFILENAME$. If you supply the file number as an argument to GETFILENAME$, you get the file's name.

After the file number there are three fields that describe the location and length of the file's data on the disk. The first field indicates the location on the disk where the file's data starts. The next field gives the total length of the data, that is, the position of what is called the *logical end of file.* This number is the same number you retrieve with the ASK EOF# statement. The third field contains the total amount of space on the disk reserved for the file's data, which corresponds to the position of the physical or actual end of file. Disk space is allocated in 512-byte blocks. If the logical end of file is not an integer multiple of 512, the physical end of file will be the next higher integer multiple of 512.

The next three fields contain the first disk location, the position of the logical end of file, and the position of the physical end of file for the file's resource fork, the hidden portion of a file used by the operating system. You will not need this information unless you are making very sophisticated use of the Macintosh toolbox routines described in Part Four of this book. Resources are described in Chapter 23.

The last two fields contain the raw data from which the Finder calculates the dates and times the file was created and last modified. Each of these fields contains a large positive integer. The integer is the number of seconds between the creation or modification time and 12:00 A.M. January 1, 1904.

File Types and Creators

The file type and creator codes you can read with GETFILEINFO connect data files with the application programs that created them. Each application program has a unique four-letter creator code assigned to it by Apple. This creator code is contained in the creator field of the application program file and in all the files the program creates.

When you work with file types and creators, be sure to copy upper- and lowercase letters exactly. For two file types or creators to be equal, the case of each letter must match exactly. The file type "TeXT", for instance, is not the same as the file type "TEXT", because the first type contains a lowercase "e."

The Finder uses the creator codes to determine which icon shape to display for each file. The Finder also uses the creator code when you open a data file to determine which application program to load with the file. You can change a data file's icon and the application program that is loaded with it by changing the creator field. You should never change the creator field of an application program file because doing so would interfere with the program identification system and might prevent the program from opening its own data files.

Major application programs often have several different types of files associated with them. Macintosh BASIC, for instance, stores BASIC program files in both text and binary formats and lets BASIC programs create three types of data files (TEXT, BINY, and DATA). The file type field that you can read with GETFILEINFO allows application programs to distinguish between the different types of files.

Most application programs use the file type field to select the file names that are displayed in the dialog window when you select Open from the File menu. Macintosh BASIC, for instance, displays the names of its program files (types BTXT and BCOD), but does not display any of the three types of data files. Table 13-2 lists the file types and creators used by some of the most common Macintosh applications.

Changing File Information

■ SETFILEINFO

SETFILEINFO changes information about a file. SETFILEINFO uses a 48-byte data block, just as GETFILEINFO does. SETFILEINFO uses the information from the 48-byte block you supply to update the file information. The best way to get the correct information in the 48-byte block is to use GETFILEINFO first to put the existing file information into your array. Then you can modify the values in the array and use SETFILEINFO to make the necessary changes to the file information.

```
DIM a%(23) ! Integer array with 48 bytes
GETFILEINFO "whatfile", @a%(0)
! Here is where you make your changes
SETFILEINFO "whatfile", @a%(0)
```

BASIC does not allow SETFILEINFO to change the file number, the starting blocks of the data and resource portions of the file, or the logical and physical end of file values. These fields are marked with asterisks in Table 13-1. You must include these fields in the 48-byte data block you supply for SETFILEINFO, but BASIC does not change the existing information. You can change the data logical end of file value with the SET EOF# command.

If you make a mistake using SETFILEINFO, you can make a file inaccessible, so it is a good idea to make a backup copy of your disk before you start experimenting with SETFILEINFO.

Table 13-2. Some File Types and Creators

File Type	Creator	Description of File
APPL	DONN	Macintosh BASIC
BTXT	DONN	Macintosh BASIC Program Text
BCOD	DONN	Macintosh BASIC Program Binary Format
TEXT	DONN	Macintosh BASIC Text File
BINY	DONN	Macintosh BASIC Binary File
DATA	DONN	Macintosh BASIC Binary DATA File
APPL	PASC	Macintosh Pascal
TEXT	PASC	Macintosh Pascal Program Text
APPL	MACA	MacWrite
WORD	MACA	MacWrite "Entire Document" File
TEXT	MACA	MacWrite "Text Only" File
APPL	MPNT	MacPaint
PNTG	MPNT	MacPaint document
APPL	MSBA	Microsoft BASIC (decimal math)
TEXT or MSBA	MSBA	Microsoft BASIC Program Text
MSBB	MSBA	Microsoft BASIC Program Compressed (Binary)
MSBP	MSBA	Microsoft BASIC Program Protected
TEXT or (blank)	(blank)	Microsoft BASIC Text File
APPL	MSBB	Microsoft BASIC (binary math)
TEXT	MSBB	Microsoft BASIC Program Text
MSBC	MSBB	Microsoft BASIC Program Compressed
MSBD	MSBB	Microsoft BASIC Program Protected
ZSYS	MACS	Note Pad File
APPL	FMOV	FontMover
FFIL	FMOV	Font File

Moving Files Between Applications

Often you will create a file with one application and then want to use it with another. Perhaps you received the text of a program via a modem and now want to run the program in Macintosh BASIC. Or perhaps your BASIC program created a text file that you now want to edit using MacWrite. Or perhaps you want to display a plain document using the official System file icon.

You can handle all of these situations by changing the file type and creator of the file, provided the file meets one additional requirement: the organization of the file and the data in it must be compatible with the new file type and creator. The file type describes the file's organization and data. It does no good to change the file to a new creator and file type if the file does not contain information in the format the new application can read.

Many applications use ordinary text files. Even many applications that use other file formats allow you to save the information in a simple text file. If this alternative is available, you should use it before you try to change the file type or creator. You should be able to change a normal text file from one application to another without suffering any complications, as long as the new application is capable of reading an ordinary text file.

Ordinary text files often have the file type TEXT. By looking at the file types in Table 13-2, you can see that Macintosh BASIC, Macintosh Pascal, MacWrite, and Microsoft BASIC are all capable of generating and reading files of this type. In addition, the file type BTXT that Macintosh BASIC uses for program files is the same as a TEXT file except that it may contain a few extra characters to remind BASIC when to turn the boldface on and off.

When you change a file's type or creator, you are responsible for knowing whether the contents of the file are compatible with the new file type and creator. BASIC does not warn you if you are about to make an error, so it is a good idea to work with a copy of the file so you will not lose the information if you make a mistake.

The program in Figure 13-3 is a general purpose program that changes a file's file type and creator. To use the program, you need to type three things: the name of the file, the new file type, and the new creator.

After you type the file name, the program uses GETFILEINFO to fill the 48-byte data block. Then it displays the file's current file

```
! Change File Type and Creator
DIM a%(23) ! Integer array with 48 bytes
PRINT "CHANGE FILE TYPE AND CREATOR"
INPUT "What file to change: "; file$
PRINT "File "; file$
GETFILEINFO file$, @a%(0)
PRINT "File type "; Letters4$(0);
PRINT ", Creator "; Letters4$(2)
DO
    INPUT "New file type: "; type$
    INPUT "New creator: "; creator$
    IF LEN(type$)=4 AND LEN(creator$)=4 THEN EXIT DO
    PRINT "4 letters in each answer, please!"
LOOP
! Store new type and creator
a%(0) = ASC(LEFT$(type$,1)) * 256 + ASC(MID$(type$,2,1))
a%(1) = ASC(MID$(type$,3,1)) * 256 + ASC(RIGHT$(type$,1))
a%(2) = ASC(LEFT$(creator$,1)) * 256 + ASC(MID$(creator$,2,1))
a%(3) = ASC(MID$(creator$,3,1)) * 256 + ASC(RIGHT$(creator$,1))
SETFILEINFO file$, @a%(0)
PRINT "File type and creator have"
PRINT "been changed to "; type$; " "; creator$
END PROGRAM
FUNCTION Letters2$(first)
Letters2$ = CHR$(a%(first) DIV 256) & CHR$(a%(first) MOD 256)
END FUNCTION
FUNCTION Letters4$(first)
Letters4$ = Letters2$(first) & Letters2$(first+1)
END FUNCTION
```

Figure 13-3. Change file type and creator

type and creator. If you do not want to change either the file type or creator, reenter the type or creator exactly as it is displayed (remember, you must match the upper- and lowercase letters).

The only error checking the program does is to make certain that your new file type and creator each contain exactly four letters. Once you have correctly entered the type and creator, the program stores the characters that make up the type and creator names into the first eight bytes of the data block and uses SETFILEINFO to change the file.

To change a plain text file into a Macintosh BASIC program file, you type BTXT for its file type and DONN for its creator. To turn a text file created by BASIC into a MacWrite document, use file type TEXT and creator MACA. To display a plain document file using the System file icon, you make its creator MACS. If you enjoy working with file types and creators, you will probably want to supplement Table 13-2 by keeping your own list of file types and creators for different types of files and applications.

Saving a Damaged MacWrite File

If you are lucky, you will never receive the message "This document can't be opened." If you are not so lucky, however, you may see this message when you try to reopen a document you saved earlier. MacWrite versions 2.20 and earlier display this message when your Macintosh does not have enough memory to open a document and also when the MacWrite control codes in the document have been damaged.

You can often open a damaged MacWrite file by changing the file type from WORD to TEXT. Try this change as a last resort to avoid retyping all the information in the file. Essentially, you are telling MacWrite to read everything in the file as text, ignoring the distinction between MacWrite control codes and text. If you succeed in convincing MacWrite to read the file as a text file, you will then need to manually delete all of the odd control code characters.

WORKING WITH VOLUMES

Macintosh BASIC provides several commands to help you manipulate volumes, which contain groups of files. It is common to think of files as residing on disks with each disk containing one volume. If you have a hard disk or other mass storage device, however, you will learn that there can be more than one volume on the same device. If you use file and volume names, you do not usually need to know what kind of physical device actually contains the stored information.

When you turn on your Macintosh and insert a disk, the operating system records the name of the disk volume from which the system is started. This volume becomes the first preset, or default, volume. It remains the default volume until you change it. Unless you include a volume name as the first part of a file name, every file operation is performed on the current default volume.

Listing the Volumes

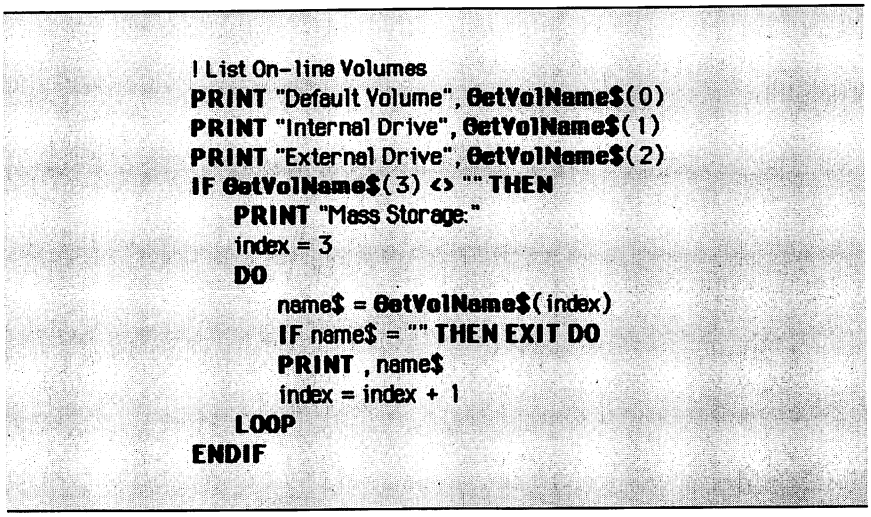
■ GETVOLNAME$

GETVOLNAME$ is a string function that returns the name of a volume. It takes one argument, a number that tells BASIC which volume name you want. Table 13-3 lists the particular volumes that correspond to each argument. An argument of 0 causes GETVOLNAME$ to return the name of the current default volume. An argument of 1 gets the name of the volume in the internal disk drive, 2 gets the name of the volume in the external disk drive, and 3 gets the name of the first volume on a hard disk or other mass storage device. If a hard disk contains more than one volume, the remaining volumes will be numbered sequentially from 4 upward.

GETVOLNAME$ returns an empty string if there is no volume available on the device for the given argument. You can use this fact to learn whether a particular storage device is currently available. To find out whether there is a disk in an external disk drive, for instance, call GETVOLNAME$ with the argument of 2. If the function returns a volume name with length greater than 0, a disk is available in an external drive. The program in Figure 13-4 displays a list of volumes and devices. This list will not include a disk drive that has no disk inserted.

Table 13-3. GETVOLNAME$ Volume Numbers

Number	Device
0	Default volume
1	Volume in internal disk drive
2	Volume in external disk drive
3	First volume on hard disk



```
! List On-line Volumes
PRINT "Default Volume", GetVolName$(0)
PRINT "Internal Drive", GetVolName$(1)
PRINT "External Drive", GetVolName$(2)
IF GetVolName$(3) <> "" THEN
    PRINT "Mass Storage:"
    index = 3
    DO
        name$ = GetVolName$(index)
        IF name$ = "" THEN EXIT DO
        PRINT , name$
        index = index + 1
    LOOP
ENDIF
```

Figure 13-4. List on-line volumes

Setting the Default Drive

■ SETVOL

SETVOL changes the preset, or default, volume or disk drive. SETVOL is short for SET VOLume. You can supply either a number or a string after the word SETVOL. If you supply a number or numeric expression, BASIC assumes it is a volume number corresponding to the numbers in Table 13-3. If you supply a string or string expression, BASIC assumes it is a volume name. In either case, BASIC generates a "No such volume" error if it cannot find a volume with the specified name or in the specified drive.

```
SETVOL 1  ! Sets internal drive as default
SETVOL "Master disk"
```

Changing Disks

■ EJECT

The EJECT command ejects a disk from the internal or external disk drive. You can follow the word EJECT with a drive number (1 for the internal drive, 2 for the external drive) or the name of the volume you want to eject. You cannot eject a volume from a hard disk or other mass storage device.

```
EJECT 1  ! Ejects disk from internal drive
EJECT "Master disk"
```

Getting Information About a Volume

- GETVOLINFO

The GETVOLINFO statement is very similar to the GETFILE-INFO statement. GETVOLINFO obtains information about a volume instead of a file. You follow GETVOLINFO with the name of the volume, a comma, and a pointer to an array element. The array must contain at least 36 bytes beginning with the element to which you point.

```
DIM a%(17)  ! Integer array with 36 bytes
GETVOLINFO "whatvol", @a%(0)
```

Macintosh BASIC lets you use any type of array, but an integer array is best for interpreting the data. If you do not know the name of the volume, you can get the name with the GETVOLNAME$ function. Table 13-4 lists the items of information that GETVOL-INFO stores in your array.

The program in Figure 13-5 decodes and displays the information in the GETVOLINFO data block. In addition to displaying the information, the program serves as an example of how to extract information from the data block so you can use it in your BASIC programs.

The descriptions in Table 13-4 adequately describe most of the items in the GETVOLINFO data block. Only a few of these items appear to be useful in BASIC programs. The dates and times the volume was initialized and last modified are expressed in seconds since 12:00 A.M. January 1, 1904. Two useful bits of information you can extract from the volume attributes word indicate whether the volume was locked by software or write-protected. If you know how many bytes a new file will occupy, you can find out whether there is room for it on the volume by dividing the file length by

Table 13-4. GETVOLINFO Information

Byte Offset	# Bytes	Description
0	2	Volume index number assigned by system
2	4	Date and time volume was initialized
6	4	Date and time volume directory last modified
10	2	Volume attributes
12	2	Number of files listed in volume directory
14	2	First block of volume directory
16	2	Number of blocks in volume directory
18	2	Number of blocks on entire volume
20	4	Number of bytes in each block
24	4	Minimum bytes to allocate for a file
28	2	Block where data storage begins
30	4	Number for the next file created
34	2	Number of unused blocks on the volume

```
! Display GetVolInfo Information
DIM a%(17)    ! Integer array with 36 bytes
v$ = GETVOLNAME$(1)
GETVOLINFO v$,@a%(0)
PRINT "Volume ";a%(0); ", "; v$
PRINT "Created "; Num4(1)
PRINT "Modified "; Num4(3)
PRINT "Attributes word is "; Num2(5).
IF LOGB(Num2(5)) = 15 THEN PRINT "Volume is locked in software."
IF LOGB(Num2(5) MOD 256) = 7 THEN PRINT "Volume is write-protected."
PRINT Num2(6); " files on the volume."
PRINT "Directory starts at block "; Num2(7)
PRINT Num2(8); " blocks in the directory."
PRINT Num2(9); " blocks on the volume."
PRINT "Size of each block is "; Num4(10); " bytes."
PRINT "Minimum allocation is "; Num4(12); " bytes."
```

Figure 13-5. Display GetVolInfo information

```
PRINT "File storage starts at block "; Num2( 14)
PRINT "Next file number is "; Num4( 15)
PRINT Num2( 17); " free blocks."
END PROGRAM
FUNCTION Num4(first)
Num4 = Num2(first) * 65536 + Num2(first+1)
END FUNCTION
FUNCTION Num2(first)
    IF a%(first) < 0 THEN
        Num2 = a%(first) + 65536
    ELSE Num2 = a%(first)
    ENDIF
END FUNCTION
```

Figure 13-5. Display GetVolInfo information *(continued)*

the block size and comparing the result to the number of unused blocks.

The volume information is maintained by the operating system. Changing it without the system's knowledge could jeopardize the integrity of the system and cause you to lose all the files on the disk. For that reason BASIC does not provide any command to change the volume information.

COMMUNICATING WITH DEVICES

You can connect external devices to the Macintosh through several plugs, or ports, on the back of the machine. This section shows how you can use the file-handling commands to open channels and communicate with those devices from your BASIC programs.

Standard Macintosh Devices

- .AIN, .AOUT, .BIN, .BOUT, .SOUND, .PRINTER

Table 13-5 lists the standard Macintosh devices available from Macintosh BASIC. The modem port (labeled with the telephone

Table 13-5. Standard Macintosh Device Drivers

Device	Driver	Access
Modem port	.AIN	INPUT or OUTIN
	.AOUT	OUTIN or APPEND
Printer port	.BIN	INPUT or OUTIN
	.BOUT	OUTIN or APPEND
Sound	.SOUND	OUTIN or APPEND
Printer	.PRINTER	OUTIN or APPEND

icon) and the printer port (labeled with the printer icon) are sometimes called serial ports A and B, respectively. The A and B ports are each treated as two devices, one for input and another for output.

.SOUND controls both the device inside the Macintosh that generates sound and the plug on the back of the machine that is labeled with the music note. If you have a printer connected to the printer port, you can use it from BASIC by referring to it as the .PRINTER device.

Each port is controlled by a small software program called a *device driver.* The drivers for most of the ports are contained in the Macintosh's permanent memory. You can control an external device from a BASIC program by opening a channel to its driver.

The last column of Table 13-5 shows the access direction you must use for each device. The access for .AIN and .BIN must be INPUT or OUTIN, and the access for .AOUT and .BOUT must be APPEND or OUTIN. Access for .SOUND and .PRINTER should always be APPEND or OUTIN.

Using Devices as Stream Files

A *stream* file is a continuing flow of information. With a stream file, you see only the flow of information. You do not know whether the information is being stored at the other end of the communications channel or not. You do not have access to a file pointer and you cannot change your position in the stream of information.

You open a stream file to a device just as you would open any other file. Use the device name instead of a file name. The statement

```
OPEN #3: ".AIN", STREAM, TEXT
```

opens a channel to receive binary data from serial port A, the modem port. The statement

```
OPEN #4: ".PRINTER", STREAM, TEXT, OUTIN
```

opens a channel to send text data to the printer.

You use INPUT# and PRINT# with text data, and you use READ# and WRITE# with binary data, just as you do with regular files. Because stream files have no file pointer, you cannot use any of the file position or contingency keywords that you use with sequential or relative files.

```
OPEN #3: ".AIN", STREAM, BINY
READ #3: byte©    ! Receive 1 BINY byte
OPEN #4: ".AOUT", STREAM, BINY, OUTIN
WRITE #4: byte©  ! Send 1 BINY byte

OPEN #8: ".BIN", STREAM, TEXT, INPUT
INPUT #8: char$   ! Receive 1 TEXT character
OPEN #9: ".BOUT", STREAM, TEXT, OUTIN
PRINT #9: char$   ! Send 1 TEXT character
CLOSE
```

Controlling a Device

■ DEVCONTROL #

DEVCONTROL# sends control information to a device. The most common use of DEVCONTROL# is to change the communications protocol (transmission speed, data bits, parity, and stop bits) used by the software drivers that run the serial ports. Both the modem port and the printer port are preset to operate at a speed of 9600 baud (bits per second), using 8 data bits, no parity, and 2 stop bits. You can use DEVCONTROL# to change those settings.

To use DEVCONTROL#, first dimension a 2-element integer array. You store the number 8 in the first element of the array and a code that represents the new protocol setting in the second element of the array. The code is constructed by adding together the values from Table 13-6 that correspond to the protocol settings you want.

Table 13-6. Control Values for Serial Ports

Protocol	Setting	Value
Baud rate	300	380
	600	189
	1200	94
	1800	62
	2400	46
	3600	30
	4800	22
	7200	14
	9600*	10
	19200	4
	57600	0
Data bits	5	0
	6	2048
	7	1024
	8*	3072
Parity	None*	0
	Odd	4096
	Even	12288
Stop bits	0	0
	1	16384
	1.5	−32767
	2*	−16384

*Asterisks mark the settings used by BASIC if you do not use DEVCONTROL# to change the settings.

The code for 1200 baud, 8 data bits, no parity, and 1 stop bit (which is a common protocol for modem communications) is 94 for 1200 baud, plus 3072 for 8 data bits, 0 for no parity, and 16384 for 1 stop bit, for a total of 19550.

The DEVCONTROL# statement itself consists of the keyword DEVCONTROL followed by a space, the channel designator sign #, the channel number, a colon, the @ sign, and the name of the first element of the integer array. The following example shows a complete DEVCONTROL# statement:

```
! Set modem port for 1200 baud modem
DIM setting%(1)    ! 2-element array
OPEN #3: ".AIN", STREAM, BINY
setting%(0) = 8     ! Always use 8 to set protocol
! 1200 baud, 8 data bits, no parity, 1 stop bit
setting%(1) = 94 + 3072 + 0 + 16384
DEVCONTROL #3: @setting%(0)
```

Checking Status of a Device

■ DEVSTATUS #

DEVSTATUS# obtains status information from a device. The most common use of DEVSTATUS# is to find out if any data has been received. The device driver for each serial input port (.AIN or .BIN) maintains a buffer in which it stores data that is received from the device that is plugged into the port. Your program needs to use DEVSTATUS# to make certain there is data in this buffer before the program tries to get the data with a READ# or INPUT# statement.

To use DEVSTATUS#, you need to dimension a 3-element integer array and store the number 2 in the first element of the array. The DEVSTATUS# statement itself consists of the keyword DEVSTATUS followed by a space, the channel designator number, the channel number, a colon, the @ sign, and the name of the first element of the integer array. When the DEVSTATUS# statement is executed, BASIC puts the number of bytes of data you can read from the buffer into the third element of the integer array, as shown in the following example.

```
! Use DEVSTATUS #
DIM status%(2)
OPEN #3: ".AIN", STREAM, BINY
status%(0) = 2 ! Always use 2 to get # bytes
DEVSTATUS #3: @status%(0)
bytes.to.read = status%(2)  ! get # from 3rd element
```

Using .PRINTER

The .PRINTER driver is more convenient to use than the .BOUT driver if you have a printer connected to the printer port of your Macintosh. The .PRINTER driver handles a number of printer-related details for you, while the .BOUT driver does not. Specifically, the .PRINTER driver sends the proper instructions to initialize the attached printer, sends a line feed character to move the paper whenever printing reaches the end of a line, starts a new line after 75 characters, automatically skips six lines when you reach the bottom of a page, and sets tab stops every four characters.

In addition to taking care of these housekeeping matters for you, the .PRINTER driver also changes some characters in your output into characters that make sense to the printer. The characters that are changed by the .PRINTER driver are listed in Table 13-7.

Table 13-7. Characters Changed by the .PRINTER Device

Character You Send	What .PRINTER Prints
π	pi
Π	PI
≤	<=
≥	>=
≠	<>
©	(*)
∞	INFINITY
CHR$(253)	turns on boldface)*
CHR$(254)	turns off boldface)*

*.PRINTER does boldfacing only with the Apple Imagewriter printer.

When your printer is operating in ASCII mode, it does not reorganize some of the special Macintosh characters. To compensate for this, the .PRINTER driver changes the following characters into strings the printer can print: the Greek letter π (lowercase and uppercase), the three one-character relational operators, the character variable designator, and the infinity sign. In addition, the .PRINTER driver changes CHR$(253) into the control code that tells the printer to start printing in boldface and changes CHR$(254) into the control code that tells the printer to stop using boldface. The two boldface control characters and the tab settings every four character positions work only if you are using the Apple Imagewriter printer.

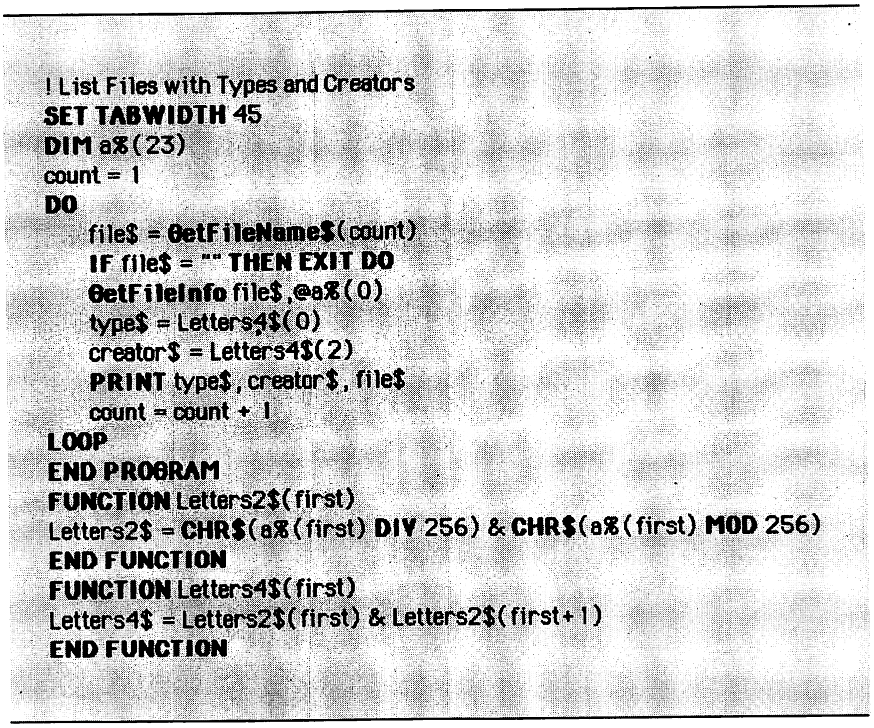
EXAMPLE PROGRAMS

The example program shown in Figure 13-6 lists all of the files on your disk along with their file types and creator codes. After setting tab stops in the output window and dimensioning a 24-element integer array, the program initializes the variable count at 1 and enters a DO loop that is executed once for each file that is on the disk.

The first statement in the DO loop gets the name of a file by using the variable *count* as an argument to GETFILENAME$. If the name returned is the empty string, the program exits from the loop and ends. If GETFILENAME$ returned a file name, the program calls GETFILEINFO with that file name and the address of the first element of the 24-element integer array as arguments.

The program uses the defined function Letters4$ to decode information from the data array. First the file's type and then its creator are decoded, stored in strings, and printed alongside the file name. Then the program increments the variable *count* and repeats the loop until all the files are listed.

The example program in Figure 13-7 prints the data that comes in over a 1200 baud modem. The program starts by dimensioning two integer arrays for use later with the DEVCONTROL# and DEVSTATUS# commands. Then the program opens channel 1 for input of binary data from the modem port, and opens channel 2 for output of text data to the printer.



```
! List Files with Types and Creators
SET TABWIDTH 45
DIM a%(23)
count = 1
DO
    file$ = GetFileName$(count)
    IF file$ = "" THEN EXIT DO
    GetFileInfo file$,@a%(0)
    type$ = Letters4$(0)
    creator$ = Letters4$(2)
    PRINT type$, creator$, file$
    count = count + 1
LOOP
END PROGRAM
FUNCTION Letters2$(first)
Letters2$ = CHR$(a%(first) DIV 256) & CHR$(a%(first) MOD 256)
END FUNCTION
FUNCTION Letters4$(first)
Letters4$ = Letters2$(first) & Letters2$(first+1)
END FUNCTION
```

Figure 13-6. List file types and creators

The program uses DEVCONTROL# to set the modem port for a 1200 baud modem, as shown in the example earlier in this chapter. The settings are 1200 baud, 8 data bits, no parity, and 1 stop bit. Two PRINT# statements send a title line (including the date) and a blank line to the printer.

The program then enters a DO loop in which the program repeatedly uses DEVSTATUS#. If *ready%(2)* contains a positive number, indicating that data has been received, the program executes a FOR/NEXT loop that reads each byte from the modem into a character variable and then converts the byte to a one-character string and prints it on the printer. Note that the READ# command is used to receive the binary information and the PRINT# command is used to send the text.

After each character is read from the modem, the program checks to see if the character is CHR$(4), the character commonly used to

```
! Print information from the modem
DIM ctl%(1),ready%(2)
OPEN #1: ".AIN", STREAM, BINY
OPEN #2: ".PRINTER", STREAM, TEXT, OUTIN
ctl%(0) = 8
ctl%(1) = 19550 ! 1200 baud, 8 data, no parity, 1 stop
DEVCONTROL #1: @ctl%(0) ! Set modem protocol
PRINT #2: "Text received by modem on "; DATE$
PRINT #2:
DO
    ready%(0) = 2
    DEVSTATUS #1: @ready%(0)
    IF ready%(2) > 0 THEN
        FOR count = 1 TO ready%(2)
            READ #1: byte©
            IF byte© = 4 THEN EXIT DO
            PRINT #2: CHR$(byte©);
        NEXT count
    ENDIF
LOOP
PRINT "End of transmission."
CLOSE
END PROGRAM
```

Figure 13-7. Print modem data

signal the end of a transmission. If the character is CHR$(4), the program displays "End of transmission." in the output window and ends. This program does not save a machine-readable copy of the information that comes in through the modem. You may want to modify it to write the information to a disk file.

PRACTICE EXERCISES

1. The program in Figure 13-1 uses GETFILENAME$ to list all the files on a volume. How would you modify that program so that it will not print out the name of any files marked as invisible?
2. How would you write a BASIC program that moves the file "Macintosh BASIC" into the System folder? You may assume that the System folder is the folder that contains the file named "System."
3. What BASIC commands would you use to open a channel to receive data from the modem port at 300 baud with 8 data bits, no parity, and 1 stop bit?
4. How would you change the program in Figure 13-1 to make the program print the list of files on a printer instead of in the output window?

Chapter 14

Using the Interactive Debugger

Command:

- STOP

Macintosh BASIC has an interactive debugger to help you locate any trouble spots in your programs. The debugger is also an excellent learning tool. You can follow your program line by line and see what each line does. You can see which lines change the values of your variables and what each line does to the output window. If something odd is happening in your program but you can't figure out what it is, the debugger can be an indispensable tool.

You will get the most from this chapter if you run the debugger while reading the text. The examples provided use the sorting program, Figure 9-4, from the end of Chapter 9.

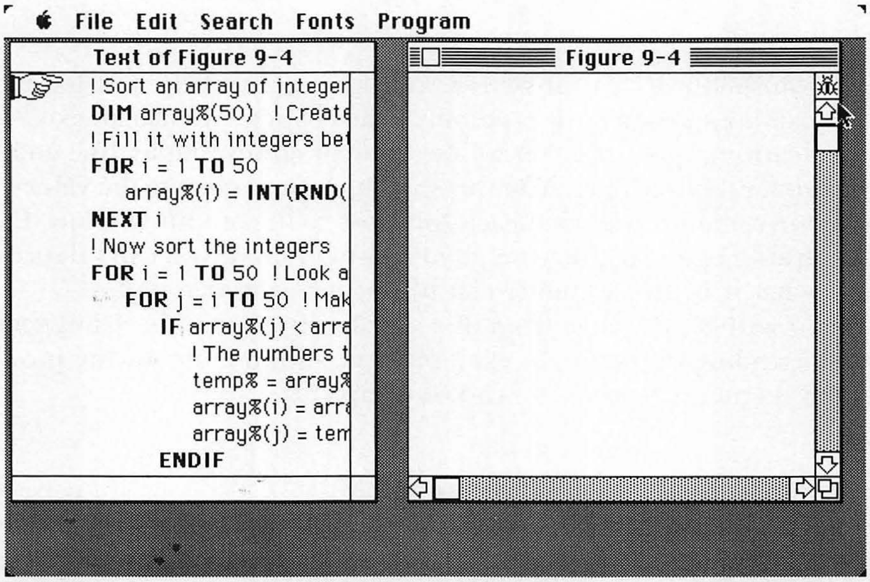
TURNING ON THE DEBUGGER

When you use the debugger, you need to let Macintosh BASIC know what program you are going to debug. You do that by making the program's text window active before starting the debugger.

If your program is not already running, make its text window active before you turn on the debugger. If your program is running, you can start the debugger with either the text or the output window active. You turn on the debugger by selecting the Debug option from the Program menu or by pressing the keyboard combination COMMAND-d.

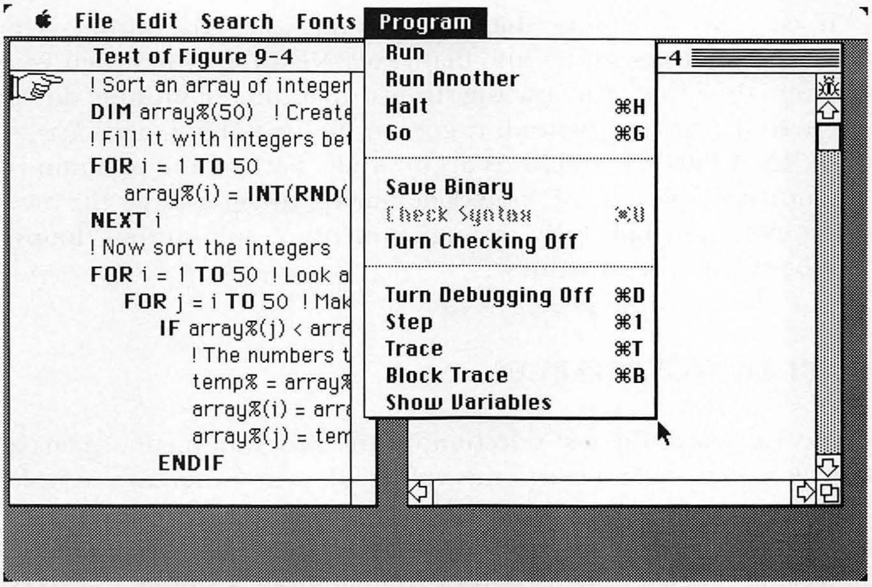
To follow the examples in this chapter, start up Macintosh BASIC and open a text window with the sorting program in Figure 9-4. Then select Debug from the Program menu to start the debugger.

Figure 14-1 shows what your screen looks like just after you turn on the debugger. When you turn it on, the debugger shifts the text of your program to the right to make room for a hand with a pointing finger. The finger points to the program line that is ready to be executed. If your program was not already running, the debugger also opens an output window and prepares to execute the program. A little bug appears in the output window's program status area to indicate that the program is running under the control of the debugger.



```
! Sort an array of integer
DIM array%(50)  ! Create
! Fill it with integers be
FOR i = 1 TO 50
    array%(i) = INT(RND(
NEXT i
! Now sort the integers
FOR i = 1 TO 50  ! Look a
    FOR j = i TO 50  ! Mak
        IF array%(j) < arra
            ! The numbers t
            temp% = array%
            array%(i) = arr
            array%(j) = tem
        ENDIF
```

Figure 14-1. Starting the debugger



```
! Sort an array of integer
DIM array%(50)  ! Create
! Fill it with integers bet
FOR i = 1 TO 50
    array%(i) = INT(RND(
NEXT i
! Now sort the integers
FOR i = 1 TO 50  ! Look a
    FOR j = i TO 50  ! Mak
        IF array%(j) < arra
            ! The numbers t
            temp% = array%
            array%(i) = arra
            array%(j) = tem
        ENDIF
```

Figure 14-2. The debugger's menu items

Turning on the debugger enables several debugging options at the bottom of the Program menu. Figure 14-2 shows the Program menu just after the debugger has been turned on. The debugger's Step, Trace, Block Trace, and Show Variables command options are all enabled. In addition, the Debug option changes to Turn Debugging Off. Both options use the same keyboard shortcut, COMMAND-d.

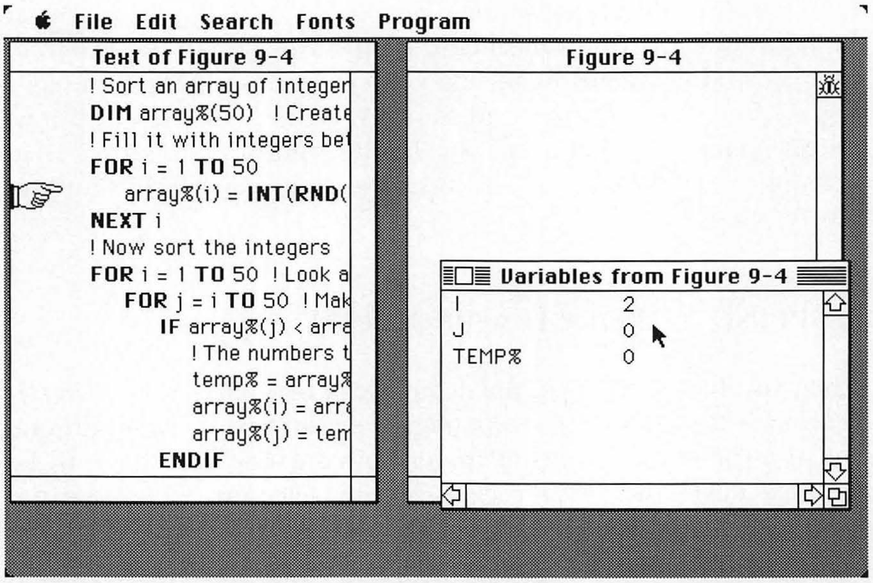
STEPPING THROUGH A PROGRAM

When you first turn it on, the debugger is in what is called *single-step mode.* In this mode the debugger executes your program one line at a time. The pointing finger points to the line that will be executed next. You execute each line by selecting Step from the Program menu or by typing either the COMMAND-1 or the COMMAND-SPACE combination from the keyboard. Once you become familiar with the debugger, you will probably find that using one of the keyboard combinations is easier.

If you are following the example on your Macintosh, step through five lines to the line that reads NEXT i. Now when you execute this line, the tracing finger does not continue down through the listing. Instead, it goes to the line above, array%(i) = INT(RND(1000)) + 1. This is appropriate, because the program is executing a FOR/NEXT loop. The tracing finger follows the program execution faithfully through functions, subroutines, loops, and other control structures.

DISPLAYING VARIABLES

Show Variables, the last selection on the Program menu, opens a new window that displays the values of your program's simple variables and functions. It does not display the values of array elements. Figure 14-3 shows the variables window for the example program. The window lists all of your program's simple variables and functions, even if execution has not reached the statements that use them. The variables *J* and *TEMP%,* which have not been



```
! Sort an array of integer
DIM array%(50)  ! Create
! Fill it with integers be
FOR i = 1 TO 50
   array%(i) = INT(RND(
NEXT i
! Now sort the integers
FOR i = 1 TO 50  ! Look a
   FOR j = i TO 50  ! Mak
      IF array%(j) < arra
         ! The numbers t
         temp% = array%
         array%(i) = arra
         array%(j) = tem
      ENDIF
```

```
I          2
J          0
TEMP%      0
```

Figure 14-3. Debugging with Show Variables active

used yet, are displayed with their initial values of zero. The variable *I,* which is being used, is 2.

Names are listed in the variables window in all capital letters because BASIC does not pay any attention to the case of the letters when it is working with variable and function names. If your program uses the variable names *temp%* and *TEMP%,* BASIC assumes they are the same variable. Listing the names in all capital letters in the variables window helps remind you of that fact.

The debugger updates the variables window display every time a program statement changes the value of a variable or function. If you continue to single-step through the example program, you will see that the value of *I* increases by one each time you execute the NEXT i statement.

TRACING EXECUTION

You do not have to single-step through all fifty iterations of the FOR/NEXT loop to get to the rest of the example program. The Trace command on the Program menu will step through the program for you. The tracing finger still points to each program line as it is executed, and the values of variables and functions are still updated in the variables window. The keyboard combination for Trace is COMMAND-t.

The debugger traces through a program quickly. If you watch carefully, you can still see which statements are being executed, but things are moving too fast to see the effects of any single command. When you want to stop the automatic tracing, give one of the single-step commands, COMMAND-1 or COMMAND-SPACE. Then you can continue to single-step through a section of the program, or you can use COMMAND-t to resume the automatic tracing.

If you want to trace, but don't want things to move quite as fast as they do with the Trace command, you can try holding down one of the keyboard combinations for single-stepping. On the Macintosh keyboard, every key is a repeating key when it is held down. You can use this feature of the keyboard to make repeated single-step commands look like a slightly slower version of the Trace command. Any time you want to stop, all you have to do is take your finger off the key. You can adjust the repetition rate of the keyboard with the Control Panel desk accessory.

USING BLOCK TRACE

Block Trace is the speed champion of the debugging commands. It executes a single control structure at full speed. You can select Block Trace from the Program menu or by pressing COMMAND-b. Block Trace is most useful when you are single-stepping or tracing through a program and reach a loop or subroutine call that you do not want to trace. One execution of the Block Trace command will execute the entire control structure and then stop at the end of the structure so you can resume tracing or single-stepping.

The control structures that can be entirely executed by a single Block Trace command are IF/ENDIF, DO/LOOP, FOR/NEXT, SELECT CASE/END SELECT, GOSUB, CALL, PERFORM, and WHEN/END WHEN. The last three kinds of control structures are described in the next chapter. While the debugger is executing a block trace, it does not move the tracing finger or update the values in the variables window. It updates both when the block trace is finished.

To experiment with Block Trace, close the output window and restart the program in Figure 9-4 from the beginning by turning on the debugger. Single-step to the line that reads FOR i = 1 TO 50. Then press COMMAND-b. When the block trace is finished, the pointing finger reappears at the comment line "Now sort the integers." The first FOR/NEXT loop has been executed, and you are ready to sort the array. Single-step once to get to the next line that reads FOR i = 1 TO 50. Press COMMAND-b to block trace again. When the finger reappears after a few seconds, the sort has been done.

Now you are ready to execute the FOR/NEXT loop that prints the sorted numbers. For variety, use COMMAND-t to start a normal trace, and then press COMMAND-b after a few numbers have been printed. You have just demonstrated that you can start a block trace from a normal trace as well as from single-stepping.

TURNING OFF THE DEBUGGER

To turn off the debugger, you can select Turn Debugging Off from the Program menu or press COMMAND-d on the keyboard. If your program was already running when the debugger was switched on, your program resumes normal execution. If you do not want your program to continue executing when you turn off the debugger, you can click on the output window's close box—that will turn off the debugger, stop execution, and return you to the text window of your program.

SETTING A BREAK POINT

- STOP

If you have a problem area in your program and you want to use the debugger, you could start the program normally and then turn the debugger on just before the problem area. This will work, but you might have to do it several times before you can start the debugger at just the right place.

A more precise way to turn on the debugger part way through your program is to insert a STOP statement at the appropriate place in your program. The STOP statement stops program execution and turns on the debugger. Its effect is the same as pressing COMMAND-d while BASIC is executing that line of your program. You can use STOP alone, or you can use STOP as one course of action in an IF or CASE statement.

```
STOP
IF a > limit THEN STOP
IF wrong~ THEN STOP
```

You can insert STOP commands into your program in as many places as you wish. When you have finished debugging, however, it is a good idea to use the Find command in the Search menu to locate all occurrences of STOP and then remove them from your finished program.

DEBUGGING EXAMPLES

This section gives a few suggestions about ways you can use Macintosh BASIC's programming tools to help you write and correct programs. It is not exhaustive. If you use the debugger and the

Find command regularly, you will undoubtedly find other ways and situations in which to use these versatile tools.

☞ Your program has been working well for six weeks, but when you use a new set of data, the program seems to "freeze up." The symbol in the output window indicates that the program is executing, but nothing is printed. This situation is fairly common, although it can sometimes be hard to diagnose. The program may be executing some type of an infinite loop.

The best way to check for the possibility of an infinite loop is to let your program run until it appears to be in the infinite loop. Then turn on the debugger and begin to trace or single-step to see what the program is doing. Most infinite loops are fairly short if the overall program is well organized. Once you have discovered the location of the loop, you can single-step with the variables window open to see why the program is stuck in the loop. You may notice a variable that takes on an unexpected value. Pay particular attention to variables that cause your program to end or exit from the loop.

☞ The value of a multiple-line function does not seem to be getting set properly. The simplest way to solve this problem is to insert a STOP statement immediately after the first line of the function's definition and then run the program. When the function is called, BASIC stops the program and turns on the debugger. You can then open the variables window and single-step through the function.

If single-stepping shows that you are executing the line that assigns a value to the function, check to see that the name of the function in that line is spelled correctly. If it is not, you may be assigning the value to some variable you will never use again.

If the assignment seems to be correct but has the wrong value, check to be sure your function is getting the correct data and that the parameter names are spelled correctly each time they are used in the function definition. When you have finished stepping through the function definition, you can let your program resume normal execution by typing COMMAND-d.

☞ Your GOSUB statement generates an "Undefined label" error message, and you are sure you used that name for a subroutine. For this situation, the global search capability of the Find command may be more helpful than the debugger. Use Find to look for the colon character (:), and check the Include Embedded Words option. The search may find a few colons in lines that do not contain labels, but it will absolutely find every label that is correctly defined in your program. You will certainly notice if the label you defined was slightly different from what you remembered.

☞ One of your variables is not set or is being set to an incorrect value. Find the line where your variable is supposed to be set, and examine it carefully. If you can't find any errors in that line, the cause may be an error in a calculation earlier in your program. Turn on the debugger and select Show Variables.

Look through the variables list carefully. If you see any variable names you don't remember using in your program, use Find on the Search menu to look at the lines containing that name. If you do this carefully, you should find all mistyped variable names that could cause wrong results. If checking the variable names does not identify the cause of the problem, you may need to trace or single-step through your program with the variables window open to discover when one of your variables is first set to an incorrect value.

☞ The text output from your program is not being printed. If you are using GPRINT, text may be printed outside the visible area. To check this, put a STOP statement just before or just after the GPRINT statement; then run the program. If the program stops and the debugger comes on, your GPRINT statement is being executed, so you need to use an AT command in the GPRINT statement to position the output in its correct location. If the program does not stop, your GPRINT statement is not being executed. You will need to check the flow of program execution by single-stepping or tracing to find out why the program is skipping around the GPRINT statement.

Chapter 15
Advanced Control Structures

Commands:

- CALL, SUB, END SUB, EXIT SUB
- PERFORM, PROGRAM, END PROGRAM, EXIT PROGRAM
- WHEN ERR, WHEN KBD, END WHEN, IGNORE WHEN

This chapter explains several advanced topics involving the flow of control in your programs. First you will be introduced to a new kind of subroutine, one that allows you to pass parameters. Then you will see how one program can summon another. That is followed by a description of interrupts, which provide a way to handle events like errors and keypresses without interfering with the execution of your main program. The chapter concludes with a discussion of how to run several programs at once.

SUBROUTINES WITH PARAMETERS

■ CALL, SUB, END SUB

A subroutine whose beginning is marked with a SUB statement and whose ending is marked with an END SUB statement is invoked with the CALL statement rather than the GOSUB statement (explained in Chapter 6). You can pass arguments to this kind of subroutine, and you can receive values in return. Here is an example:

```
SUB AreaRect(width,height)
PRINT "Now we're in the subroutine."
PRINT "The area is "; width * height
END SUB
```

This subroutine begins with a SUB command followed by the name of the subroutine, AreaRect. The name is followed by a list of parameters in parentheses with the parameters separated by commas. You can define as many parameters as you wish in the SUB statement. You can use any of the ten data types for parameters, and you can use arrays as well as single variables. When you use an array, you must include parentheses after the name with a comma for each dimension after the first.

You use a CALL statement to start execution of a subroutine defined with SUB. If the SUB statement specified parameters for the subroutine, the CALL statement must include the correct number of arguments in the proper order and with the proper data types to match the parameters. If an array is passed to a subroutine, it must have the same number of dimensions in the subroutine and the calling program. Here is a short program that uses the CALL statement:

```
! Main program
CALL AreaRect(9,3*8)
! Control comes here after END SUB
PRINT "Back in the main program."
END PROGRAM

SUB AreaRect(width,height)
PRINT "Now we're in the subroutine."
PRINT "The area is "; width * height
END SUB
```

When you only want to pass a value from the calling program to the subroutine, you can use any BASIC expression of the proper data type in the CALL statement. When you want to return a value to the calling program, the corresponding argument in the CALL statement must be the name of a variable of the proper type. In this case, the variable receives the value of the corresponding subroutine parameter when execution of the subroutine is finished.

If you do not want a value passed back from the subroutine to one of your variables in the calling program, you can use an expression instead of a variable name in the CALL statement. For instance, instead of using a variable named *testY* in the CALL statement, you could use the expression *testY*+ 0 to prevent a value from being passed back to the calling program.

END SUB must be the last statement in the subroutine. When BASIC reaches this statement, it copies any values being returned to the calling program into the proper variables. Then control is transferred to the statement in the calling program following the CALL statement that invoked the subroutine.

Exiting Subroutines

■ EXIT SUB

You can exit a subroutine early with an EXIT SUB command, which transfers control to the END SUB statement, copying any return parameters to the calling program. BASIC does not require the word SUB in the EXIT SUB statement, but you should use both words to avoid confusion among EXIT SUB, EXIT FOR, and EXIT DO.

```
SUB AreaRect(width,height)
PRINT "Now we're in the subroutine."
IF width * height <= 0 THEN EXIT SUB  ! Leave early
PRINT "The area is "; width * height
END SUB
```

CALLING ANOTHER PROGRAM

■ PERFORM, PROGRAM, END PROGRAM

One program can be summoned and executed from another just like a special kind of subroutine. Unlike subroutines, programs reside in files separate from the program that calls them. The program being summoned can be in a disk file, or it can be in memory if you have opened it earlier.

The program to be summoned is defined with PROGRAM and END PROGRAM statements. The PROGRAM statement must be the first statement in the program. If values will be passed to the program or back to the program that called it, they must be specified in a parameter list in the PROGRAM statement.

The program being summoned does not have access to variables from the calling program unless they are passed as parameters. The following program, AreaTriangle, receives two values (*width* and *height*) from the calling program and uses the values to calculate the area of a triangle.

```
PROGRAM AreaTriangle(width, height)
PRINT "Now we're in the summoned program."
PRINT "Area = "; width * height / 2
END PROGRAM
```

Programs that are written to be called with a PERFORM statement reside in the same kind of disk files as other programs. Because there is no way to tell without opening the file whether the program starts with a PROGRAM statement and requires parameters, it is a good idea to adopt your own naming convention for programs that must be called with parameters.

The END PROGRAM statement is normally the last statement of a program. It returns control to the program containing the PERFORM statement. If the END PROGRAM statement is missing, BASIC returns to the calling program when it runs out of statements to execute.

You use PERFORM to execute one program from another. The keyword PERFORM is followed by the name of the program to be executed. If the program to be executed requires parameters, you put the arguments, separated by commas, in parentheses after the program name. Do not type any spaces between the program name and the left parenthesis that begins the parameter list.

Arguments that will receive values from the summoned program must be variable names preceded by the @ character; other arguments can be expressions that match the type of parameter in the PROGRAM statement. Otherwise, all of the rules for passing parameters in subroutines apply. No value will be returned to the calling program unless you use the @ character in the PERFORM statement. The following example shows a PERFORM statement that calls a program named AreaRect, which calculates the area of a rectangle. The @ character in the PERFORM statement causes a value to be returned in the variable *area.* The @ character in the PROGRAM statement has no effect; it is included only as a reminder that the value is being returned to the calling program.

```
! First program:
PERFORM AreaRect(9, 3*8, @area)
! Control comes here when AreaRect is done.
PRINT "The area is "; area
END PROGRAM

! Second program, separate file:
PROGRAM AreaRect(width, height, @result)
result = width * height
END PROGRAM
```

You can use the same name for a function, a subroutine, and a program if you can keep them straight. When one program executes another, BASIC preserves the variables of the original. After control returns from the summoned program, all the calling program's variables will have their previous values except those whose values were passed from the summoned program.

When BASIC executes a PERFORM statement in your program, it looks for a file that matches the program name you specify. If the program is not already in memory, BASIC looks on the default disk drive. You can specify a particular drive by using a volume name as the first part of the file name in the PERFORM statement.

Because programs are stored in disk files, there will usually be a delay while your program reads a file during a PERFORM statement. To avoid constant delays during program execution, you should either use PERFORM only for segments of your program that you use infrequently or open the program to be performed prior to executing the calling program. Code segments that you use frequently should be included as subroutines in your main program file.

Exiting Programs

- EXIT PROGRAM

You can terminate the execution of a program at any point with the statement EXIT PROGRAM. If the program was summoned with a PERFORM statement from another program, parameters are passed and control is returned to the other program as if an END PROGRAM statement had been encountered. If you started executing the program from BASIC's programming environment, you are returned to the programming environment. Both words of the EXIT PROGRAM command are required.

```
! Second program, separate file
PROGRAM AreaRect(width,height)
PRINT "Now we're in the second program."
IF width * height <= 0 THEN EXIT PROGRAM  ! Leave early
PRINT "The area is "; width * height
END PROGRAM
```

USING INTERRUPTS

An *interrupt* is an event that causes an interruption in your program's normal activities. If you enable, or allow, interrupts for a particular type of event, whenever that event occurs your program is suspended while another small program (called an *interrupt-handling routine*) is executed. When the interrupt-handling routine is finished, your program resumes execution. If you do not enable any interrupts, your program proceeds normally without interruptions. Macintosh BASIC allows you to enable interrupts and write your own interrupt-handling routines for two types of events: errors and keypresses.

The real power of interrupts comes from the fact that you can enable and disable interrupts and change your interrupt-handling routines as often as you wish while your program is executing. When your program is reading files, for instance, you might want to use an interrupt-handling routine that takes special actions if there is a file-related error. At another point in your program, you might want to create an interrupt-handling routine that handles errors in input typed from the keyboard.

Trapping Errors

- WHEN ERR, END WHEN

The WHEN ERR statement allows your program to handle errors. WHEN ERR serves two purposes: it enables interrupts for errors, and it also marks the beginning of your interrupt-handling routine. The END WHEN statement is required to mark the end of an interrupt-handling routine.

When BASIC reaches a WHEN ERR statement, it does not immediately execute the instructions between WHEN ERR and END WHEN. Instead, BASIC records the location of the WHEN ERR, enables the error interrupt, and continues executing your program beginning with the statement immediately following the END WHEN.

When an error occurs, for whatever reason, BASIC immediately suspends execution of your program and begins executing the interrupt-handling routine beginning with the WHEN ERR statement. When the END WHEN statement is reached, BASIC resumes execution of your program at the place where it was suspended.

The interrupt-handling routine you supply between the WHEN ERR and END WHEN statements replaces the actions BASIC normally takes when an error occurs. That makes you responsible for handling the error or presenting a message describing the error. Usually you will want to use the ERR system function (described in Chapter 7) to find out what type of error occurred. A complete list of error messages and codes is listed in Appendix B. Generally, error codes from 66 to 97 are system errors, codes from 98 to 153 are errors related to files, and codes from 154 up are related to your program and its execution.

If BASIC encounters a second WHEN ERR statement in your program, it replaces the first error interrupt-handling routine with the second one. The interrupt remains enabled until it is turned off with an IGNORE WHEN ERR statement. WHEN ERR traps syntax errors in your program as well as errors from other causes, so until you know that you have found and removed any syntax errors in your program, you should use it with caution. A syntax error between a WHEN ERR and an END WHEN statement will cause the error-handling routine to keep interrupting itself until BASIC runs out of memory.

When an INPUT statement in your program requires a number and you type text instead, BASIC displays a dialog box containing the message "Expected a number." The following example replaces this dialog box with a message in the output window, but it does not report the nature of any other error.

```
WHEN ERR
IF ERR = 182 THEN ! Expected a number
    PRINT "Please retype using numbers only:"
    ENDIF
END WHEN
```

Trapping Keyboard Input

- WHEN KBD, END WHEN

WHEN KBD works similarly to WHEN ERR, except that the statements between WHEN KBD and END WHEN are executed whenever a character key (any key other than SHIFT, CAPS LOCK, OPTION, and COMMAND) is pressed on your keyboard. If you hold down a key long enough to repeat, the WHEN KBD statements are executed once for each repetition. Your interrupt-handling routine can use the KBD function described in Chapter 7 to get the ASCII value of the key that was pressed.

Some uses of WHEN KBD are to check for a particular key stroke, to ignore certain keys altogether, to allow only numbers to be input, or to make a sound whenever a key is pressed. The following loop waits for a key to be pressed. The program stays in the DO loop until a keypress causes the WHEN KBD statements to be executed, setting the variable *key* to 1.

```
! Wait for keypress
key = 0
WHEN KBD
    key = 1
END WHEN
DO
IF key = 1 THEN EXIT DO
LOOP
PRINT "A key was pressed."
```

Turning Interrupts Off

- IGNORE WHEN

IGNORE WHEN clears an interrupt setting so BASIC will not attempt to execute an interrupt-handling routine. The statement IGNORE WHEN ERR clears the WHEN ERR setting. IGNORE WHEN KBD clears the WHEN KBD setting.

```
IGNORE WHEN KBD  ! Turns off KBD interrupts
IGNORE WHEN ERR  ! Turns off ERR interrupts
```

RUNNING SEVERAL PROGRAMS

Macintosh BASIC allows you to run several programs at once by dividing the time between programs. The first program executes for a fraction of a second, the second program executes for a fraction of a second, and so on. As a result, each program runs slower than when it is running by itself.

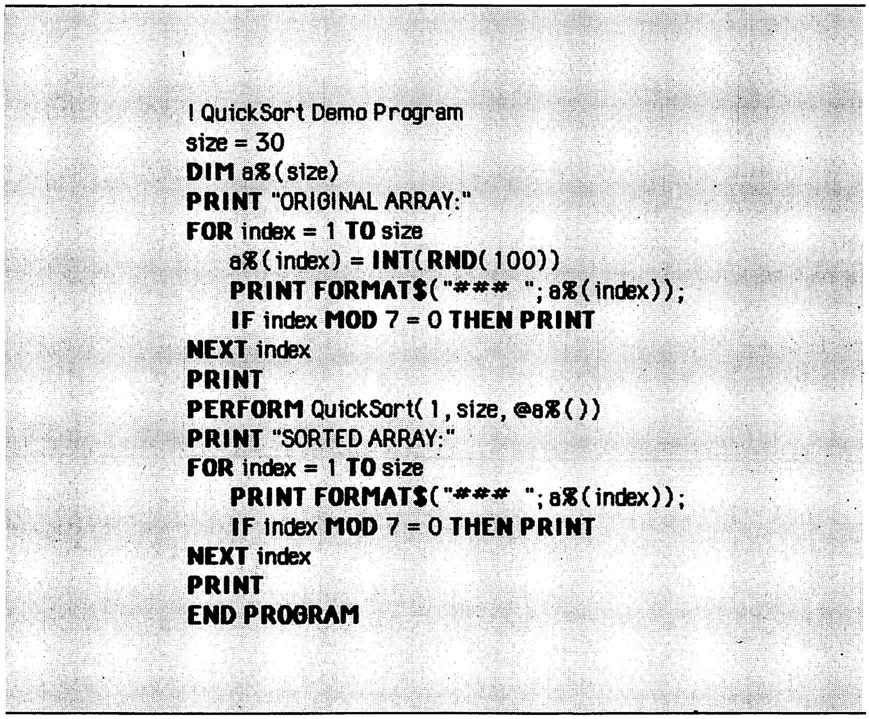
The interrupt-handling routines in your program are only active when your program's output window is active. Things like keypresses and mouse button clicks only affect the program with the active output window, so several programs will not all react to the same action. If your program must always keep track of interrupt events, it will not work properly if more than one program is running.

Another thing to keep in mind when writing programs that may share time with other programs is that programs may compete for access to the same data file, serial port, or display screen. For example, if two programs both enlarge their output windows to cover the entire screen, only one of the windows will be visible. Or if two programs use the same data file, one of them will get there first, and the second, when it tries to access the same file, will probably stop with an error message because the file is already in use. If you plan for this when you write your programs, you can use WHEN ERR to trap the error and have each program wait until the file or device is available.

EXAMPLE PROGRAM

The Quicksort Demo program in Figure 15-1 demonstrates recursive use of the PERFORM command. The Quicksort Demo program creates and displays an array of 30 integers with random values from 0 to 100. A PERFORM statement summons the Quicksort program shown in Figure 15-2 to sort the array. The @ sign in front of the array name in the parameter list indicates that the values in the array will be returned to the Demo program from the Quicksort program.

The interesting part of the example is the Quicksort program (Figure 15-2). Note the match between the parameters in the PROGRAM statement and the arguments in the PERFORM statement that calls the program.



```
! QuickSort Demo Program
size = 30
DIM a%(size)
PRINT "ORIGINAL ARRAY:"
FOR index = 1 TO size
    a%(index) = INT(RND(100))
    PRINT FORMAT$("###  "; a%(index));
    IF index MOD 7 = 0 THEN PRINT
NEXT index
PRINT
PERFORM QuickSort( 1, size, @a%())
PRINT "SORTED ARRAY:"
FOR index = 1 TO size
    PRINT FORMAT$("###  "; a%(index));
    IF index MOD 7 = 0 THEN PRINT
NEXT index
PRINT
END PROGRAM
```

Figure 15-1. Quicksort Demo program

```
PROGRAM Quicksort(start, end, @b%())
pivotposition = (start+end) DIV 2
pivotnumber = b%(pivotposition)
L = start
R = end
DO
    IF L >= R THEN EXIT DO
    DO
        IF b%(L) >= pivotnumber THEN EXIT DO
        L = L + 1
    LOOP
    DO
        IF b%(R) <= pivotnumber THEN EXIT DO
        R = R - 1
    LOOP
    IF L <= R THEN
        temp% = b%(L)  ! Swap L and R elements
        b%(L) = b%(R)
        b%(R) = temp%
        L = L + 1
        R = R - 1
    ENDIF
LOOP
IF start < R THEN PERFORM Quicksort(start, R, @b%())
IF L < end THEN PERFORM Quicksort(L, end, @b%())
END PROGRAM
```

Figure 15-2. Quicksort

The quicksort procedure is slightly more complicated than the bubble sort procedure you saw earlier. Quicksort works by repeatedly subdividing segments of the array until each segment is either sorted or has less than two elements in it. Each time a segment is divided, all of the numbers in one segment are higher than any of the numbers in the other segment.

The Quicksort program starts by picking an arbitrary point, called the pivot position, in the middle of the segment. The variable *L* starts at the low end of the segment and skips past all the other elements that are lower than the pivot number. Starting the variable *R* from the high end of the segment, the program skips past all the numbers that are higher than the pivot number. If the variables *L* and *R* meet at the pivot number, the segment can be subdivided at the pivot number. If not, the program swaps the values pointed to by *L* and *R* and repeats the checking operation.

The Quicksort program further divides the segments on each side of the pivot number by executing itself twice, once for the new smaller segment below the pivot number and once for the new smaller segment above the pivot number. These calls are recursive, which makes this Quicksort program a good example. The quicksort algorithm is usually faster than the bubble sort algorithm, but the recursive PERFORM statements use a lot of memory.

PRACTICE EXERCISES

1. Try your hand at writing a subroutine that takes the average of two numbers and passes the result back to the main program through a parameter. The program will have three parameters: the two numbers to be averaged and the result. Write the CALL statement as well.
2. Rewrite the subroutine and CALL statement from the first exercise as a separate program and a PERFORM statement.
3. Can you write a loop that prints "Is that a Lisa diskette?" whenever error number 102 ("Not a MAC Diskette") is encountered?
4. Try writing a loop that prints "See page 40 in the manual" whenever a question mark is typed from the keyboard.

Part three

Special Macintosh Techniques

Chapter 16

Graphics and Shapes

Commands:

- PLOT, FRAME, PAINT, ERASE, INVERT
- SET/ASK PENPOS, SET/ASK PENSIZE
- SET/ASK PENMODE, SET/ASK PATTERN
- SET/ASK OUTPUT, SET/ASK DOCUMENT
- SET/ASK SCALE, SET/ASK LOCATION
- SET/ASK PICSIZE

Shapes:

- RECT, OVAL, ROUNDRECT/WITH

The Macintosh is a superb graphics machine. This chapter describes the graphics commands that are in the Macintosh BASIC language. In addition to these commands, Macintosh BASIC allows you to use the QuickDraw routines found in the Macintosh toolbox to perform more specialized functions. Some of the QuickDraw routines are described in Chapter 22.

In order to get the full benefit of this chapter, you will need to understand how the Macintosh screen is divided into pixels, how to specify a location in the coordinate system of your program's output window, and how to use the SET/ASK PENPOS command to position the pen on the graphics screen. These subjects were covered in the discussion of the GPRINT command in Chapter 10. If you are not familiar with them, you should read the relevant portions of Chapter 10 before reading the rest of this chapter.

DRAWING POINTS AND LINES

■ PLOT

The PLOT command allows you to draw both points and lines. If you specify a single point, PLOT draws that point. If you specify two or more points, PLOT draws the first point and then draws a line to the next point. A line is drawn from one point to the next until there are no remaining points. You could draw an entire picture using one PLOT statement by supplying the coordinates of each dot in the picture.

Each point is specified by two numbers: the horizontal and vertical coordinates at which the point is located in the output window. Use a comma to separate the two numbers. The value of a coordinate may be any number between −32767 and +32767. If the value of a coordinate is not already an integer, BASIC rounds it to the nearest integer.

Negative coordinate positions can be plotted, but you cannot see them because they will be beyond the left edge or above the top of the window. Positive coordinates greater than 240 will not be visible unless you enlarge the normal output window or scroll to examine a different part of the document behind the window. You can use the SET OUTPUT command described later in this chapter to enlarge the window.

Semicolons separate the points in a PLOT statement. You can use an extra semicolon at the end of a list of points, where its effect is similar to that of a semicolon at the end of a PRINT statement. The ending semicolon allows the next PLOT statement to resume drawing where the last one stopped. In other words, the next PLOT statement will start drawing from the current position of the graphics pen to the first point in the list. You can cancel the effect of a trailing semicolon by using a statement consisting of only the word PLOT.

```
PLOT h,v            ! Plots one point at h,v
PLOT 1,3; 8,90      ! Line from 1,3 to 8,90
PLOT 9,30;          ! Plot point to start line
PLOT 30,30;         ! Line from previous semicolon
PLOT                ! Cancels previous semicolon
```

SHAPES

■ RECT, OVAL, ROUNDRECT/WITH

In addition to points and lines, Macintosh BASIC provides you with three basic shapes: the rectangle, the oval, and the rectangle with rounded corners. The keywords for these shapes are RECT, OVAL, and ROUNDRECT, respectively. Figure 16-1 shows examples of all three shapes.

You specify the size and location of each shape by using two points to designate the top left and bottom right corners of the rectangle that contains the shape. This rectangle is sometimes called the *bounding rectangle.* Squares and circles are merely those special cases of rectangles and ovals in which the sides of the bounding rectangle are of equal length.

The ROUNDRECT shape requires a third set of coordinates, preceded by the keyword WITH, to indicate the degree of roundness to be used for the corners of the rectangle. The curve used for

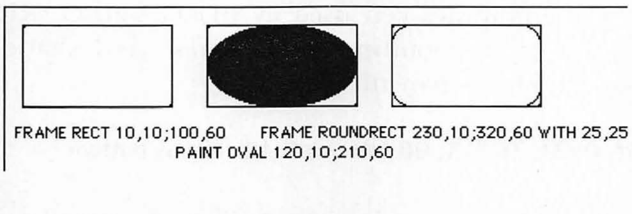
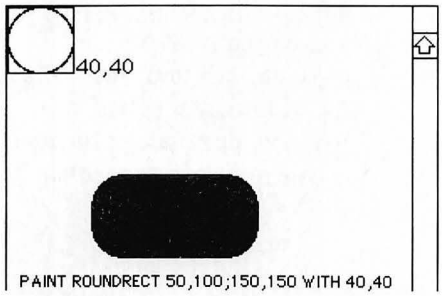


Figure 16-1. Shapes in their rectangles



```
PAINT ROUNDRECT 50,100;150,150 WITH 40,40
```

Figure 16-2. ROUNDRECT

the corners of the rounded rectangle is the curve of the oval that fits in a rectangle whose top left corner is at the point 0,0 and whose bottom right corner is the point specified by the third set of coordinates. Figure 16-2 shows a rounded rectangle and the hypothetical rectangle that defines the degree of roundness.

DISPLAYING SHAPES

■ FRAME, PAINT, ERASE, INVERT

The four commands that display a shape are FRAME, PAINT, ERASE, and INVERT. To display a shape, use one of these four commands followed by the name of the shape you want to display (RECT, OVAL, or ROUNDRECT) and the coordinates that describe the shape. The shape display commands do not affect the location of the graphics pen used by PLOT and GPRINT.

FRAME draws the outline of the prescribed shape with the graphics pen. The statement

```
FRAME OVAL 30,30; 90,90  ! left,top; right,bottom
```

draws the outline of an oval inside the bounding rectangle with its top left corner at 30,30 and its bottom right corner at 90,90.

PAINT draws a solid shape filled with the graphics pen pattern (solid black unless you use the SET PATTERN command to change it). The statement

```
PAINT RECT 10,20; 90,110  ! left,top; right,bottom
```

draws a solid black rectangle with the top left corner at 10,20 and the bottom right corner at 90,110.

ERASE fills the shape with the background pattern. The background pattern is solid white unless you use the toolbox routine BackPat described in Chapter 22 to change it. The statement

```
ERASE RECT 1,1; 200,200  ! left,top; right,bottom
```

erases all graphics and text from the rectangle whose top left corner is at 1,1 and whose bottom right corner is at 200,200.

INVERT reverses the color of every pixel in the shape—black pixels become white, and white pixels become black. The statement

```
INVERT RECT 30,40; 90,100  ! left,top; right,bottom
```

reverses the color of every pixel in the rectangle whose top left corner is at 30,40 and whose bottom right corner is at 90,100.

CHANGING THE GRAPHICS PEN

The graphics pen is the imaginary pen that moves around the screen carrying out your drawing instructions. Usually, the graphics pen acts like an ordinary pencil or pen—it draws a thin black line on top of whatever was already there. By now you should know, though, that very few things about the Macintosh are truly ordinary. The graphics pen can grow or shrink, change its shape, draw with ink of different colors and patterns (including invisible ink), and it can change the way the ink interacts with what is already on the paper.

The graphics pen does all these things, of course, under the control of the program you write. You can be realistic and stick close to the rules of the ordinary world, or you can be bold and experiment with new sets of rules to see what they do.

Pen Width and Height

■ SET/ASK PENSIZE

When the pen draws a simple line, it is tempting to think of the tip of the pen as tapering to a "point." That "point," however, does have dimensions. It is one pixel wide and one pixel tall. When Macintosh BASIC begins executing a new program, it sets the PENSIZE to 1,1.

You can change the size of the pen's "point" with the SET PENSIZE command. The command should be followed by the desired width and height in pixels, both numeric expressions, with a comma between them. You can use the same number for both dimensions to keep the pen square, or you can make one dimension larger than the other to get a different effect. Making both dimensions zero makes the pen invisible.

```
SET PENSIZE 3,5  ! Sets pen 3 pixels wide by 5 high
SET PENSIZE 0,0  ! Makes it invisible
SET PENSIZE 1,1  ! Sets pen back to normal size
```

The change in pen size will affect all PLOT and FRAME commands. It will not affect GPRINT, PAINT, ERASE, or INVERT. When the pen size is larger than one pixel in either direction, the pen is centered on the coordinates you give in the PLOT command. If you set the pen size to 5,5, the statement PLOT 9,9 draws a solid square 5 pixels on a side with its center on the point 9,9.

When you use the FRAME command to draw a shape, the coordinates you give are the coordinates of the rectangle that encloses the shape. A larger pen size will make FRAME draw a wider line, but the line will still be entirely inside the bounding rectangle. In other words, the shape you draw with FRAME stays the same size, and the extra width of the pen comes entirely from the area inside the shape's outer boundary.

You can use an ASK PENSIZE statement to find the current size of the graphics pen. ASK PENSIZE must be followed by the names of two numeric variables separated by a comma. When it executes this command, BASIC places the width of the graphics pen in the first variable and the height of the graphics pen in the second variable. Executing the statement

```
ASK PENSIZE a,b
```

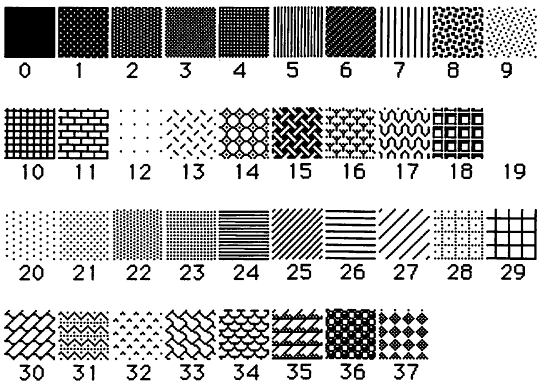


Figure 16-3. Pattern reference chart

saves the width of the graphics pen in the variable *a*, and the height of the pen in *b*.

Pen Pattern

■ SET/ASK PATTERN

Normally, the "ink" from the graphics pen produces a solid black pattern. The SET PATTERN command allows you to select any one of 38 predefined patterns. They are numbered from 0 to 37 and are the same patterns included in MacPaint. Figure 16-3 is a reference chart of the patterns and their numbers. Solid black is pattern 0, and solid white is pattern 19. Pattern 3 is a uniform gray that produces a dotted line when you use the normal pen size of 1,1.

Macintosh BASIC allows you to use names instead of numbers for five of the most commonly used patterns. As shown in Table 16-1, these are BLACK (pattern 0), DKGRAY (pattern 2), GRAY (pattern 3), WHITE (pattern 19), and LTGRAY (pattern 22). These names are actually predefined constants that BASIC turns into the appropriate pattern numbers when you use them in a SET PATTERN statement.

Table 16-1. Names for Common Patterns

Name	Pattern number
BLACK	0
DKGRAY	2
GRAY	3
WHITE	19
LTGRAY	22

To change the pattern, you use the SET PATTERN command followed by one of the five predefined pattern names or a numeric expression representing the pattern number. If the number you supply is not an integer, BASIC rounds it to the nearest integer. The PATTERN setting affects the PLOT, FRAME, and PAINT commands. A pattern number less than 0 or higher than 37 will result in an unpredictable pattern.

```
SET PATTERN 3       ! Sets pattern to gray
SET PATTERN GRAY    ! Also sets it to gray
SET PATTERN 11      ! Sets brick wall pattern
SET PATTERN 0       ! Back to dull black
```

You can use an ASK PATTERN statement to obtain the number of the pen's current pattern. Follow the words ASK PATTERN with the name of a numeric variable. BASIC puts the number of the current pattern into the variable. Executing the statement

```
ASK PATTERN pat     ! Puts pattern number in pat
```

puts the number of the current pattern in the variable *pat*.

Pen Mode

- SET/ASK PENMODE

The pen mode refers to the way the "ink" from the graphics pen interacts with the design already in the output window. The official values of PENMODE range from 8 to 15, but integer multiples of these values often produce the same result. The normal setting for PENMODE when execution of a program begins is 8. The PENMODE setting affects the PLOT, FRAME, and PAINT commands. It has the same effects for shapes as GTEXTMODE does for graphics text.

Figure 16-4 shows the effect of each of the eight pen modes. To read the figure, assume that the design in the top row (horizontal stripe) already appears in the output window and that the design in the middle row (vertical stripe) is the graphics pen pattern being drawn. The design in the bottom row is what results from the operation for that pen mode setting. The final design is determined pixel by pixel, using logical operators, with black pixels corresponding to TRUE and white pixels corresponding to FALSE.

Pen mode 8 is called the copy mode. It ignores the design already in the window and copies the pen pattern exactly. This is the normal mode setting when a program begins execution. Pen mode 9 is the OR mode. A pixel in the result is black if that pixel was black in either the original design or the new pattern.

Pen mode 10 corresponds to the exclusive OR operation (abbreviated XOR). A pixel is black if the background and the new pattern are opposite colors, and it is white if the background and the new pattern are the same color. This mode allows the black portion of the pen pattern to be discernible on a background of any color.

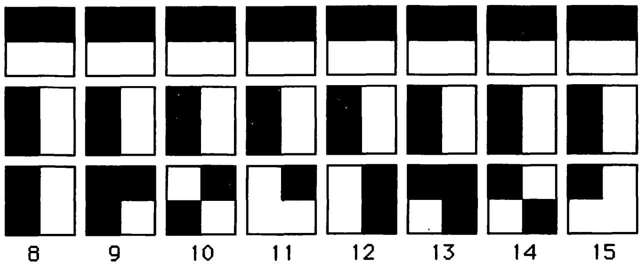


Figure 16-4. Effects of the pen modes

Writing the same pattern for a second time in XOR mode restores the background design to its original state. XOR mode is often used for dotted lines with the GRAY pen pattern.

Pen mode 11 is a clear or erase mode. The pixels behind the white areas in the pen pattern do not change, but the pixels that are beneath black pixels are changed to white. The results of modes 12 through 15 appear close to the results of modes 8 through 11. Modes 12 through 15 are called *inverted modes*. Their results are obtained by reversing the color of every pixel in the pen pattern and then applying the rules for modes 8 through 11.

Here are some samples of the syntax for SET PENMODE and ASK PENMODE:

```
SET PENMODE 10 ! Sets pen to XOR mode
ASK PENMODE x  ! Puts 10 in x
SET PENMODE 14 ! Sets pen to inverse XOR mode
SET PENMODE 8  ! Restores normal copy mode
```

Resetting the Pen

- PENNORMAL

When you start your BASIC program running, the pen pattern is BLACK (pattern 0), the pen mode is 8 (copy mode), and the pen size is 1,1. Those three settings are used for most normal drawing. You can use the PENNORMAL command to restore all three settings at once. The effect of using PENNORMAL is the same as executing the three statements SET PATTERN BLACK, SET PENMODE 8, and SET PENSIZE 1,1.

```
SET PENSIZE 3,3 ! Makes the pen large
PENNORMAL       ! Resets PENSIZE to 1,1
SET PATTERN 18
SET PENMODE 9
PENNORMAL       ! Resets PATTERN to 0 and PENMODE to 8
```

MOVING SHAPES

You can move graphics shapes or even text by erasing and redrawing. FOR/NEXT loops are especially handy for controlling continuous movement. You now know all the commands to understand the following graphics program (first shown in Figure 1-1).

```
! Make moving design
DO
   FOR i = 1 TO 500
      PAINT RECT i,30;i+120,150
      INVERT OVAL i,30;i+120,150
   NEXT i

   FOR i = 500 TO 1 STEP -1
      PAINT OVAL i,40;i+100,140
      INVERT RECT i,40;i+100,140
   NEXT i
LOOP
```

The time it takes to draw large graphics on the screen, even at QuickDraw's impressive speed, slows down the program. Try reducing the size of the moving shapes and see how much faster the program runs. Start the debugger and use the Step and Trace commands to see what each statement does and how the statements combine to form a moving pattern.

ALTERING THE DISPLAY

Macintosh BASIC allows you to change four fundamental settings that govern your program's output display. You can change the location and size of the output window, the document behind the output window, and the graphics area within the output document. You can also change the scale of your graphics output.

Changing the Output Window

- SET/ASK OUTPUT

The SET OUTPUT command determines the size and location of your program's output window. The statement SET OUTPUT TOSCREEN enlarges the output window to occupy the full screen below the menu bar. The edges of the window remain visible on the screen so you can use the title bar, close box, size box, and scroll bars. The statement SET OUTPUT without any additional words resets the output window to its original size and location.

```
SET OUTPUT TOSCREEN
    ! Enlarges output window to full screen size
SET OUTPUT
    ! Resets window to original size and location
```

If you want to choose a different window size or location, you need to supply two points. The punctuation is the same as you use for FRAME RECT and other shape-drawing commands, but the points you supply for SET OUTPUT are different. SET OUTPUT requires the bottom left and top right points instead of the top left and bottom right used by the drawing commands. In addition, the coordinates of the points for SET OUTPUT must be given in inches—not in pixels—from the upper left corner of the screen. There are 72 pixels per inch on the normal 9 inch Macintosh screen.

```
SET OUTPUT left,bottom; right,top
SET OUTPUT 0,3; 4,1
    ! Makes window 4" wide and 2" high
    ! Positioned on left edge of screen
```

When you use SET OUTPUT to move or resize your program's output window, the window first appears in its regular size and location and then moves to its new size and location. If the SET OUTPUT command changes the size of the output window, the contents of the window are erased. The coordinates of the original output window on the 9-inch screen are 3.333,3.889 and 6.667,0.556, and the coordinates that are supplied if you use SET OUTPUT TOSCREEN are 0,4.528 and 6.889,0.527.

ASK OUTPUT followed by two pairs of numeric variable names gets the screen coordinates of the current output window in inches. The variable names in each pair are separated by commas, and a semicolon is used between the pairs:

```
ASK OUTPUT left,bottom; right,top
    ! Gets window coordinates
```

Dimensioning the Output Document

■ SET/ASK DOCUMENT

SET DOCUMENT resets the size and location of the document that lies behind the output window, and ASK DOCUMENT saves the current settings in variables for you. The order of the coordinates is just like SET OUTPUT, with each coordinate measured in inches. When your program begins, the document is set at 0,11; 8.5,0 for the size of a normal sheet of paper. You can reset it to this size by using just the words SET DOCUMENT. If a SET DOCUMENT command changes the size of the output document, the contents of the document are erased.

In addition to limiting the area in which your program can draw graphics, SET DOCUMENT also affects the number of lines of text the output document can hold. If your program prints more lines than the document can hold, BASIC cuts off a line from the top of the document to make room for each new line printed. The original document height of 11 inches holds about 49 lines of 12 point type.

```
SET DOCUMENT 0,23; 8.5,0
    ! Enlarges document to hold 100 lines of 12-point text
ASK DOCUMENT left,bottom; right,top
    ! Gets document coordinates
SET DOCUMENT
    ! Restores document to original size
```

Setting the Graphics Area

■ SET/ASK LOCATION

SET LOCATION sets the area of the output document in which graphics are drawn. ASK LOCATION puts those coordinates into variables. Both commands use coordinates measured in inches in the order left, bottom, right, top. The graphics area is set automatically to the full size of the output document when your program begins and whenever a SET OUTPUT statement changes the size of the output window. SET LOCATION TOWINDOW sets the graphics area to match the output window's size and position. SET

LOCATION without any other words resets the graphics area to coincide with the output document.

You can think of the edges of the graphics area as if they were a picture frame. That frame is used when you copy a picture to the Clipboard. If your picture is to fit into the Scrapbook desk accessory, the location into which you draw the picture cannot be more than 2.5 inches high or 4.5 inches wide. If you want to paste the picture into MacPaint, the location cannot be more than 3.5 inches high or 5 inches wide.

```
SET LOCATION 0,4; 4,0
    ! Sets graphics area to a 4" square at upper left of document
ASK LOCATION left,bottom; right,top
    ! Gets the current location of the graphics area in inches
SET LOCATION TOWINDOW
    ! Sets the graphics area to the output window
SET LOCATION
    ! Sets the graphics area to the entire document
```

Setting Your Own Graphics Scale

- SET/ASK SCALE

SCALE specifies the logical picture frame your program uses when it draws a picture. Unless you are doing something fancy, SCALE is set to the same coordinates as the output document. The coordinates for SET/ASK SCALE are written in the same order as the coordinates for SET/ASK LOCATION. SCALE's coordinates, however, are measured in screen pixels instead of inches.

When your program begins, scale is set to 0,792; 612,0, which is the equivalent in pixels of 8 1/2 by 11 inches. If you use SET SCALE without coordinates, BASIC resets SCALE to the original 0,792; 612,0 setting.

SCALE's coordinates describe the picture frame that the graphics commands in your program use when drawing. If the SCALE coordinates are different from the LOCATION coordinates, BASIC shifts SCALE's picture frame until its top left corner matches the same corner of the location picture frame and then expands or contracts the dimensions of SCALE's frame until they match the frame specified by LOCATION. All of the graphics inside the scaled frame expand or contract along with the frame. After locations inside the frame are rescaled, they are rounded to integers if necessary.

To enlarge a drawing without changing its proportions, make the width and height of the graphics area proportional to the width and height of the drawing's scale. If you switch the two horizontal coordinates or the two vertical coordinates, SCALE will flip the picture in that direction.

```
SET SCALE left,bottom; right,top
    ! Dimensions in pixels, not inches
SET SCALE 0,72; 72,0
    ! Causes drawing in upper left 1" of
    ! document to be scaled to the graphics location
ASK SCALE left,bottom; right,top
    ! Gets the current scale setting in pixels
SET SCALE left,top; right,bottom
    ! Flips the picture vertically
```

HOW BASIC HANDLES PICTURES

■ SET/ASK PICSIZE

BASIC stores the graphics portion of your program's output document as a picture. Technically, a QuickDraw picture is not a copy of what you see on the screen, but a record of the actions necessary to recreate that image. For each running program, BASIC allocates an area of memory 2048 bytes long. This area is called the *picture buffer.*

Each time your program uses GPRINT or a graphics-drawing command, a record of the command is added to the information stored in the picture buffer. When the picture buffer is filled, BASIC still executes graphics commands but stops adding new information to the picture buffer. The CLEARWINDOW command erases the picture buffer in addition to the output document.

When part of the output document is hidden and then becomes visible again, BASIC uses the information in the picture buffer to redraw the picture. If your program uses a lot of graphics commands, you will notice that sometimes only part of the picture is redrawn. That is because the picture buffer had room for only part of your picture. The same thing can happen when you use Copy Picture in the Edit menu to copy a picture to the Clipboard.

You can change the size of the picture buffer with the SET PICSIZE command followed by a numeric expression that gives the number of bytes you want included in the buffer. If you try to set the buffer smaller than 2048 bytes, BASIC will set the buffer size to 2048. You can set the picture buffer size as large as 32,767 bytes. The ASK PICSIZE statement followed by the name of a numeric variable tells BASIC to put the size of the existing buffer in that variable.

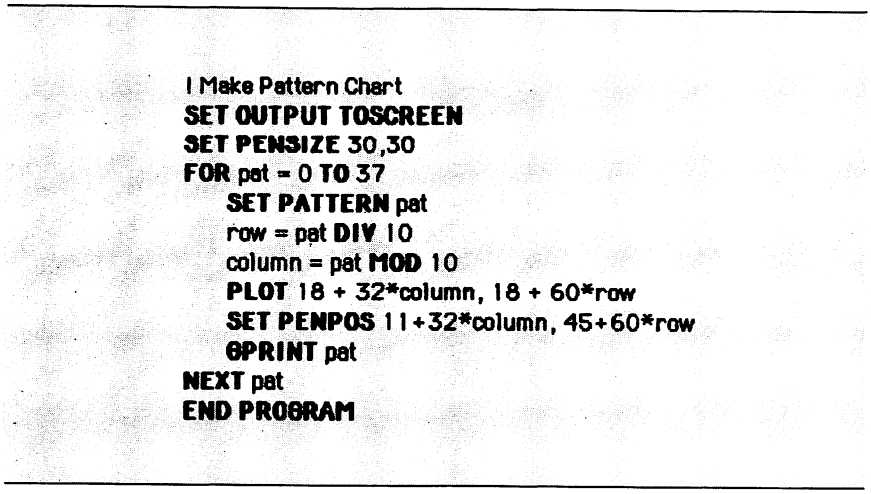
```
SET PICSIZE 4096
    ! Doubles original size of picture buffer
ASK PICSIZE sze
    ! Puts picture buffer size in sze
SET PICSIZE 32767
    ! Sets picture buffer as large as possible
SET PICSIZE 2048
    ! Restores original picture buffer size
```

EXAMPLE PROGRAMS

The examples in this section are the programs that produced the pattern and pen mode reference charts in Figures 16-3 and 16-4. In addition to demonstrating many of the commands introduced in this chapter, these programs will allow you to make extra copies of the reference charts so you can always have them handy.

The Make Pattern Chart program in Figure 16-5 draws each sample pattern by using the PLOT command to plot one point with an oversized pen. After using a SET OUTPUT TOSCREEN statement to enlarge the output window to full screen size, the program sets the pen size to a square 30 pixels on each side.

A FOR/NEXT loop steps through the patterns from 0 to 37. The DIV operator is used to calculate the row in which the pattern should appear. The MOD operator is used to calculate the number of the column in which the pattern should appear. If the pattern number is less than 10, the result of *pat* DIV 10 will be zero, so the



```
! Make Pattern Chart
SET OUTPUT TOSCREEN
SET PENSIZE 30,30
FOR pat = 0 TO 37
    SET PATTERN pat
    row = pat DIV 10
    column = pat MOD 10
    PLOT 18 + 32*column, 18 + 60*row
    SET PENPOS 11+32*column, 45+60*row
    GPRINT pat
NEXT pat
END PROGRAM
```

Figure 16-5. The Make Pattern Chart program

rows are numbered from 0 to 2. The lowest result of *pat* MOD 10 is also zero, so the columns are numbered from 0 to 9.

The PLOT statement calculates the location in the output window for each pattern by starting at the point 18,18 (a 3 pixel margin from the edge of the window plus 15 to get to the center of the first 30 pixel square). The number 32 is added to the horizontal coordinate for each column, and 60 is added to the vertical coordinate for each row. Those numbers include room for the 30 pixel square pattern plus the desired amount of white space between patterns. After each pattern "point" is plotted, the SET PENPOS statement positions the graphics pen under the pattern and the GPRINT statement prints the pattern number.

The Make PENMODE Demonstration Chart program in Figure 16-6 is slightly longer, but not much more complicated. The loop that starts with FOR mode = 8 TO 15 controls the printing of the reference chart. The loop is executed once for each column in the reference chart. The first statement inside the loop calculates the horizontal displacement of the column to be printed, with each column allocated 48 pixels. The displacement is calculated once for each column and stored in the variable *horiz* instead of being calculated in every graphics statement in the loop.

```
! Make PENMODE Demonstration Chart
SET OUTPUT TOSCREEN
! Print the illustration
FOR mode = 8 TO 15
    horiz = 48 * (mode - 8)
    ! Print sideways background pattern
    PAINT RECT 8+horiz,8;48+horiz,28
    PAINT RECT 8+horiz,104;48+horiz,124
    ! Print black half of vertical mode pattern
    PAINT RECT 8+horiz,56;28+horiz,96
    SET PENMODE mode
    PAINT RECT 8+horiz,104;28+horiz,144
    ! Print white half of vertical mode pattern
    SET PATTERN WHITE
    PAINT RECT 28+horiz,104;48+horiz,144
    ! Draw frames around all the squares
    PENNORMAL  ! Back to black and copy mode
    FRAME RECT 7+horiz,7;49+horiz,49
    FRAME RECT 7+horiz,55;49+horiz,97
    FRAME RECT 7+horiz,103;49+horiz,145
    ! Print the mode numbers
    GPRINT AT 20+horiz,160; mode
NEXT mode
END PROGRAM
```

Figure 16-6. The Make PENMODE Demonstration Chart program

This program leaves the pen at its normal size 1,1 and uses PAINT RECT to draw the patterns in three rectangles in each column. The top row represents the background design, the middle row represents the pattern drawn with each pen mode setting, and the bottom row represents the result. The first two PAINT RECT statements draw horizontal black stripes to represent the background design in the upper half of the top and bottom rectangles. Then another PAINT RECT statement draws a vertical stripe in the left half of the rectangle in the middle row to represent the pattern being drawn with the pen mode setting.

After drawing the stripe in the middle row, the program uses a SET PENMODE statement to set the pen mode to the mode being demonstrated. Then it uses that pen mode to paint black on the left half and white on the right half of the rectangle in the bottom row.

Now the program has only a little straightening up to do before repeating the loop for the next column. The PENNORMAL statement sets the pen mode and pattern back to normal before the program draws frames around the three designs in the column. The frames are drawn after the contents of the rectangles so the frames will not be affected by the unusual pen modes.

GPRINT is used, once again, to print the proper identification under the column. The printing location is specified by the AT clause in the GPRINT statement.

PRACTICE EXERCISES

1. Write a program to draw a line 5 pixels wide from 30,30 to 90,12.
2. Can you write statements to outline an oval with a gray line 6 pixels wide? Let the top left corner of its bounding rectangle be 10,10 and the bottom right corner 80,200.
3. Draw a black square bounded by a rectangle whose top left corner is at 10,10 and whose bottom right corner is at 200,200 and put a white circle inside it. The circle can be bounded by the rectangle whose top left corner is at 30,30 and whose bottom right corner is at 180,180.
4. How would you erase text that is located in a rectangle whose bottom right corner is 200,90 and whose top left corner is 10,10?

Chapter 17
Using the Mouse

Command:

- BTNWAIT

System functions:

- MOUSEH, MOUSEV, MOUSEB, MOUSEB~

The mouse is one of the key elements of the Macintosh user interface. It is most useful as a pointing device. To select a command from a menu, you point to it instead of typing the command. To mark a position on the screen, you point to it instead of using control keys to move the cursor.

Using the mouse in your programs is not difficult. You need to know only two pieces of information about the mouse: where on the screen the mouse's pointer, the cursor, is located, and whether its button is up or down. This chapter shows you how to get that information and how to use it in your programs.

FINDING THE MOUSE

■ MOUSEH, MOUSEV

MOUSEH and MOUSEV are numeric system functions that tell you where the cursor is located in your program's output window. Both take no arguments. MOUSEH gives the horizontal position of the cursor, and MOUSEV gives the vertical position. Both positions are given as the number of screen pixels from the upper left corner of the window's output document.

Unless you change it, the size of Macintosh BASIC's output window is 240 pixels in each direction. If MOUSEH is greater than 240, the cursor will probably be located to the right of the window's normal viewing area. If MOUSEH is less than zero, the cursor is located to the left of the window. Similarly, if MOUSEV is greater than 240, the cursor is probably below the window, and if MOUSEV is less than zero, the cursor is above the top of the window. If you use a SET OUTPUT statement to change the size of the output window, the window's new height and width is measured in pixels (72 pixels per inch).

You should keep in mind that the output window has scroll bars. What you see in the window frame is only a portion of the output document. If you have used the scroll bars or if your program has printed something outside the original viewing area, the upper left corner of the output document may not coincide with the upper left corner of the window frame. When the two do not coincide, the MOUSEH and MOUSEV functions return the position of the cursor with respect to the upper left corner of the document behind the window, even if that corner has been scrolled outside the visible area.

The following short example uses MOUSEH, MOUSEV, and the PLOT command to turn the cursor into a drawing tool. The semicolon at the end of the PLOT command makes your drawing a continuous line by plotting a line instead of a single point. Without the semicolon, the program would leave gaps whenever you moved the cursor faster than BASIC could plot all the individual points.

```
! Draw with mouse pointer
DO
    PLOT MOUSEH, MOUSEV;
LOOP
```

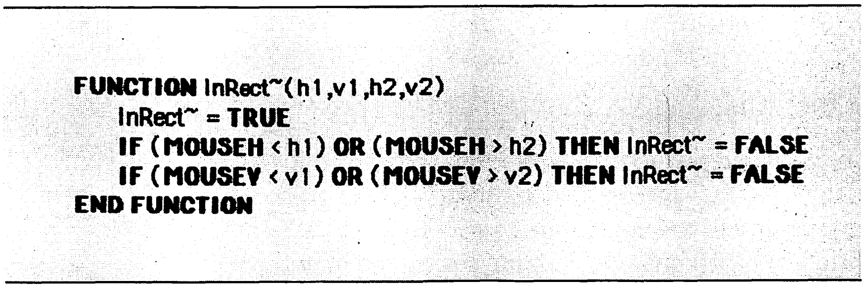
TESTING FOR PROXIMITY

Your program needs to know more than just the location of the cursor in the window. It also needs to know whether the cursor is within a specific area, like a rectangle, or close to a certain point. If, for instance, you want to give the program instructions by clicking the mouse button with the cursor in a specific location, the program should not require the click to be at a single point because you could waste time trying to position the cursor on a single screen pixel. Instead, the program should accept clicks within a reasonably sized area of the screen.

Because tests of the cursor location are used so often, it makes sense to define them as functions. The function InRect~ in Figure 17-1 is a Boolean function that tests whether or not the cursor is located inside the rectangle whose top left point is at *h1,v1* and whose bottom right point is at *h2,v2*.

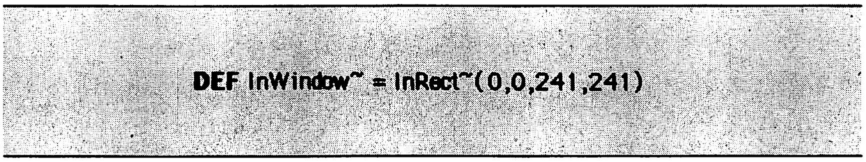
The first line after the function definition sets the value of the function to true. The second line tests whether the horizontal position, MOUSEH, is in the rectangle and changes the value of the function to false if it is not. The third line changes the value of the function to false if MOUSEV is not in the rectangle.

The InRect~ function returns a value of true if the cursor is located within the specified rectangle and false if it is not. You can use the returned value in additional function definitions as well as in your main program. For instance, the statement in Figure 17-2 uses the InRect~ function in the definition of a new function InWindow~, which returns the value true if the cursor is in the normal output window and false if it is not.



```
FUNCTION InRect~(h1,v1,h2,v2)
    InRect~ = TRUE
    IF (MOUSEH < h1) OR (MOUSEH > h2) THEN InRect~ = FALSE
    IF (MOUSEV < v1) OR (MOUSEV > v2) THEN InRect~ = FALSE
END FUNCTION
```

Figure 17-1. In Rect ~ function



```
DEF InWindow~ = InRect~(0,0,241,241)
```

Figure 17-2. InWindow function

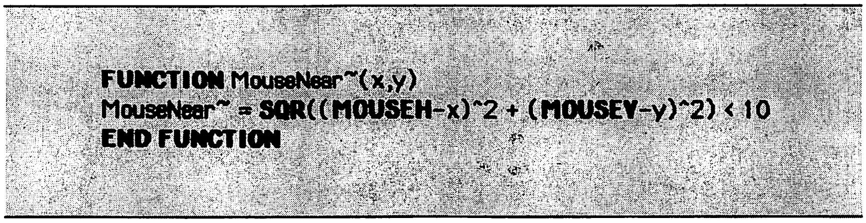
The shape most appropriate for measuring the proximity to a single point is a circle centered on the point. The radius of the circle is the number of pixels away from the point you will accept as a "hit" or a "match" for that point. The Boolean function MouseNear~ in Figure 17-3 returns true if the cursor is less than 10 pixels away from the target point and returns false if the cursor is 10 or more pixels from the point.

You can, of course, use a different number in place of 10 in the MouseNear~ function if you want to require greater or less precision in the positioning of the cursor. You can also let your program perform calculations with the actual distance of the cursor from your target point. If you want to do this, use the numeric function MouseNear in Figure 17-4, which returns the distance between the two points.

GETTING INPUT FROM THE MOUSE

■ MOUSEB, MOUSEB~

MOUSEB and MOUSEB~ are system functions that tell your program whether or not the mouse button is down. Neither function takes any arguments. MOUSEB returns a number, and MOUSEB~



```
FUNCTION MouseNear~(x,y)
MouseNear~ = SQR((MOUSEH-x)^2 + (MOUSEV-y)^2) < 10
END FUNCTION
```

Figure 17-3. MouseNear ~ function

```
FUNCTION MouseNear(x,y)
MouseNear = SQR((MOUSEH-x)^2 + (MOUSEV-y)^2)
END FUNCTION
```

Figure 17-4. MouseNear function

returns a Boolean value. When the mouse button is down, MOUSEB returns 1 and MOUSEB~ returns true. When the mouse button is not down, MOUSEB returns 0 and MOUSEB~ returns false. You can use either function when you want to test whether the button is down:

```
IF MOUSEB=1 THEN PRINT "The button is down."
IF MOUSEB~ THEN PRINT "The button is down."
```

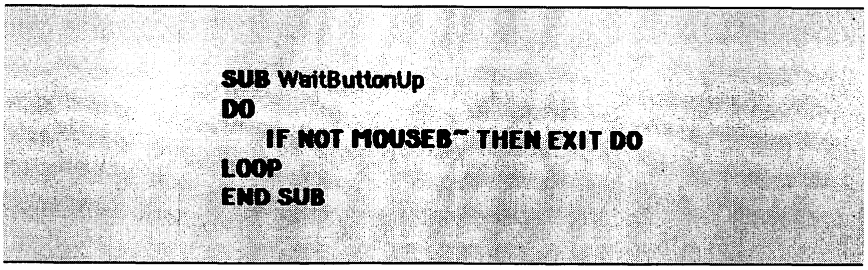
The processing speed inside the Macintosh is much faster than the reponse time of either people or mechanical devices like mouse buttons. If you are writing a program that is supposed to take an action like plotting a point each time the mouse button is pressed, your program will be fast enough to take the action several times during a single press of the mouse button. To ensure that the action is recorded only once for each press of the mouse button, your program needs to wait for the mouse button to return to the up position after it completes the action.

Your program can use either MOUSEB or MOUSEB~ to test whether the mouse button has been released. The loops

```
DO
   IF NOT MOUSEB~ THEN EXIT DO
LOOP
```

and

```
DO
   IF MOUSEB = 0 THEN EXIT DO
LOOP
```



```
SUB WaitButtonUp
DO
    IF NOT MOUSEB~ THEN EXIT DO
LOOP
END SUB
```

Figure 17-5. WaitButtonUp subroutines

are equivalent ways to wait for the mouse button to be released. The appropriate structure to use for these loops is a subroutine, not a function, because the loops actually wait for an action to occur instead of merely returning a value. The subroutine Wait-ButtonUp in Figure 17-5 can be called from your main program with the statement CALL WaitButtonUp.

WAITING FOR THE MOUSE BUTTON

- BTNWAIT

The BTNWAIT command is a shortcut. It causes the program to wait until the mouse button is pushed in the program's output window. If the mouse button is already down when BTNWAIT is executed, it waits until the button is released and pushed again. BTNWAIT is a shorter way to execute this program segment:

```
! Wait for mouse click in window
IF MOUSEB~ THEN
    DO  ! Wait for button up if it was down
        IF NOT MOUSEB~ THEN EXIT DO
    LOOP
ENDIF
DO
    IF MOUSEB~ AND InWindow~ THEN EXIT DO
LOOP
END PROGRAM
```

```
FUNCTION InWindow~
    InWindow~ = TRUE
    IF (MOUSEH<0) OR (MOUSEH>240) THEN InWindow~=FALSE
    IF (MOUSEV<0) OR (MOUSEV>240) THEN InWindow~=FALSE
END FUNCTION
```

The BTNWAIT command can be inserted into your program at any point where you want the program to wait for the mouse button to be pushed. Your program should, of course, print a message to tell you why it is waiting instead of just stopping abruptly with no warning. Since a message should be printed every time BTNWAIT is used, the two actions can be combined in a subroutine. This subroutine can then be called with a single statement whenever the PRINT/BTNWAIT combination is desired. The subroutine in Figure 17-6, which can be executed by the statement CALL ButtonWait, is one way to do this.

EXAMPLE PROGRAMS

The Pattern Changer program in Figure 17-7 changes the pattern of a rectangle if the mouse button is clicked while the cursor is inside the rectangle. The program combines the use of MOUSEB~, the function InRect~, and the subroutine WaitButtonUp.

The program begins by setting the variables *h1, v1, h2,* and *v2* to the values that define the rectangle to be displayed. Then it initializes the variable *pat* at zero, uses a SET PATTERN statement to initialize the graphics pattern, and paints the display rectangle.

```
SUB ButtonWait
    PRINT "Click mouse button to continue..."
    BTNWAIT
END SUB
```

Figure 17-6. ButtonWait subroutine

```
! Pattern changer
! Changes pattern if mouse button is clicked
!    inside the rectangle.
h1 = 20
v1 = 20
h2 = 100
v2 = 100
pat = 0
SET PATTERN pat
PAINT RECT h1,v1; h2,v2
SET PENPOS 20,130
GPRINT "Click in pattern to change it."
DO
    IF MOUSEB~ THEN
        IF InRect~(h1,v1,h2,v2) THEN
                pat = pat + 1
                IF pat > 37 THEN pat = 0
                SET PATTERN pat
                PAINT RECT h1,v1; h2,v2
        ENDIF
        CALL WaitButtonUp
    ENDIF
LOOP
END PROGRAM
FUNCTION InRect~(h1,v1,h2,v2)
    InRect~ = TRUE
    IF (MOUSEH < h1) OR (MOUSEH > h2) THEN InRect~ = FALSE
    IF (MOUSEV < v1) OR (MOUSEV > v2) THEN InRect~ = FALSE
END FUNCTION
SUB WaitButtonUp
    DO
        IF NOT MOUSEB~ THEN EXIT DO
    LOOP
END SUB
```

Figure 17-7. Pattern changer

The SET PENPOS statement positions the graphics pen 30 pixels below the left edge of the rectangle, and the GPRINT statement prints a message there so you know what the program is waiting for you to do.

The rest of the program, with the exception of the function definition, is contained in a single DO loop that in turn contains a multiple-line IF statement. The statements inside the multiple-line IF statement are executed only if MOUSEB~ is true—that is, if the mouse button is down. If the mouse button is not down, the loop just keeps repeating until the button is pressed.

When the mouse button is pressed, the statements inside the IF MOUSEB~ statement are executed. The first of these is another multiple-line IF statement that uses the InRect~ function to test whether the cursor is inside the display rectangle. If the cursor is inside the rectangle, the program adds one to the variable *pat,* sets the new pattern, and repaints the rectangle. Note that if the variable *pat* reaches 37, the number of the last defined pattern, the program resets it to pattern 0. The CALL to the subroutine WaitButtonUp then causes the program to wait until the mouse button has been released before repeating the loop.

The Roundrect Changer program in Figure 17-8 allows you to see the effects of various WITH coordinates on the shape of a rounded rectangle. The program displays a painted ROUNDRECT plus the rectangle that defines the roundness of the ROUNDRECT's corners. When you use the mouse to drag the bottom right corner of the roundness rectangle to a different location, the program redraws both rectangles and displays the new coordinates of the bottom right corner of the roundness rectangle. Figure 17-9 shows two output windows with the roundness rectangle set to different shapes.

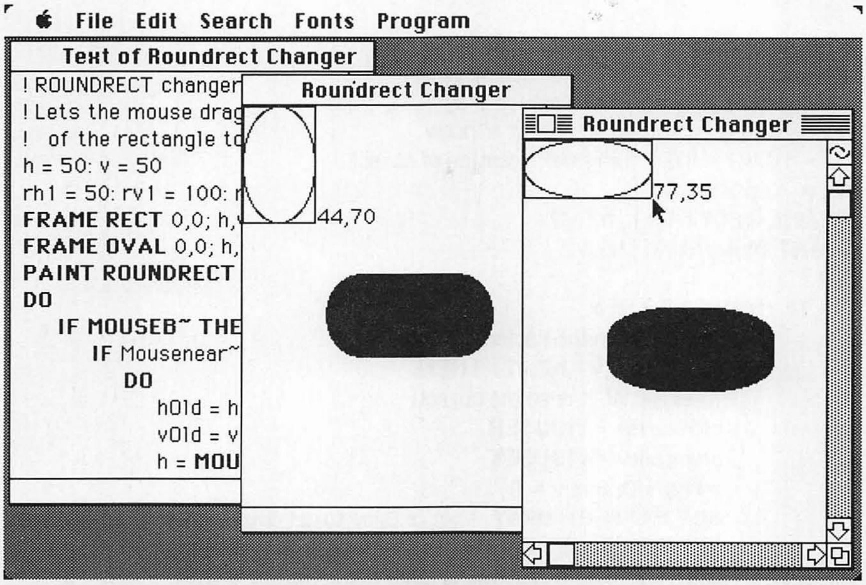
After initializing variables and drawing the objects in their starting locations, the program enters a DO loop, which tests whether the mouse button is down. If the button is down, the program uses the function Mousenear~ to test whether the mouse is close to the movable corner of the rectangle. If the mouse is close enough to the point, the program executes a DO loop that keeps erasing and redrawing the rectangle and oval until the mouse button is released, and then the program repaints the rounded rectangle. If the mouse is not close enough to the starting point, an ELSE statement calls the subroutine WaitButtonUp to wait for the mouse button to be released.

Note that the MouseNear~ function tests whether the cursor is within ten pixels of the point passed as an argument to the function. Ten pixels is a fairly long distance. It was chosen in this

```
! ROUNDRECT changer
! Lets the mouse drag right bottom corner
!   of the rectangle to change the shape.
h = 50: v = 50
rh1 = 50: rv1 = 100: rh2 = 150: rv2 = 150
FRAME RECT 0,0; h,v
FRAME OVAL 0,0; h,v
PAINT ROUNDRECT rh1,rv1; rh2,rv2 WITH h,v
DO
    IF MOUSEB~ THEN
        IF Mousenear~(h,v) THEN
            DO
                hOld = h
                vOld = v
                h = MOUSEH: v = MOUSEV
                IF v <> vOld OR h <> hOld THEN
                    CLEARWINDOW
                    FRAME RECT 0,0; h,v
                    FRAME OVAL 0,0; h,v
                    SET PENPOS h,v
                    GPRINT h; ","; v;
                    ENDIF
                IF NOT MOUSEB~ THEN EXIT DO
            LOOP
            PAINT ROUNDRECT rh1,rv1; rh2,rv2 WITH h,v
        ELSE  ! MouseNear~ is FALSE
            CALL WaitButtonUp
        ENDIF  ! MouseNear~
    ENDIF  ! MOUSEB~
LOOP
END PROGRAM
FUNCTION MouseNear~(x,y)
MouseNear~ = SQR((MOUSEH-x)^2 + (MOUSEV-y)^2) < 10
END FUNCTION
SUB WaitButtonUp
    DO
        IF NOT MOUSEB~ THEN EXIT DO
    LOOP
END SUB
```

Figure 17-8. Roundrect changer

instance to make the program easy to use. You can, if you wish, select a shorter distance.



```
! ROUNDRECT changer
! Lets the mouse drag
!  of the rectangle to
h = 50: v = 50
rh1 = 50: rv1 = 100:
FRAME RECT 0,0; h,
FRAME OVAL 0,0; h,
PAINT ROUNDRECT
DO
   IF MOUSEB~ THE
      IF Mousenear~
         DO
            hOld = h
            vOld = v
            h = MOU
```

Figure 17-9. Running the Roundrect Changer program

The example in Figure 17-10 shows how you can use the mouse to drag an object from one location to another in the output window. A dotted outline of the object follows the mouse movements while you hold the button down, and the entire object follows as soon as you release the button. The impression that the dotted line is moving is created by erasing and redrawing it very quickly, changing the location each time to match the movement of the mouse.

The program frames a rectangle and paints an oval in it to create the object to be moved. Then it enters the familiar DO loop with multiple-line IF statements. If the mouse button is pressed in the rectangle, the program saves the original mouse position in variables named *oldmouseh* and *oldmousev*. These variables are used later to calculate how far the mouse has been moved from its original position. The distance the mouse has been moved is kept in *incrh* and *incrv*, so they are initialized to zero. The program sets the graphics pen pattern to gray to get a dotted line and sets the pen mode to 10 so the dots in the line will be visible against any background.

```
! Object mover
! Shows how you can let the mouse drag
!   objects around in your output window.
h1 = 10: v1 = 10  ! "original" location of object
h2 = 70: v2 = 50
FRAME RECT h1,v1; h2,v2
PAINT OVAL h1,v1; h2,v2
DO
    IF MOUSEB~ THEN
        ! Is the mouse in the rectangle?
        IF InRect~(h1,v1,h2,v2) THEN
            ! Yes, so let it drag the object.
            oldmouseh = MOUSEH
            oldmousev = MOUSEV
            incrh = 0: incrv = 0
            SET PATTERN GRAY  ! Set to Gray to get dotted line
            SET PENMODE 10  ! Set to XOR to see line everywhere
            DO    ! Move the outline around while button is down.
                IF MOUSEH <> oldmouseh OR MOUSEV <> oldmousev THEN
                    FRAME RECT h1+incrh,v1+incrv; h2+incrh,v2+incrv
                                ! Draw dotted line in new location
                    FRAME RECT h1+incrh,v1+incrv; h2+incrh,v2+incrv
                                ! XOR on top of the line to erase it
                    incrh = MOUSEH - oldmouseh
                    incrv = MOUSEV - oldmousev
                ENDIF
                IF NOT MOUSEB~ THEN EXIT DO
            LOOP
            ! Mouse button is up now, so move the object.
            ERASE RECT h1,v1; h2,v2
            h1 = h1 + incrh
            h2 = h2 + incrh
            v1 = v1 + incrv
            v2 = v2 + incrv
            PENNORMAL
            FRAME RECT h1,v1; h2,v2
            PAINT OVAL h1,v1; h2,v2
        ELSE CALL WaitForMouseUp
        ENDIF ! InRect~
    ENDIF ! MOUSEB~
```

Figure 17-10. Object mover

```
LOOP
END PROGRAM
FUNCTION InRect~(h1,v1,h2,v2)
    InRect~ = TRUE
    IF (MOUSEH < h1) OR (MOUSEH > h2) THEN InRect~ = FALSE
    IF (MOUSEV < v1) OR (MOUSEV > v2) THEN InRect~ = FALSE
END FUNCTION
SUB WaitForMouseUp
    DO
        IF NOT MOUSEB~ THEN EXIT DO
    LOOP
END SUB
```

Figure 17-10. Object Mover (*continued*)

A DO loop then controls the erasing and redrawing of the dotted outline until releasing the mouse button causes an exit from the loop. An IF statement allows the erasing and redrawing to occur only if the mouse has been moved from its original position. This prevents the dotted outline from appearing until the movement begins. The first FRAME RECT statement in the loop draws a dotted outline, and the second FRAME RECT statement immediately erases it. The program then calculates a new location from MOUSEH and MOUSEV and repeats the process until the mouse button is released.

Releasing the mouse button allows the program to move past the end of the DO loop. An ERASE RECT statement erases the original rectangle, using the carefully preserved variables that marked the original rectangle's corners. Then the program adds *incrh* to the horizontal values and *incrv* to the vertical values to change the "original" location to the new location.

The program uses a PENNORMAL statement to reset the pen pattern to black and the pen mode to 8 (normal copy mode) before drawing the object in its new location. Because the variables that contain the original position of the rectangle (*h1,v1*; *h2,v2*) are updated to reflect the new "original" position of the rectangle, the loop can be executed and the object moved again without any further preparations.

PRACTICE EXERCISES

1. Write a loop that puts a dot at each place where the mouse button is pushed down. Don't worry about erasing any of the dots.
2. Write a loop that draws a line from each place where the mouse button is clicked to the next place the mouse button is clicked. This loop will allow you to draw a design by clicking the mouse button in different locations in the output window.
3. Can you write a subroutine with a loop that waits until either the mouse button or a key on the keyboard is pressed? You should exit from the subroutine if either event occurs. Have your main program print something after the call to the subroutine so that you can tell when control returned from the subroutine to the main program.
4. The loop

```
DO
   PLOT MOUSEH, MOUSEV;
LOOP
```

   keeps drawing all the time. Can you change it to make it draw only while the mouse button is being held down? Don't forget to tell the PLOT command to end the line when the mouse button is released.

Chapter 18
Making Music

Commands:
- SOUND, STOPSOUND

Functions:
- TONES, SOUNDOVER~

The Macintosh has three different sound synthesizers: a square-wave synthesizer, which generates single-voice sound; a four-tone synthesizer, which generates four different sounds simultaneously; and a free-form synthesizer, which can make more complex sound effects. Macintosh BASIC contains commands designed to help you use the square-wave synthesizer. You can use the four-voice and free-form synthesizers only if you supplement BASIC with routines written in assembly language using the Apple 68000 Development System.

USING SINGLE-VOICE SOUND

■ SOUND

The simplest sound is a single beep. You can make a beep by using the simple statement SOUND.

```
SOUND  ! Beeps
```

To make a sound more musical than a beep, follow the word SOUND with a group of three numbers called a *triplet.* The three numbers define the sound's frequency, volume, and duration. You can use any type of numeric variable or expression in the triplets, but each must have a value in the range 0 to 32767 so BASIC can convert the number to an integer. The numbers are separated from each other by commas.

```
SOUND 100,200,120  ! Low, loud note 2 seconds long
SOUND 4000,50,30   ! High, soft note half a second long
```

The first number is the frequency of the desired sound, measured in cycles per second, or Hertz. A low frequency produces a low tone and a high frequency, a high tone. You can supply your own frequency number if you wish, but it is usually more convenient to use the TONES function described in the next section.

The second number is the volume. Volume is measured on an arbitrary scale ranging from 0 (silence) to 255 (loudest). If you supply a number outside that range, BASIC brings your number inside the proper range by performing a MOD 256 operation on it. A value of 255 for volume produces the loudest sound consistent with the volume setting in the Control Panel desk accessory.

The last of the three numbers is the duration of the sound in 60ths of a second. A duration number of 60 produces a sound one second long, and the maximum duration number of 32767 produces a sound lasting a little over 9 minutes. You can specify a period of silence between sounds by giving the duration of the silence as the third number and using zero (no volume) for the second number.

You can specify more than one sound in the same SOUND statement. In order to do so, give a complete triplet (all three numbers) for each sound you want, and separate the triplets from each other with semicolons. For example,

```
SOUND 100,200,10; 1100,200,10 ! Plays two short notes
SOUND 1200,200,10; 1,0,30; 1200,200,10
    ! Two notes separated by half a second of silence
```

GETTING THE RIGHT NOTES

■ TONES

TONES is a function that returns the frequency corresponding to a note in the musical scale. TONES uses the chromatic scale, which is the scale corresponding to a piano keyboard. TONES(0) returns the frequency for Middle C.

Each increase or decrease in the argument of TONES represents a half step, or one key (counting both white and black keys) on the piano keyboard. For instance, TONES(−1) returns the frequency for B below Middle C, and TONES(1) returns the frequency of C#, a half step above Middle C. There are twelve keys in an octave (seven white keys and five black keys), so an increase of 12 in the argument of TONES returns the frequency of the note an octave higher than the previous note. The TONES function is frequently used as part of a SOUND statement.

```
freq = TONES(0) ! Puts frequency of Middle C in freq
SOUND TONES(0),200,60 ! Plays Middle C for 1 second
SOUND TONES(-1),200,60 ! Plays B below Middle C
SOUND TONES(1),200,60 ! Plays C# just above Middle C
SOUND TONES(12),200,60 ! Plays C above Middle C
```

TONES returns valid frequencies for arguments ranging from −36 to +48. Thus, TONES covers a range of seven octaves, three below Middle C and four above it. Table 18-1 shows the notes whose frequencies are returned by TONES for each argument within this range.

Figure 18-1 shows notes in the treble clef labeled with their TONES numbers. If you already have sheet music for the song you want to enter, you can locate the correct TONES number for each note by comparing the music with the figure.

Table 18-1. Tones

Note	Tones							
C	−36	−24	−12	0	12	24	36	48
C#	−35	−23	−11	1	13	25	37	
D	−34	−22	−10	2	14	26	38	
D#	−33	−21	−9	3	15	27	39	
E	−32	−20	−8	4	16	28	40	
F	−31	−19	−7	5	17	29	41	
F#	−30	−18	−6	6	18	30	42	
G	−29	−17	−5	7	19	31	43	
G#	−28	−16	−4	8	20	32	44	
A	−27	−15	−3	9	21	33	45	
A#	−26	−14	−2	10	22	34	46	
B	−25	−13	−1	11	23	35	47	

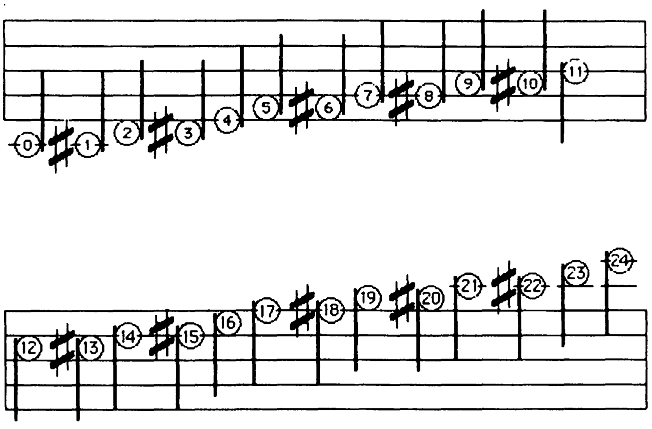


Figure 18-1. Notes on the treble clef with TONES numbers

TAKING SOUND VALUES FROM AN ARRAY

You can tell BASIC to take the sound triplets directly from an integer array. This is especially convenient when you want to play a long tune or want to play the same tune more than once. When you have the triplet values in an integer array, you follow the keyword SOUND with the number of tones, a space, and a pointer to the element of the array that contains the beginning of the first triplet.

```
SOUND 10 @trips%(0)  ! Play 10 notes
```

You must separate the two items by a space, not a comma. If you use a comma, you will receive an error message. The @ sign tells BASIC to pass a pointer to the address in memory that is occupied by the array element you specify. The array must be an integer array (with the % sign as the last character of its name) and should have only one dimension, so the triplet values are stored in adjacent memory locations.

You will usually get the values to put in the array from DATA statements in your program or from a disk file. Remember that the first of the three triplet values is the frequency, not the note numbers used by TONES. If your data has note numbers, you need to use TONES to get the frequency as you put the data in the array. The @ pointer does not have to point to the first element of the array, as long as it points to the beginning of the first triplet to be played. Here is a short example that puts triplets from a DATA statement into an array and then plays the tune:

```
DIM array%(12)
DATA 1484,200,20,1177,200,20
DATA 990,200,20,1484,190,40
FOR count = 1 TO 12
    READ array%(count)
NEXT count
SOUND 4 @array%(1)
END PROGRAM
```

THE SOUND BUFFER

Macintosh BASIC maintains a single area in memory to hold the data that describes sounds waiting to be generated by the sound synthesizer. This memory area is called the *sound buffer.* It operates just like a line at the supermarket or the post office. The first one in line is the first one out. Technically, this is called a "first-in first-out" buffer.

When BASIC executes a SOUND statement in your program, it adds the sounds requested by your statement to the list in the sound buffer. Then it starts the sound synthesizer if the synthesizer is not already playing. Once these two tasks are done, BASIC starts executing the next statement in your program. Thus, your program can be doing other things while the music is playing. The sound keeps playing until it is finished, or you close your program's output window.

If the sound synthesizer is already in use when BASIC executes a SOUND statement in your program, the sounds requested by your statement are played after all the other sounds that were already in the buffer. Once a sound description is added to the buffer, BASIC does not know or care what program sent the sound data. If you have two BASIC programs trying to generate sound at the same time, sounds will be placed into the sound buffer from each of the two programs. Because of this, the sounds requested by one program's SOUND statement may occur between sounds requested by SOUND statements in the other program.

STOPPING SOUND EARLY

■ STOPSOUND

STOPSOUND stops all sound generation instantly. It turns off the sound synthesizer and empties the sound buffer. If several BASIC programs are running simultaneously, STOPSOUND stops the sound from all of them. Sound starts again when one of the programs executes a SOUND statement.

```
STOPSOUND  ! Stops the sound
```

```
SUB WaitSoundOver
DO
   IF SOUNDOVER~ THEN EXIT DO
LOOP
END SUB
```

Figure 18-2. WaitSoundOver subroutine

DETECTING SILENCE

■ SOUNDOVER~

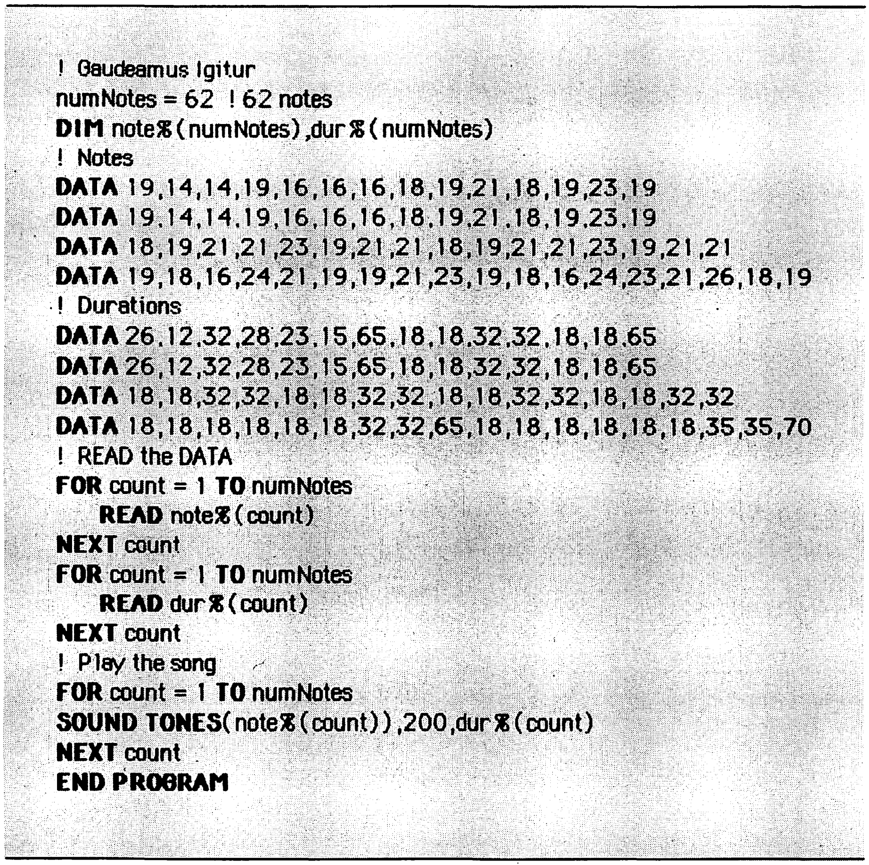
SOUNDOVER~ is a Boolean system function that takes no arguments. It returns true when no sound is being generated and false if sound is being generated. When SOUNDOVER~ returns true, the SOUND buffer is empty.

```
IF SOUNDOVER~ THEN PRINT "Sound is done."
IF NOT SOUNDOVER~ THEN PRINT "Still playing."
```

Sometimes you may want your program to wait until the sound is finished. You will need to do this if you use a musical introduction to your program or if you are making graphics move with the sound. If your program might be running simultaneously with another program that generates sound, you might want to make certain the other program's sound has finished before this one starts. The subroutine WaitSoundOver in Figure 18-2 can be used for this purpose.

EXAMPLE PROGRAMS

The program in Figure 18-3 is a fairly simple example of a program that converts note numbers to frequencies in the SOUND statement itself. The program plays one verse of a traditional song, "Gaudeamus Igitur."



```
! Gaudeamus Igitur
numNotes = 62  ! 62 notes
DIM note%(numNotes),dur%(numNotes)
! Notes
DATA 19,14,14,19,16,16,16,18,19,21,18,19,23,19
DATA 19,14,14,19,16,16,16,18,19,21,18,19,23,19
DATA 18,19,21,21,23,19,21,21,18,19,21,21,23,19,21,21
DATA 19,18,16,24,21,19,19,21,23,19,18,16,24,23,21,26,18,19
! Durations
DATA 26,12,32,28,23,15,65,18,18,32,32,18,18,65
DATA 26,12,32,28,23,15,65,18,18,32,32,18,18,65
DATA 18,18,32,32,18,18,32,32,18,18,32,32,18,18,32,32
DATA 18,18,18,18,18,18,32,32,65,18,18,18,18,18,18,35,35,70
! READ the DATA
FOR count = 1 TO numNotes
    READ note%(count)
NEXT count
FOR count = 1 TO numNotes
    READ dur%(count)
NEXT count
! Play the song
FOR count = 1 TO numNotes
SOUND TONES(note%(count)),200,dur%(count)
NEXT count
END PROGRAM
```

Figure 18-3. Gaudeamus Igitur

The first program statement sets the variable *numNotes* equal to 62, the number of notes in the DATA statements. The number of notes does not change during this program, so it does not have to be in a variable. It is put there, though, as a programming convenience, so if you change the number of notes in the DATA statements, you only have to change the number once instead of five times in the program.

The first four DATA statements in this program hold the note numbers for the song, and the second four DATA statements hold the durations of the notes. This program uses the same volume setting for all the notes, so it does not need to keep volume settings in DATA statements or an array. After the DATA statements, the program has two FOR/NEXT loops that read the DATA items and store them in arrays. The first loop puts the note numbers in an array named *note%,* and the second loop puts the duration numbers in an array named *dur%.*

Repeated execution of the SOUND statement in the last FOR/NEXT loop plays the song. The SOUND statement takes the note number for each note from the *note%* array and uses the TONES function to convert it to a frequency. The statement uses a volume setting of 200 (fairly loud) for all notes and takes the duration of each note from the *dur%* array.

The Mini-piano program in Figure 18-4 provides a way to get note and duration numbers for a tune. It simulates a piano keyboard with Middle C located at the G key and C an octave higher at the RETURN key. Holding the SHIFT key down while you press a letter key makes the note an octave higher. Each time you press a key, the program looks at TICKCOUNT and records the time since the previous keypress. You will need to refine the resulting duration numbers later by hand to allow for rests between notes.

```
! Mini-piano
! Record Tones and Durations
DIM trans(255),t%(200),length%(200)
PRINT "MINI-PIANO"
PRINT "Remembers the tune you play "
PRINT "Press BACKSPACE to end your tune."
! 'g' on the keyboard is Middle C
DATA a,w,s,e,d,r,f,g,y,h,u,j,k,o,l,p,;,[,"'"
DATA A,W,S,E,D,R,F,G,Y,H,U,J,K,O,L,P,:,{,'"'
! Put data in translation array
FOR i = 1 TO 19
READ temp$
```

Figure 18-4. Mini-piano

```
trans(ASC(temp$)) = i+4  ! First octave
NEXT i
FOR i = 1 TO 19
READ temp$
trans(ASC(temp$)) = i + 16  ! SHIFT key octave
NEXT i
trans(13) = 24  ! Data for RETURN key
n = 1  ! Start with note number one
! Save and play notes until BACKSPACE key is pressed
DO
    a$ = INKEY$  ! Check keys
    IF a$ <> "" THEN  ! Test if key was pressed
        tc = TICKCOUNT  ! Get the time
        lth = tc - tcOld  ! Subtract previous time
        IF lth < 0 THEN lth = lth + 32768  ! Wrap-around
        length%(n-1) = lth  ! Save length of note
        tcOld = tc  ! Save the time
        IF a$ = CHR$(8) THEN EXIT DO  ! Exit if BACKSPACE
        note = trans(ASC(a$))  ! Translate key to note
        t%(n) = note  ! Store the note
        SOUND TONES(note),200,10  ! Play it
        n = n + 1  ! Set for next note
        ENDIF
LOOP
! Replay the tune
FOR i = 1 TO n-1
    SOUND TONES(t%(i)),200,length%(i)
NEXT i
! Save the triplets to a disk file
OPEN #1: "Song", APPEND
PRINT #1: n-1  ! Number of triplets
FOR i = 1 TO n-1
    PRINT #1: TONES(t%(i))  ! Frequency
    PRINT #1: 200  ! Volume
    PRINT #1: length%(i)  ! Duration
NEXT i
CLOSE
PRINT "Done"
END PROGRAM
```

Figure 18-4. Mini-piano (*continued*)

The program keeps recording notes until you press the BACKSPACE key. At that time it replays the stored tune and then writes the sound triplets for the tune to a sequential disk file named "Song." Musicians will find many ways to improve this program. It does, however, provide a slightly better way than trial and error to find the right notes.

The first statement dimensions three arrays. The array *trans* is used to store note numbers that correspond to the ASCII value resulting from a keypress, so it is dimensioned to cover the ASCII range, 0 to 255. The arrays *t%* and *length%* are used to record the tones and lengths of your notes. You can dimension the arrays to hold more than 200 notes if you wish. Three PRINT statements display brief instructions in the output window.

The first DATA statement contains the characters of the Macintosh keys that represent piano keys in ascending order. The single quotation mark is enclosed in double quotation marks so BASIC will treat it as data instead of the beginning of a string. The second DATA statement contains the characters of the same keys with the SHIFT key held down.

Two FOR/NEXT loops read each DATA item, convert the character to its ASCII value, and use the ASCII value as an index to the location in the *trans* array to store the number of the note the character represents. The note number 24 for the RETURN key, ASCII 13, is stored in a separate statement because the RETURN key cannot be typed as a character in a DATA statement.

The recording portion of the program consists of a DO loop that continually checks the INKEY$ function. When a key is pressed, the program records TICKCOUNT, calculates the number of ticks that have passed since the previous key was pressed, and stores the result at the previous note's location in the *length%* array. When you press the first key, the length value is stored in element 0 of the array, which is never used again. Then the tick count is saved in *tcOld* so it can be used in the next time calculation.

If the BACKSPACE key was pressed, the program exits from the DO loop. If another key was pressed, the program obtains the corresponding note by using the ASCII value of the keypress character as an index into the array *trans*. The rest of the loop then stores the note number in the *t%* array, plays the note for a short time, and increases the index *n* for the next note.

After completion of the main recording loop, the program uses a SOUND statement in a FOR/NEXT loop to replay the tune using the duration numbers it calculated and then stores the data in a disk file. Note that the program uses TONES to convert the note numbers to frequencies as it writes them to the disk file, so the numbers in the file are triplets ready to be read and played.

PRACTICE EXERCISES

1. Sometimes only numeric digits are acceptable when your program is getting input from the keyboard. Instead of waiting until the entire entry is typed to check for improper characters, it is more informative to beep as soon as the improper key is pressed. Can you write a WHEN KBD loop that beeps whenever a character other than the digits 0 through 9 is entered?
2. What statement would you write to play the note G above Middle C at half volume for half a second?
3. If sound triplets are stored beginning with element 1 of an array named *trips%,* what statement would you write to play nine notes starting with the second note stored in the array?
4. What combination of statements would you use at the beginning of your program to stop any sound that might already be playing and to guarantee that there is at least one second of silence before your music begins?

Part four

The Macintosh Toolbox

Chapter 19

Using the Macintosh Toolbox

Command Identifier:
- TOOLBOX

Function Identifier:
- TOOL

Functions:
- VALPOINTER, HIGHWORD, LOWWORD
- INDIRECT], ADDRESS]

Much of the power and flexibility of Macintosh software comes from the routines contained in the Macintosh's read-only memory (ROM). These routines are called the Macintosh toolbox. This chapter describes how to use the Macintosh toolbox routines from BASIC. It provides a general background and then describes the methods for calling and passing arguments to the toolbox routines.

Apple Computer's Macintosh BASIC manual does not describe how to access the toolbox routines, and Apple does not provide any official support for the use of these routines from BASIC. You should treat these routines as an undocumented feature and use them with caution.

THE TOOLBOX

The Macintosh contains over 400 subroutines and functions in its read-only memory. These routines provide the foundation for all the major programs that are written for the machine, including BASIC and other high-level languages. The ROM routines include QuickDraw graphics (the high-speed calculations and display of graphics on the screen), the low-level portions of the Macintosh operating system, and utilities to create and manipulate windows, menus, controls, and other aspects of the Macintosh user interface. While only this last group of utilities is properly called the Macintosh toolbox, the term "toolbox" has come to be applied to all the routines in the machine's read-only memory.

The descriptions of toolbox routines in this book are only a brief introduction to some of the most useful routines in the toolbox. Appendix D contains a complete list of the toolbox routines recognized by BASIC. If you want to explore the toolbox in greater detail, you should purchase a copy of Apple's book, *Inside Macintosh* (Apple Computer Corporation, Cupertino, Calif.: 1985).

TOOLBOX SUBROUTINES AND FUNCTIONS

■ TOOLBOX, TOOL

Apple Computer has given Macintosh BASIC the ability to recognize the names of the most useful toolbox routines. To use one of these toolbox routines, you must precede the name of the routine with a special keyword and use the correct arguments. You use the word TOOLBOX in front of subroutine names and the word TOOL in front of function names.

In the following listing, the first line calls the toolbox routine DrawMenuBar, which draws the menu bar across the top of the screen. The second line calls the toolbox function StringWidth, which returns the width of a string in screen pixels.

```
ToolBox DrawMenuBar   ! Call to toolbox subroutine
width = Tool StringWidth (string$)   ! Call to toolbox function
```

At this point you do not need to be concerned about what the individual toolbox routines do, only about the way you call them.

LEAVING BASIC'S PROTECTED ENVIRONMENT

When you use the toolbox routines, you leave BASIC's "safe" programming environment and enter a harsher environment where an error is usually fatal. If you like the safety of BASIC's error checking, you may not want to venture into the toolbox at all. But if you enjoy solving puzzles and experimenting to see what works, you may enjoy exploring the toolbox.

Most of the individual toolbox routines are quite simple. The difficulty in using them is that there are so many of them. You have to learn how to use them together—in what order and in what combinations. In addition, you have to learn which things are taken care of by BASIC and which things you need to do yourself.

Your BASIC program is not in complete control of the machine. Your program operates in an environment that is created and maintained by Macintosh BASIC. When you use toolbox routines, you need to keep in mind that BASIC is also using the toolbox to maintain its programming environment. The next few chapters of this book will describe some of the things you need to know to keep your toolbox calls from interfering with BASIC.

One of the reasons the toolbox routines are so fast is that they do not spend any time checking for errors. If you make an error using a toolbox routine, your program will fail or crash. Instead of an explanatory error message written in English, you will see a dialog box with a picture of a bomb and an error number. The system error numbers and explanations are listed in Appendix B, but the error number may not help you find the problem. If you venture out of BASIC's protected environment into the toolbox, the dialog box with the picture of a bomb will become familiar.

An excellent habit when using the toolbox is to save the text of your program on disk and then eject the disk every time you make any changes. If you save your program before you try to execute it, you will have a copy on disk if a fatal error occurs. Be prepared for a crash every time.

BASIC does only a little bit of checking on your calls to toolbox routines. It checks for the right number of bytes in the arguments you are passing to the toolbox routine. This is an important check and will help you catch many errors. However, BASIC does not check the content of the arguments you are passing. You are responsible for that yourself.

PARAMETER PASSING

If all you do is use a few of the examples in the next few chapters exactly as they are written, you may not need to learn the details of variable types and parameter passing. However, if you want to make extensive use of toolbox routines, you will need to understand the different types of variables you can use and how to pass them as parameters.

Toolbox Variable Types

The routines in the Macintosh toolbox were not designed to be called from BASIC. They were designed to be called from another high-level language, Pascal. As a consequence, the toolbox uses several variable types that are different from the variable types you use in BASIC.

When you pass arguments from BASIC to a toolbox routine, you need to make your arguments look like the kinds of variables the toolbox routine expects. When you receive a block of data from a toolbox routine, you may have to manipulate the data to turn it into a format that matches a BASIC variable type. The methods you use to do this are similar to the methods used in Chapter 13 to call GETFILEINFO, SETFILEINFO, and GETVOLINFO.

Toolbox Data Types

Table 19-1 lists the data types most often used in calls to toolbox routines. Integer, pointer, handle, character, and Boolean data types are essentially the same for BASIC and the toolbox routines. A *pointer* is a 4-byte value that points to the address of a variable or an array element. A *handle* is a 4-byte value that points to the address of a pointer (a pointer to a pointer, if you want to think of it that way). You can think of pointers and handles as performing a similar function. However, they are separate data types and cannot be mixed.

You receive a pointer or handle when you call a toolbox function that allocates storage space in memory for data related to a new

Table 19-1. Common Toolbox Data Types

Data Type	Bytes	Description
integer	2	integer
ptr	4	pointer
handle	4	handle
char	1	character
Boolean	1	Boolean
str255	256	255-character string
longint	4	long integer
point	4	two integers
rect	8	four integers
packed array [1..4] of char	4	four characters, one per byte

object, such as a new window or menu. You save the pointer or handle in a BASIC variable and then use it as an argument for other toolbox routines that require a pointer or handle.

The toolbox data type *str255* is a string of up to 255 characters. When this string is stored in memory, the first byte of the storage area gives the number of characters in the string.

The data type *longint* is an integer, called a long integer, which is twice as large as a normal BASIC integer. The long integer occupies 4 bytes, while a normal integer occupies 2 bytes.

The data type *point* used by toolbox routines is really a combination of two integers. The first integer is the vertical coordinate of a point and the second, the horizontal coordinate. The data type *rectangle* consists of two points (the top left and bottom right corners of a rectangle), so it is really a group of four integers.

The last of the common toolbox data types, a packed array [1..4] of char, is a sequence of four characters. It does not have a length byte at the beginning, so it is not interchangeable with the string data type. The packed array of characters is used for file types and creators and for resource type identifiers, which will be introduced in Chapter 23.

Passing Parameters by Value and by Reference

A toolbox routine will respond correctly as long as it receives the correct number of bytes and the information in those bytes fits the type of data the routine needs. The toolbox routine does not know or care what type of variable you use to generate the information.

The normal way to pass arguments is by passing their actual values. However, if the size of a parameter is more than 4 bytes, the toolbox expects to receive a pointer to an address in memory where the values are located. Passing a pointer instead of the value itself is sometimes called passing a parameter by reference.

You use the @ sign before the name of an array element to pass a pointer to that element. For example, in the statement ToolBox GetWTitle(w],@array(0)), the sequence @array0) tells BASIC to pass a pointer to the address in memory occupied by element 0 of the array named array. Usually it is easiest to put information into an array if the array consists of either integer or character variables.

You must also pass a pointer to a variable or array element when the toolbox routine changes the value of the argument you supply. In normal toolbox notation, this type of parameter is preceded by the letters VAR. The arguments in the list of toolbox routines in Appendix D use the @ notation when you need to pass a pointer instead of a value.

Table 19-2 summarizes how each of the common data types is passed as a parameter. The first column lists the names of the most common toolbox data types. The second column lists the length of each type in bytes, and the third column shows the corresponding data type in BASIC.

The column titled "By Value" illustrates how to pass a parameter of a particular data type by value, and the column titled "By Reference" illustrates the way to pass that data type by reference. The column titled "Array Size" shows an array dimensioned to the number of bytes required to hold a particular data type.

The integer, pointer, and handle data types are straightforward. To pass a value, you supply any expression of the appropriate type. If you supply any type of numeric expression, BASIC converts the number to an integer for the toolbox call. If the toolbox routine returns a value, you must supply a pointer to either a simple variable or an array element.

The character and Boolean data types are each one byte long. Toolbox routines discard the extra byte when they receive a character or Boolean value. When a toolbox routine expects a character value, you can use a character variable or any numeric expression, as long as the value is an integer in the range 0 to 255. When a toolbox routine expects a Boolean value, you can use any Boolean expression.

Strings are an exception to the 4-byte rule. You can use them directly as arguments unless the string is to be changed by the toolbox routine. Toolbox routines accept strings from BASIC, but you must take special care in BASIC to allocate a full 256 bytes whenever you are receiving a string back from the toolbox. You can do this by dimensioning a 256-element character array or a 128-element integer array (remember, integers are two bytes each).

Rectangle data types, since they occupy eight bytes, are always passed by reference. To do this, you need to dimension a four-element integer array and store the rectangle's coordinates in the array in the order top, left, bottom, right.

Table 19-2. Toolbox Routine Parameter Passing

Toolbox Type	Bytes	BASIC Type	By Value	By Reference	Array Size
integer	2	integer	a%	@a%	
ptr	4	pointer	a]	@a]	
handle	4	handle	a}	@a}	
char	1*	character	a©	@a©	
Boolean	1*	Boolean	a~	@a~	
str255	256	character array	a$	@str©(0)	dim str©(255)
longint	4	two integers	lowHalf%,hiHalf%	@num%(0)	dim num%(1)
point	4	two integers	h%,v%	@point%(0)	dim point%(1)
rect	8	integer array	@rect%(0)	@rect%(0)	dim rect%(3)
packed array [1..4] of char	4	two integers	last2%,first2%	@type©(0)	dim type©(3)

*The char and Boolean types each occupy 1 byte, but a minimum of 2 bytes is always passed to the toolbox routines. The extra byte is ignored.

Data types that occupy four bytes get special treatment when you pass them by value. You must pass them as two BASIC integers, with the second integer first. For example, a point data type occupies four bytes. The first two bytes are the point's vertical coordinate and the second two bytes are the point's horizontal coordinate. When you pass a point by value, you pass the last two bytes with the horizontal coordinate first, as in this example:

```
! Pass a point by value and by reference
DIM pt%(1)    ! 2 integers
ToolBox GlobalToLocal (@pt%(0))   ! by reference
! pt%(0) contains the vertical coordinate
! pt%(1) contains the horizontal coordinate
ToolBox MoveTo (pt%(1),pt%(0))   ! by value
```

This reversed ordering occurs only when passing a 4-byte type by value. The four bytes occur in their normal order when you pass by reference.

Very few toolbox routines require long integers. For those that do use long integers, however, you can use a sequence like the following to pass a long integer by value:

```
! Pass a longint
! Assume a\ contains a positive longint value
hi# = TRUNC(a\ / 65536)
lo# = a\ - hi# * 65536
! Adjust a very large number for different
!  storage format in BASIC and toolbox.
! Adjustment not necessary for numbers under 32767.
    IF hi# > 32767 THEN hi# = hi# - 65536
    IF lo# > 32767 THEN lo# = lo# - 65536
! BASIC converts lo# and hi# to short integers before it passes them.
Toolbox Secs2Date( lo#,hi#,@daterec%(0))
```

With the 4-character data type, you first pass an integer containing the last two characters and then an integer containing the first two characters. You build the integers from the ASCII values of the characters, as in the following example.

```
! Pack 4 chars into 2 integers
! The value that gets packed is 'FONT'
ResTypeFirst2% = 256 * ASC ('F') + ASC ('O')
ResTypeLast2% = 256 * ASC ('N') + ASC ('T')
! Last2, then First2
ToolBox AddResMenu (Menu},ResTypeLast2% ,ResTypeFirst2% )
```

Sometimes you need to pass a handle or pointer with a value of nil to a toolbox routine. A nil pointer or handle is simply two words containing all zeros, so you can pass a nil pointer by passing 0,0. Some routines take a special action when they receive the value pointer(−1). You can pass the value of pointer(−1) by passing the two integers −1, −1.

CONVERTING FROM ONE TYPE TO ANOTHER

Macintosh BASIC provides five functions that you can use to convert from one variable type to another. The VALPOINTER, HIGHWORD, AND LOWWORD functions help you decode results returned by toolbox functions. The INDIRECT] and ADDRESS] functions allow you to manipulate pointers and handles.

Treating Pointers as Numbers

- VALPOINTER, HIGHWORD, LOWWORD

Macintosh BASIC does not have a 4-byte long integer data type to match the toolbox longint data type. So when a toolbox function returns a 4-byte long integer, Macintosh BASIC stores it in a different 4-byte variable, a pointer. Storing the longint in a pointer variable simplifies BASIC's internal operations—but your program then has to read the value in the pointer as a number.

The VALPOINTER function gives you the numeric value of a pointer. It requires one argument, the pointer whose value you want to read as a number. Because the value stored in the pointer was a 4-byte long integer, the value may be too large to store in BASIC's 2-byte integer variables. If you need to store the value in a variable, you should store it in a double-precision or extended-precision real variable or in a computational (8-byte long integer) variable.

```
a] = Tool TickCount
ticks = VALPTR(a])
PRINT "TickCount is "; ticks
```

A few of the toolbox functions pack two integers into a single longint before returning the 4-byte long integer in a pointer variable. After using one of these functions, you usually need to separate the result into the two integers. The HIGHWORD function returns the integer value contained in the high half of the longint and LOWWORD returns the integer value contained in the low half of the longint. Both functions take one argument, the pointer or number that contains the value you want to unpack.

```
p] = Tool PinRect (@rect%(0), h%, v%)
PRINT "h = "; LOWWORD(p]); ", v = "; HIGHWORD(p])
a = HIGHWORD(n) ! same as n DIV 65536
b = LOWWORD(n) ! same as n MOD 65536
```

The results returned by VALPOINTER, HIGHWORD and LOWWORD make sense with a pointer as the argument only if the pointer contains a longint value from a toolbox routine.

Converting Pointers and Handles

■ INDIRECT], ADDRESS]

A pointer points to the address of a variable or an array element, and a handle points to the address of a pointer. You cannot mix handles and pointers in the same expression. However, you *can* obtain the pointer to which a handle points and the address or value to which a pointer points. This ability is very helpful when working with toolbox routines, because some use pointers to refer to data and others use handles to refer to the same data.

The INDIRECT] function requires one argument. If the argument is a handle, INDIRECT] returns the pointer to which the handle points. If the argument is a pointer, INDIRECT] returns the 4 bytes to which the pointer points. If you supply a string variable, INDIRECT] returns a pointer to the string.

```
p] = INDIRECT](h})   ! Handle to pointer
a] = INDIRECT](p])   ! Gets the 4 bytes pointed to
```

The ADDRESS] function also requires one argument—a handle, pointer, string, or number. While the INDIRECT] function returns the value to which its argument points, the ADDRESS] function returns a memory address that you can store in a pointer variable. You can use ADDRESS] in combination with VALPOINTER to perform arithmetic on pointer values and store the results back into a pointer variable as in this example:

```
! Make pointer point 8 bytes higher in memory:
p] = ADDRESS](VALPOINTER(p])+8)     ! Adds 8 to pointer
```

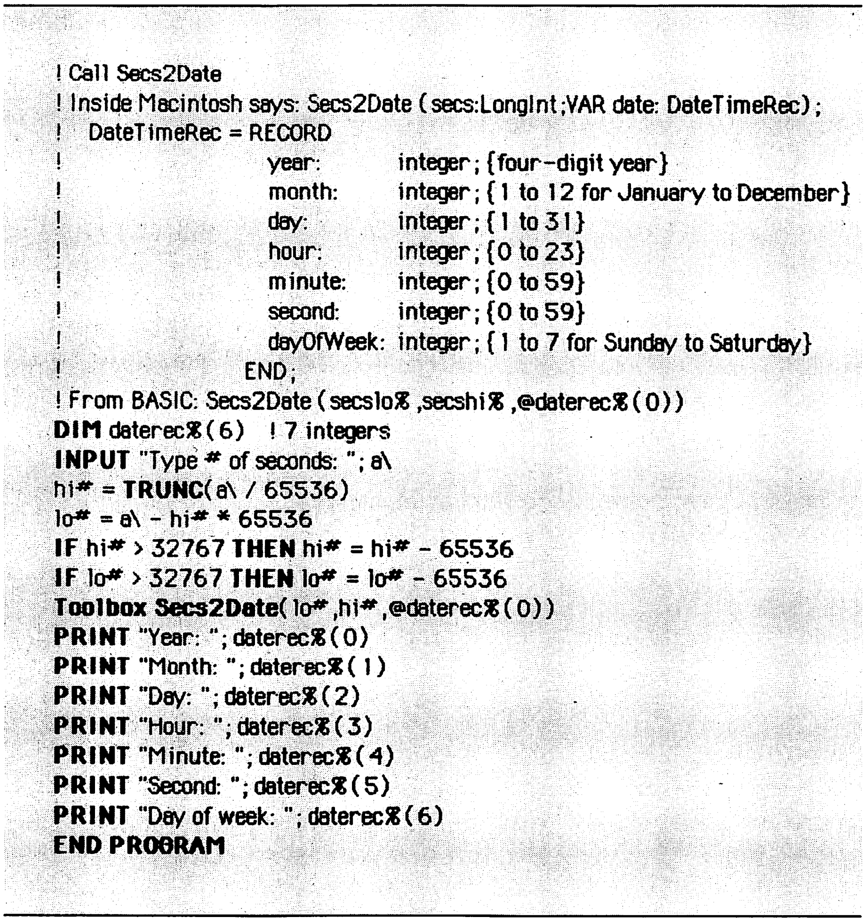
EXAMPLE PROGRAM

The example program in Figure 19-1 calls the toolbox routine Secs2Date. This routine converts the number of seconds since 12:00 A.M., January 1, 1904, into the actual date and time.

According to Apple's *Inside Macintosh,* Secs2Date requires two arguments, a longint and a pointer to a data type called a DateTimeRec. The DateTimeRec is actually an array of seven integers that contain the year, month, day, hour, minute, second, and day of the week. Comments in the program listing describe the record in detail.

From BASIC, you use two integers to simulate the long integer and an integer array to simulate the DateTimeRec. The example program starts by dimensioning a 7-element integer array for the DateTimeRec. The number of seconds to be converted is received from the keyboard and stored in an extended-precision variable (in case it is a very large number).

The program calculates the high half of the number by dividing by 65536 and truncating the result. The low half is the original number less 65536 times the high half. If either half of the number



```
! Call Secs2Date
! Inside Macintosh says: Secs2Date (secs:LongInt;VAR date: DateTimeRec);
!   DateTimeRec = RECORD
!                   year:        integer; {four-digit year}
!                   month:       integer; {1 to 12 for January to December}
!                   day:         integer; {1 to 31}
!                   hour:        integer; {0 to 23}
!                   minute:      integer; {0 to 59}
!                   second:      integer; {0 to 59}
!                   dayOfWeek:   integer; {1 to 7 for Sunday to Saturday}
!                 END;
! From BASIC: Secs2Date (secslo% ,secshi% ,@daterec%(0))
DIM daterec%(6)  ! 7 integers
INPUT "Type # of seconds: "; a\
hi# = TRUNC(a\ / 65536)
lo# = a\ - hi# * 65536
IF hi# > 32767 THEN hi# = hi# - 65536
IF lo# > 32767 THEN lo# = lo# - 65536
Toolbox Secs2Date( lo#,hi#,@daterec%(0))
PRINT "Year: "; daterec%(0)
PRINT "Month: "; daterec%(1)
PRINT "Day: "; daterec%(2)
PRINT "Hour: "; daterec%(3)
PRINT "Minute: "; daterec%(4)
PRINT "Second: "; daterec%(5)
PRINT "Day of week: "; daterec%(6)
END PROGRAM
```

Figure 19-1. Call Secs2Date

is greater than 32767, that half is reduced by 65536, which makes it a negative number. This ensures that each half of the longint will be a short integer in the range +32767 to −32767. The toolbox routine Secs2Date converts the negative numbers back into positive ones.

The argument list for the toolbox routine Secs2Date consists of the low half and then the high half of the longint followed by a pointer to the array that is to receive the answer. The last part of the program merely displays each element of the array along with a label that describes its meaning.

PRACTICE EXERCISES

1. How many bytes in memory do each of the following BASIC arrays occupy?
   a. DIM a%(10)
   b. DIM b©(8)
   c. DIM c(3)
2. How many bytes in memory do each of the following toolbox data types occupy?
   a. integer
   b. point
   c. str255
   d. rect
3. How large should you dimension an integer array to receive a rect value from a toolbox routine? To receive a str255 value? How large should you dimension a character array to receive a str255 value?

Chapter 20
Windows and Menus

Commands:
- WHEN WINDOW, WHEN MENU

System Functions:
- OUTPUTWINDOW], MENU}, MENUID MENUITEM

This chapter describes how to use toolbox routines to create and manipulate windows and menus. In addition to describing the toolbox routines, the chapter also discusses how to keep your toolbox calls from interfering with the windows and menus that BASIC maintains.

WORKING WITH WINDOWS

When you use BASIC's standard output window, BASIC takes care of all the details involved in maintaining the window. If you make your own window, you have to watch out for some of the details yourself. For example, if your window has a size box, you need to use the toolbox to change the window's size when the size box is dragged. In addition, you are responsible for closing the extra window when your program is finished with it.

Finding the Right Window

■ OUTPUTWINDOW]

When you call a routine that manipulates a window, you need to tell the routine which window to manipulate. You do this by including a window pointer as one of your arguments. If you create the window yourself, you will already have a pointer to the window. Usually, however, you can use one of the two standard window pointers returned by OUTPUTWINDOW] and Front-Window.

BASIC's system function OUTPUTWINDOW] returns a pointer to the current program's output window. The toolbox function FrontWindow returns a pointer to the frontmost, or active, window. FrontWindow is less useful than OUTPUTWINDOW]. Since BASIC allows you to run several programs at once, the front window may belong to another program. It could even be your program's text window if you clicked on it while the program was running. Using BASIC's OUTPUTWINDOW] function ensures that your window operations are being performed on the output window.

```
w] = OutputWindow]      ! Pointer to output window
w] = Tool FrontWindow   ! Pointer to frontmost window
```

Making Your Own Windows

You can use the toolbox function NewWindow to create additional windows. Remember, though, that operating your own windows can become complicated. It may just be simpler to use your program's standard output window. The next example shows how to call NewWindow.

```
DIM rect%(3)
rect% (0) = top%
rect% (1) = left%
rect% (2) = bottom%
rect% (3) = right%
W]=Tool NewWindow (0,0,@rect%(0),title$,vis~,procID%,-1,-1,goAway~,0,0)
```

The first two zeros in the argument list tell NewWindow to find its own storage space in memory. Do not try to allocate storage space yourself from BASIC. The next argument is a pointer to an array in your program that contains the four coordinates of the rectangle in which the window is to appear. You store the coordinates in the array in the order top, left, bottom, right. The next argument is the title of the window. It is followed by a Boolean expression that tells NewWindow whether the window will be visible. This Boolean expression should always be true unless you really have use for a window nobody can see.

NewWindow's sixth parameter is a procedure ID that tells NewWindow what kind of window you want. Figure 20-1 shows the six predefined window types. Type 0 is a regular document

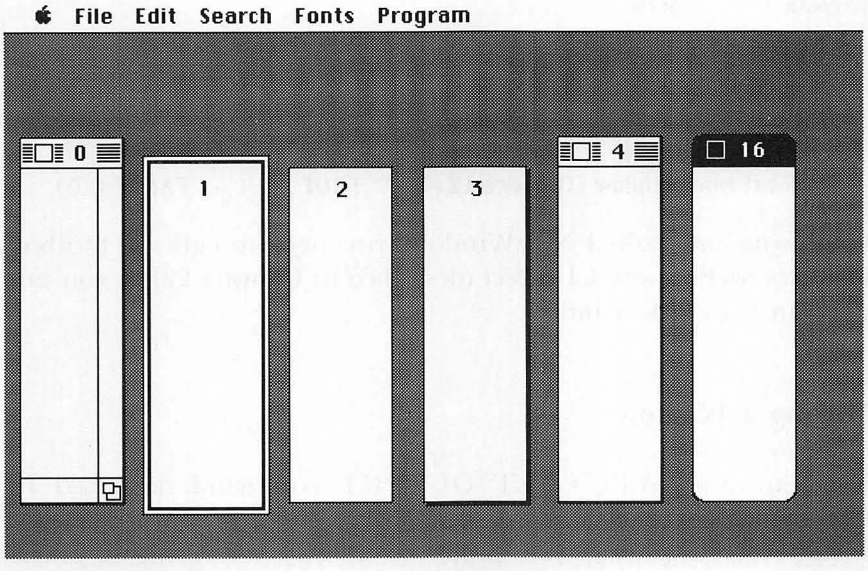


Figure 20-1. Types of windows

window. It has a close box and may have a size box if you call the toolbox routine DrawGrowIcon to draw the size box. Type 4 is a document window without a size box. Type 1 is the double-bordered window used for alerts and some dialog boxes. Type 2 is a simple, plain box, and type 3 is the same box with a little shadow. Type 16 is the rounded-corner window used for the Calculator and Puzzle desk accessories.

After the window procedure ID, you use two integers to tell NewWindow whether the new window is to be in front of or behind other windows. Values of −1,−1 cause your window to be in front, and values of 0,0 cause it to be behind all others. Normally, you should use −1,−1 to make the newest window appear in front.

The next Boolean expression tells NewWindow whether the window will have a close box. It is usually true for types 0, 4, and 16 and false for the other three types. The last two arguments are normally zeros. Here are some more examples using NewWindow:

```
DIM rect%(3)
rect% (0) = top%
rect% (1) = left%
rect% (2) = bottom%
rect% (3) = right%
! Normal document window
W]=Tool NewWindow (0,0,@rect%(0),title$,TRUE,0,-1,-1,TRUE,0,0)
! Rounded corners window
W]=Tool NewWindow (0,0,@rect%(0),title$,TRUE,16,-1,-1,TRUE,0,0)
! Alert dialog window
W]=Tool NewWindow (0,0,@rect%(0),"",TRUE,1,-1,-1,FALSE,0,0)
```

Once you have called NewWindow, you need to call the toolbox routines SetPort and ClipRect (described in Chapter 22) so you can draw in your new window.

Moving a Window

You can use BASIC's SET OUTPUT command described in Chapter 16 to move or change the size of your program's output window. If, however, you prefer to specify locations in pixels instead of inches, you can use the toolbox routine MoveWindow instead. You also need MoveWindow if you want to move a window that is not your program's output window.

MoveWindow requires four arguments. The first is a pointer to the window you want to move. The second and third arguments are the new horizontal and vertical locations of the window's top left corner. The location is measured in pixels from the top left corner of the screen. For example, the position 8,12 is eight pixels to the right and twelve pixels downward from the top left corner of the screen.

The last argument is a Boolean expression that tells MoveWindow whether to make this window the active window. If the Boolean expression is true and the window is not already active, MoveWindow makes this window the active window. If the window being moved is already the active window, the setting of the fourth argument makes no difference. MoveWindow does not change the window's size or contents.

```
ToolBox MoveWindow (OutputWindow],8,12,TRUE)
ToolBox MoveWindow (w], left%, top%, front~)
```

Changing the Size of a Window

Using BASIC's SET OUTPUT command is an easy way to change the size of the standard output window. However, if you want to move a window that is not your program's output window or you want to specify locations in pixels instead of inches, you can use the toolbox routine SizeWindow.

SizeWindow requires four arguments. After a pointer to the window that you want to change, you supply the new width and height of the window in pixels. The last argument is a Boolean expression that tells SizeWindow whether you want the contents of the changed area to be redrawn. You should almost always use a value of true for this argument.

```
ToolBox SizeWindow (OutputWindow],width%,height%,fUpdate~)
ToolBox SizeWindow (w],200,200,TRUE)
```

If the window you are resizing contains a size box or scroll bars, you need to handle those items separately after calling SizeWindow. Redraw the size box using the toolbox routine DrawGrowIcon, which requires the window pointer as an argument.

The toolbox routines for moving, resizing, and drawing scroll bars and other controls are described in Chapter 21.

```
ToolBox SizeWindow (W],100,200,TRUE)
ToolBox DrawGrowIcon (W])     ! redraw size box
```

Changing a Window's Title

The toolbox contains two routines that allow you to change and obtain the title of a window. To change a window's title, you use the toolbox routine SetWTitle. Its parameters are a pointer to the window that is to display the new title and a string that contains the new title.

```
ToolBox SetWTitle (W], title$)
ToolBox SetWTitle (OutputWindow],'Window Title')
```

To get a copy of the current title of a window, use GetWTitle. In addition to the appropriate window pointer, GetWTitle needs a pointer to an array with 256 bytes into which it can put the string that contains the window's title. After calling GetWTitle, you will need to translate the data in the array into a string that BASIC can understand. This is most easily done from an array of characters.

```
DIM title©(255)  ! 256 bytes with 0th element
ToolBox GetWTitle (OutputWindow],@title©(0))
length = title©(0)
title$ = ""
FOR count = 1 TO length
    title$ = title$ & CHR$(title©(count))
NEXT count
PRINT "The window's title is:"
PRINT title$
```

Closing Your Own Windows

If your program creates its own window, your program is also responsible for closing the window when you are finished with it.

The toolbox routine DisposeWindow closes a window and frees up the memory areas occupied by the data that describes the window. It takes only one argument, a pointer to the window to be closed. Use DisposeWindow only on windows you have opened yourself. If you use it to close one of the windows opened by BASIC, you may cause BASIC to crash.

```
ToolBox DisposeWindow (w]) ! Closes your window
ToolBox DisposeWindow (OutputWindow]) ! BASIC will crash
```

Learning When to Close Windows

- WHEN WINDOW

When you click on the close box of one of BASIC's windows, BASIC closes the window. However, if you click on the close box of a window your program created, your program needs to intercept the request and close the window itself.

You can enable a WHEN WINDOW interrupt in your program to intercept the request to close a window. Use the command WHEN WINDOW followed by a pointer to the window. The statements between the WHEN WINDOW statement and its END WHEN statement will be executed whenever you click on the window's close box.

```
WHEN WINDOW myWindow]
    close~ = TRUE
END WHEN
! At appropriate places in your main program:
IF close~ THEN
    IGNORE WHEN WINDOW myWindow]
    ToolBox DisposeWindow (myWindow])
ENDIF
```

The WHEN WINDOW interrupt routine should usually set a Boolean variable to tell your main program to close the window. The main program needs to know when the window is closed so it will not try to print or draw in a window that does not exist.

You can use WHEN WINDOW OutputWindow] to find out when the output window's close box is clicked. When this happens, you should close your own windows before BASIC terminates the program. If you want to intercept closing requests for more than one window, use a separate WHEN WINDOW structure for each window as in the following example:

```
WHEN WINDOW OutputWindow]
    CALL Quit
END WHEN
WHEN WINDOW myWindow]
    CALL Quit
END WHEN
! At appropriate place in main program:
SUB Quit
    IGNORE WHEN WINDOW myWindow]
    ToolBox DisposeWindow (myWindow])
    END PROGRAM
END SUB
```

WORKING WITH MENUS

There are two ways to identify menus when using toolbox routines. Some of the routines use a menu identification number, and other routines use a handle to a menu (called a *menu handle* for short). Identification numbers are assigned when the menus are first created. If you know a menu's number, you can use that number as an argument to the toolbox function GetMHandle to obtain a menu handle.

```
theMenu} = Tool GetMHandle( 1 )  ! Gets handle to menu ID #1
```

With menus, as with windows, you need to avoid interfering with BASIC's operation. Specifically, you should not tamper with the items in BASIC's menus or try to use any menus of your own with the same menu ID numbers used by BASIC. BASIC's menus are numbered from 1 through 6, starting from the left end of the menu bar.

Removing a Menu

Your program can safely remove an entire menu if you are willing to do without the commands on that menu. You cannot tell BASIC to execute the commands on a menu when the menu is not on the menu bar. To delete a menu from the menu bar, you use the toolbox routine DeleteMenu. DeleteMenu takes only one argument, the ID number of the menu to be deleted. After you have deleted the menu, you call DrawMenuBar, which takes no arguments, to redraw the menu titles. Otherwise, the titles on the menu bar will not truly reflect the menus that are present.

```
ToolBox DeleteMenu(4)   ! Deletes Search menu
ToolBox DeleteMenu(5)   ! Deletes Fonts menu
ToolBox DeleteMenu(6)   ! Deletes Program menu
ToolBox DrawMenuBar     ! Redraw changed menu bar
```

If you wish, you can remove BASIC's Search, Fonts, and Program menus while your program is executing. You should not, however, remove the Apple and Edit menus. Both the Apple and Edit menus must be present for desk accessories like the Note Pad and Scrapbook to work. In addition, these menus are an important part of the Macintosh user interface.

Inserting a Menu

To add a new menu to the menu bar, you use the toolbox routine InsertMenu. The first argument for InsertMenu is a menu handle that specifies what menu to add. The routine takes a second argument, which is an integer that tells InsertMenu where on the menu bar to put the new menu. If the integer is 0, the new menu follows the existing menus. If the integer is the ID number of a menu that is already on the menu bar, the new menu is inserted just before that menu.

```
ToolBox InsertMenu (myMenu},0)  ! Adds at end of menu bar
ToolBox InsertMenu (myMenu},2)  ! Inserts before menu ID #2
ToolBox DrawMenuBar
```

The short program in Figure 20-2 shows one way to have your program delete BASIC's Search, Fonts, and Program menus while it is running. Before the program deletes each menu from the menu bar, it uses the GetMHandle function to get the menu's handle. Then, just before the end of the program, the saved menu handles are used to restore the three menus to the menu bar, leaving BASIC's normal menus in place.

Handling Menu Items

Once a menu is installed, you cannot remove or rearrange the items listed on it. You can, however, make a number of changes to individual items. The toolbox contains routines that allow you to change the item's text, add or remove checkmarks, enable or disable the item, and change the style of the item's text.

The items on each menu are numbered from top to bottom, starting with the number 1. All of the toolbox routines that work

```
! Remove Menus during Program
DIM Menus}(6)
! Save handles and delete menus
FOR count = 4 TO 6
    Menus}(count) = Tool GetMHandle(count)
    ToolBox DeleteMenu(count)
NEXT count
ToolBox DrawMenuBar

! Your program goes here

! Restore menus before program ends
FOR count = 4 TO 6
    ToolBox InsertMenu(Menus}(count),0)
NEXT count
ToolBox DrawMenuBar
END PROGRAM
```

Figure 20-2. Remove menus during program

on individual menu items have the same first two parameters, the menu handle and the item number, on which to operate. The toolbox routines DisableItem and EnableItem require no other arguments. When an item is disabled, its name appears in gray instead of black, and you cannot select it. When you enable the item, its name appears in black and you can select it once again. Programs usually disable an item when selecting it from the menu would be inappropriate.

```
FontsMenu} = Tool GetMHandle(5)
ToolBox DisableItem (FontsMenu},4)! Disables fourth fontsize
ToolBox EnableItem (FontsMenu},4)  ! Enables it again
```

The toolbox routines SetItem and GetItem operate on the text of an item. SetItem's third parameter is a string that becomes the new text of the item. Generally, you should not change the entire text of an item, but sometimes changing a portion of the text (such as switching between Show Clipboard and Hide Clipboard) increases ease of use. GetItem requires a pointer to an array in which to store the current text of the specified item. The array must be dimensioned to hold at least 256 bytes.

```
FontsMenu} = Tool GetMHandle(5)
ToolBox SetItem (FontsMenu},7,'Too Big')
DIM item©(255)  ! 256 bytes with 0th element
ToolBox GetItem (FontsMenu},9,@item©(0))
length = item©(0)
name$ = ""
FOR count = 1 TO length
    name$ = name$ & CHR$(item©(count))
NEXT count
PRINT "The font name is:"
PRINT name$
```

The toolbox routine CheckItem is used both to add and to delete a checkmark in front of a menu item. Its third argument, a Boolean expression, indicates whether the checkmark is to be added (true) or removed (false).

The toolbox routine SetItemStyle takes as its third argument an integer that defines the style to be used for the item. The integer is derived in the same way as the style number for GPRINT. You add together the values for each style characteristic you want to use: 0 for plain text, 1 for boldface, 2 for italics, 4 for underlined text, 8 for outline type, and 16 for shadow type. SetItemStyle's counterpart, GetItemStyle, needs a pointer to an integer variable for its third argument. It puts the item's current style value into the integer variable. SetItemStyle is often used to change the available font sizes to outline type in the Fonts menu.

```
FontsMenu} = Tool GetMHandle(5)
ToolBox CheckItem (FontsMenu},1,TRUE)    ! Checks first size
ToolBox CheckItem (FontsMenu},1,FALSE)   ! Removes the check
ToolBox SetItemStyle (FontsMenu},2,8) ! Outlines 2nd item
ToolBox GetItemStyle (FontsMenu},2,@looks%) ! Gets style
```

Making a New Menu

To make a new menu of your own, first use the toolbox function NewMenu to allocate space for a new, empty menu and to get the menu's handle. NewMenu takes two arguments, a menu ID that you make up and the title you want the menu to have on the menu bar. Your menu ID can be any positive number from 7 to 255 (BASIC is already using the numbers 1 through 6).

```
myMenu} = Tool NewMenu (99,'New Choices')
```

You need to save the handle returned by NewMenu to use when installing the menu. The toolbox routine AppendMenu puts item names into the empty menu. It requires two arguments—the handle of the menu and a string containing the names of items to be added to the end of the menu. You can use AppendMenu to add each item individually, or you can combine the items in a single string and add them all at once. If you have more then one item in the string, use a semicolon to separate the items. You can assign a COMMAND-letter combination to a menu item by following its name with a slash (/) and the letter you are assigning. If you start an item with a left parenthesis, the item is disabled.

```
ToolBox AppendMenu (myMenu},'Item 1')
ToolBox AppendMenu (myMenu},'Item 2;Item 3') ! 2 items
ToolBox AppendMenu (myMenu},'(-')    ! Disabled dotted line
ToolBox AppendMenu (myMenu},'Item 5/W')  ! CMD-w = # 5
ToolBox InsertMenu (myMenu},0)
ToolBox DrawMenuBar
```

Once you have filled the menu with items, you need to finish the installation by inserting the menu into the menu bar and redrawing the menu bar. Later, when your program finishes, you're responsible for removing your menu from the menu bar and disposing of the storage that NewMenu allocated to it. To do this, you use the toolbox routines DeleteMenu and DisposeMenu.

```
ToolBox DeleteMenu (99)
ToolBox DisposeMenu (myMenu})
ToolBox DrawMenuBar
```

Selecting From Your Own Menu

- WHEN MENU, MENU}, MENUID, MENUITEM

When you select an item from a menu, BASIC first checks to see whether the menu from which you made the selection was one of its own. If so, BASIC performs the action requested by your selection. If the selection was not made from one of BASIC's menus, BASIC allows your program to find out what menu and item were selected.

To receive menu selection information from BASIC, your program needs to install an interrupt, using a WHEN MENU statement. The WHEN MENU interrupt works similarly to the WHEN ERR and WHEN KBD interrupts described in Chapter 15. You enable the menu selection interrupt by executing a WHEN MENU statement followed by a handle to the menu for which you want to enable selections. If you want to enable selections for more than one menu, you can use more than one WHEN MENU statement. You can disable an interrupt by using an IGNORE WHEN MENU command followed by the handle of the menu whose interrupt you wish to disable.

```
WHEN MENU menu8}  ! Enable interrupt for menu8}
! Statements here are executed
! when selection is made from menu8}
END WHEN      ! Back to where we came from
IGNORE WHEN MENU menu8} ! Turn off interrupt
```

When an item from the menu specified in the WHEN MENU statement is selected, BASIC executes the program instructions that follow the WHEN MENU statement until it encounters an END WHEN statement; then control returns to the place where the menu selection interrupt occurred.

In your interrupt-handling routine, you can use the system functions MENUID, MENU}, and MENUITEM, MENUID returns the ID number of the menu from which the item was selected, and MENU} returns a handle to that menu. MENUITEM returns the number of the item that was selected. None of these functions requires any arguments. When your program has finished acting on a menu selection, it should call the toolbox routine HiliteMenu with an argument of 0 to turn off the inverting of the menu's title.

```
WHEN MENU myMenu}  ! Enable interrupt for myMenu}
    PRINT "Item "; MENUITEM ; " was selected"
    PRINT "from menu number "; MENUID
    ToolBox GetItem (MENU}, MENUITEM, @title©(0))
    ToolBox HiliteMenu (0)  ! Make menu title normal again
END WHEN      ! Back to where we came from
```

EXAMPLE PROGRAMS

The Window Shrinker program in Figure 20-3 uses BASIC's pointer to the output window and three toolbox routines. First the value of BASIC's OUTPUTWINDOW] function is assigned to the program's own window pointer variable, *w]*. This allows you to use a shorter name for the window pointer in subsequent statements. A call to SetWTitle changes the title of the output window to "Shrinking Window."

The program concludes with a loop that calls MoveWindow and then Size Window. Each time through the loop, the program

```
! Window Shrinker
w] = OutputWindow]
ToolBox SetWTitle (w],'Shrinking Window')
FOR i = 1 TO 240 STEP 2
ToolBox MoveWindow(w],239+i,40+i,TRUE)
ToolBox SizeWindow(w],240-i,240-i,FALSE)
NEXT i
END PROGRAM
```

Figure 20-3. Window shrinker

moves the top left corner of the window two more pixels right and downward and reduces the size of the window by two pixels in each direction. The combined effect is that the window slowly collapses into the bottom right corner until it reaches its smallest possible size, a narrow vertical bar.

The program in Figure 20-4 shows how to structure a program so that it responds to menu selections. The program replaces BASIC's File menu with a File menu of its own, removes the Search and Program menus, and puts up a new menu labeled Choices.

```
! Menu Driven Program
oldFile} = Tool GetMHandle (2)    ! Take down some of BASIC's menus
oldSearch} = Tool GetMHandle (4)
oldProgram} = Tool GetMHandle (6)
ToolBox DeleteMenu (2)
ToolBox DeleteMenu (4)
ToolBox DeleteMenu (6)
myFile} = Tool NewMenu (98,"File")  ! Put up a new File menu
a$ = "Stop/K;( - ;Print Document/P;( - ;Full Window/F;Small Window/S"
ToolBox AppendMenu (myFile}, a$)
ToolBox InsertMenu (myFile},3)
myChoices} = Tool NewMenu (99,"Choices")  ! Put up a second new menu
```

Figure 20-4. Menu-driven program

```
ToolBox AppendMenu (myChoices}, "Tone;Music;Picture")
ToolBox InsertMenu (myChoices},0)
ToolBox DrawMenuBar
WHEN MENU myFile}
    SELECT CASE MENUITEM
        CASE 1:     CALL Reset
                    END MAIN
        CASE 3:     DOCUMENT PRINT
        CASE 5:     SET OUTPUT TOSCREEN
        CASE 6:     SET OUTPUT
    END SELECT
    ToolBox HiliteMenu (0)  ! Turn off inverted title
END WHEN
WHEN MENU myChoices}
    SELECT CASE MENUITEM
        CASE 1 ! Tone
            SOUND TONES(12),200,100
        CASE 2 ! Music
            SOUND TONES(6),200,10; 1,0,2; TONES(6),200,10; 1,0,2
            SOUND TONES(6),200,10; 1,0,2; TONES(2),200,50
        CASE 3 ! Picture
            PAINT RECT 10,10; 485,280
            INVERT OVAL 50,50; 445,240
            ASK PATTERN pat: SET PATTERN pat+1
            PAINT RECT 100,100; 395,190
    END SELECT
    ToolBox HiliteMenu (0)  ! Turn off inverted title
END WHEN
WHEN WINDOW OutputWindow]
    CALL Reset
END WHEN
DO      ! Program waits here for a menu selection
LOOP
END PROGRAM
SUB Reset
ToolBox DeleteMenu (98)
ToolBox DeleteMenu (99)
ToolBox InsertMenu (oldFile},3)
ToolBox InsertMenu (oldSearch},5)
ToolBox InsertMenu (oldProgram},0)
ToolBox DrawMenuBar
END SUB
```

Figure 20-4. Menu-driven program (*continued*)

The program starts by saving handles to BASIC's File, Search, and Program menus and removing those three menus from the menu bar (remember, BASIC's menus are numbered 1 to 6 from left to right). Then the program creates a new File menu (ID 98), puts items on the menu, and uses the toolbox routine InsertMenu to insert the new File menu in the menu bar. The number 3 in the argument list of the InsertMenu is the ID number of the Edit menu. The new File menu is inserted just before the Edit menu.

After inserting the new File menu, the program creates and inserts its second new menu, Choices, and redraws the menu bar. Two WHEN MENU structures, for the File and Choices menus, process your selections from the menus. Each WHEN MENU structure contains a SELECT CASE statement that uses the MENUITEM function. The statements for each case carry out the menu commands.

The program also contains a WHEN WINDOW OutputWindow] interrupt routine. If you try to close the output window while the program is running, this interrupt routine allows the program to put BASIC's menus back on the menu bar before control returns to BASIC. Your program should always remove its menus from the menu bar and restore BASIC's menus before quitting.

Each time you select a command from the File or Choices menus, the corresponding WHEN MENU structure acts on your selection and returns control to the DO loop. The program ends when you close the program's output window or select Quit from the File menu.

PRACTICE EXERCISES

1. What statement would you use to change the size of the output window to 400 pixels wide and 300 pixels tall?
2. How would you create an alert window with its top left corner at 100,100 and its bottom right corner at 300,300?
3. How would you change "About Macintosh BASIC" on the Apple menu to read "About My Program"?
4. Can you create a menu labeled "Choice" with two items named "Yes" and "No," a disabled dotted line, and an item named "Maybe"? Allow COMMAND-Y to be used to select "Yes." Let your new menu be the last one on the menu bar.

Chapter 21
Using Controls

The Macintosh is so easy to use because it has been designed to let you select and point—instead of typing long sentences or lines of special codes on the keyboard. Controls like buttons and scroll bars are key components of the Macintosh user interface. This chapter describes how you can use these controls in your programs.

TYPES OF CONTROLS

Scroll bars are the controls you see most often. There are, however, several different types of controls. All of them can be used in your programs whenever you can give an instruction or an answer by pointing. One of the most useful characteristics of a control is that it can save a numeric setting. The setting of a button is saved as on (1) or off (0), and scroll bars save the position of the scroll box.

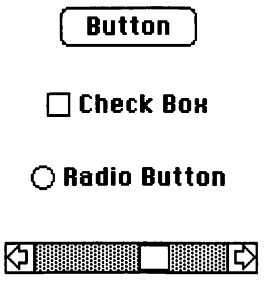


Figure 21-1. Types of controls

Controls always appear in windows, never directly on the desktop. Each control's location and size is defined by a surrounding (boundary) rectangle. Figure 21-1 shows illustrations of the four predefined kinds of controls: the button, check box, radio button, and scroll bar.

Buttons

You use buttons to trigger actions. Buttons always appear with rounded corners with the title centered inside the outline of the button. When you click a button, the program takes the action indicated by the button's title. If the title is too large to fit inside the button, the title is truncated at both ends. Buttons look best if their surrounding rectangles are 20 pixels tall.

Check Boxes

You use check boxes when you want to modify or control some future action. A check box retains one of two settings. The control is on (checked) with a setting of 1 and off (not checked) with a setting of 0. Each time you click the control, the setting is reversed.

When toolbox routines draw the check box, they draw an "X" in the check box if it is on and leave the box empty if it is off.

The title of a check box appears to the right of the box itself, but still inside the rectangle that defines the control. If the enclosing rectangle is not large enough, you lose the right end of the title. Your program is responsible for reversing the check box's setting when you click it. You should reverse the setting for a click anywhere inside the enclosing rectangle. The enclosing rectangle for a check box is usually 20 pixels tall.

Radio Buttons

Radio buttons are very much like check boxes, except that they should be used in groups. Each radio button in a group represents a mutually exclusive choice. Pressing one button should turn all the others off, just like the buttons on a car radio. A radio button is on with a setting of 1 and off with a setting of 0. Your program is responsible for turning on the radio button that has been clicked and turning off all the other radio buttons in that group. When toolbox routines draw the radio button, they draw a solid circle if the radio button is on or an empty circle if the button is off.

The title of a radio button appears to the right of the button itself but is still inside the rectangle that defines the control. If the enclosing rectangle is not large enough, you lose the right end of the title. The enclosing rectangle for a radio button is usually 20 pixels tall.

Scroll Bars

Scroll bars are one example of a larger category of controls: dials. Dials are capable of displaying a continuous range of settings. You can set the minimum and maximum of the scroll bar's range to any number from −32767 to +32767. The minimum, maximum, and actual settings must all be integers. If you try to set the scroll bar to a value below the minimum or above the maximum, it will be set to the minimum or the maximum, respectively. Scroll bars look best when the enclosing rectangle is 16 pixels wide.

MAKING A NEW CONTROL

You make a new control with the toolbox function NewControl. NewControl is a function that returns a handle to the control it creates. NewControl requires ten arguments, but that is not as complicated as it sounds. Here is a sample call to NewControl:

```
c}=Tool NewControl(w],@rect%(0),title$,vis˜,val%,min%,max%,procID%,0,0)
```

The first argument you give to NewControl is a pointer to the window in which you want to create the control. If you want to position the control in your program's normal output window, you can use OUTPUTWINDOW]. The second argument is a pointer to the first element of an integer array that defines the control's boundary rectangle. You store the coordinates of the boundary rectangle in the order top, left, bottom, and right. The third argument is the control's title, and the fourth is a Boolean expression that specifies whether you want the control to be visible. This value should almost always be true.

The fifth, sixth, and seventh arguments to NewControl are the values for the control's current setting, the minimum value, and the maximum value. The initial values for buttons, check boxes, and radio buttons should be 0,0,1. If you want a particular check box or radio button to be on as soon as it is created, use 1 instead of 0 for the first of the three values.

The eighth argument is a procedure ID number that tells NewControl which type of control to draw. Table 21-1 lists the procedure ID numbers corresponding to the four predefined controls.

Table 21-1. Control Definitions

Control Type	ProcID	Normal Width
Button	0	20
Check Box	1	20
Radio Button	2	20
Scroll Bar	16	16

The procedure ID number is 0 for buttons, 1 for check boxes, 2 for radio buttons, and 16 for scroll bars. You can make the control's title appear in the type font being used in the current window (instead of in the system font, Chicago) by giving NewControl a number that is 8 larger than the control's usual procedure ID number.

The last two arguments to NewControl should be zeros to substitute for a long integer data type. Here are several examples that create different types of controls:

```
! Button
cntrl} = Tool NewControl (w],@rect%(0),"OK",TRUE,0,0,1,0,0,0)
! Check box
cntrl} = Tool NewControl (w],@rect%(0),'Check Me',TRUE,0,0,1,1,0,0)
! Radio button
cntrl} = Tool NewControl (w],@rect%(0),title$,TRUE,0,0,1,2,0,0)
! Scroll bar
cntrl} = Tool NewControl (w],@rect%(0),"",TRUE,0,0,100,16,0,0)
```

MODIFYING A CONTROL

Once you have created a control, there are several toolbox routines that allow you to modify the control. You can move a control to a new position in the window, change the size of a control, or change the control's title by simply calling the appropriate toolbox routine.

While you are modifying a control, you can hide it so that it is not visible. Making a control invisible prevents the screen image from blinking each time the control is redrawn.

You can also read or change the value of a control by calling the appropriate toolbox routine. This allows you to turn a button on or off or change the position of the scroll box in the scroll bar.

Moving a Control

You can move an existing control to a different place in the window by calling the toolbox routine MoveControl. MoveControl takes three arguments. You first specify the handle returned from New Control and then the horizontal and vertical pixel locations where you want to move the control. The pixel locations are expressed in the coordinate system of the window in which the controls are located. For example, the location 8,12 is 8 pixels to the right and 12 pixels down from the top left corner of the window.

```
ToolBox MoveControl (cntrl},8,12)
```

MoveControl is often used to move scroll bars after a window is resized. When this happens, the call to MoveControl is usually followed by a call to SizeControl. MoveControl redraws the control at its new location.

Changing a Control's Size

The toolbox routine SizeControl changes the size of an existing control. You give it three arguments: the control handle, the control's new width, and the control's new height. SizeControl redraws the control at its new size.

```
ToolBox SizeControl (cntrl},80,20) ! Makes it 80 wide and 20 tall
```

Changing a Control's Title

Buttons, check boxes, and radio buttons have titles. You can change the title of one of these controls by using the SetCTitle routine. SetCTitle requires the handle of the control and the new title you want it to display. If your new title is too large to fit in the control's boundary rectangle, the title will be truncated.

```
ToolBox SetCTitle (cntrl},'New Title')
```

GetCTitle allows your program to get the title of a control. You give this toolbox routine a handle to the control whose title you want and a pointer to a 256-byte array in which to store the title. Then you convert the title into a BASIC string.

```
DIM title©(255)  ! 256 bytes with 0th element
ToolBox GetCTitle (cntrl},@title©(0))
length = title©(0)
title$ = ""
FOR count = 1 TO length
    title$ = title$ & CHR$(title©(count))
NEXT count
PRINT "The control's title is:"
PRINT title$
```

Making a Control Invisible

HideControl makes a control invisible. It takes one argument, the control's handle. HideControl fills the area occupied by the control with the window's background pattern. ShowControl, which also takes the handle as an argument, makes the control visible again and draws it in the window.

```
ToolBox HideControl (cntrl})
ToolBox ShowControl (cntrl})
```

HideControl is usually used to make a control invisible while you make changes to it. Because MoveControl and SizeControl both redraw the control, the control blinks when you call these routines. A simple way to avoid the blinking is to call HideControl before you move and resize the control. Then you can call Show-Control to make it visible and redraw it.

```
ToolBox HideControl (cntrl})    ! Hides the control
! Make the changes to it here
ToolBox ShowControl (cntrl})    ! Makes it visible again
```

Changing a Control's Values

You can change or find a control's current value, maximum value, or minimum value. To make a change, use the toolbox routines SetCtlValue, SetCtlMax, or SetCtlMin. Each routine takes two arguments: the control's handle and the new value you are setting.

```
ToolBox SetCtlMin (ctl},0)   ! Sets minimum to 0
ToolBox SetCtlMax (ctl},99) ! Sets maximum to 99
ToolBox SetCtlValue (ctl},3) ! Sets value to 3
```

To get the current value, maximum value, or minimum value of a control, use one of the three toolbox functions GetCtlValue, GetCtlMax, and GetCtlMin. Each of these functions requires the control's handle as an argument and returns the requested value.

```
minimum% = Tool GetCtlMin (ctl})
maximum% = Tool GetCtlMax (ctl})
setting% = Tool GetCtlValue (ctl})
```

TESTING FOR CURSOR POSITION

TestControl combines two purposes in a single function. You can use it to test whether a point is in a specific control and also to tell you in what part of the control the point is located. The point you want to test is usually the location of the cursor when the mouse button is pressed or released. TestControl takes three arguments: the first is the handle of the control you want to check, the second is the vertical coordinate of the point, and the third is the horizontal coordinate of the point. As is usual, you specify the vertical and horizontal coordinates in pixels from the top left corner of the window.

```
number% = Tool TestControl (ctrl},v% ,h% )
number% = Tool TestControl (ctl},MOUSEV,MOUSEH)
```

TestControl returns a number that depends on where the point you specify is located. It returns a different number for each control or part of a control. Buttons, check boxes, and radio buttons have only one part. TestControl returns the number 10 if the point is in a button and 11 if the point is in a check box or radio button. TestControl returns 0 if the point is not in an active control.

Scroll bars have five different parts. Figure 21-2 indicates the parts of a scroll bar and the numbers TestControl returns for each part. TestControl returns 20 for the up arrow, 21 for the down arrow, 22 for the page-up area, 23 for the page-down area, and 129

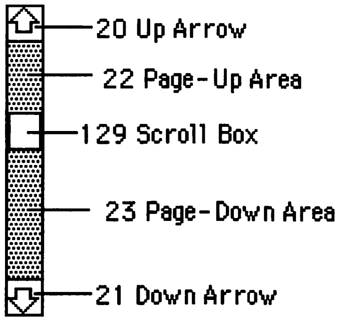


Figure 21-2. Parts of a scroll bar

if the point is in the scroll box. If the scroll bar is horizontal, the up areas are left and the down areas are right.

PUTTING THINGS AWAY

When you are finished with a control, use DisposeControl to close the control and free the memory space it occupies. DisposeControl takes one argument, the handle to the control. When you are finished, be sure to dispose of any controls you created, but do not dispose of any controls that BASIC created. If you fail to dispose of controls your program creates, you may cause problems for BASIC.

```
ToolBox DisposeControl (myControl})
```

EXAMPLE PROGRAMS

The example program in Figure 21-3 uses a scroll bar as a speed control to let you change the speed of the moving graphics you saw in Chapters 1 and 16. The value of the control is used as the

```
! Speed Control
SET OUTPUT TOSCREEN
w] = OutputWindow]
ToolBox SetWTitle (w],'Speed Control')
DIM rect%(3)
rect%(0) = 220  ! top
rect%(1) = 160  ! left
rect%(2) = rect%(0)+16   ! bottom
rect%(3) = rect%(1)+180 ! right
speed} = Tool NewControl(w],@rect%(0),"",TRUE,10,1,101,16,0,0)
DO
    CALL SetSpeed
    FOR i = 1 TO 500 STEP step%
        PAINT RECT i,30; i+120,150
        INVERT OVAL i,30; i+120,150
    NEXT i

    CALL SetSpeed
    FOR i = 500 TO 1 STEP -step%
        PAINT OVAL i,40; i+100,140
        INVERT RECT i,40; i+100,140
    NEXT i
LOOP
END PROGRAM
SUB SetSpeed
    step% = Tool GetCtlValue (speed})
    ERASE RECT 160,240; 340,270
    GPRINT AT 240,260; step%
END SUB
```

Figure 21-3. Speed Control

step increment in the two FOR/NEXT loops that draw the moving graphics. When the step is small, the graphics move slowly. When the step is large, they move very quickly. The program only recognizes the scroll box and takes no action when you press the mouse button in the other four parts of the scroll bar.

The program starts with a SET OUTPUT TOSCREEN statement that enlarges the output window. Then a copy of the pointer to the output window is saved in the variable *w]*, and the toolbox routine SetWTitle is called to put the title "Speed Control" on the output window. The program then dimensions an integer array named *rect%* to hold the control's boundary rectangle and puts the values that define the rectangle into the array.

The toolbox function NewControl creates the scroll bar and returns the control handle that the program stores in *speed}*. The NewControl arguments give an empty string as the control's title, since titles are not displayed for scroll bars. The initial control value is 10, the minimum 1, and the maximum 101. A more common range is 0 to 100, but 1 to 101 is used in this instance because the program would never get out of the FOR/NEXT loop if a step value of zero were chosen.

The moving graphics portion of the program looks almost as it did in Chapters 1 and 16. The only differences are that the variable *step%* is used for the step size, and the program now calls the subroutine SetSpeed just before each FOR/NEXT loop. The SetSpeed subroutine sets *step%* equal to the current control value and prints the new value underneath the scroll bar. The scroll bar value is updated as soon as you release the mouse button, but the program waits until the end of the loop to update the variable *step%* because BASIC does not allow you to change the step value of a FOR/NEXT loop from inside the loop.

The program in Figure 21-4 is an example that uses buttons.

```
! Show Buttons
DIM rect%(3)
SET FONT 0
GPRINT AT 30,150; "Ready to eat?"
w]= OutputWindow]
rect%(0) = 200     ! top
rect%(1) = 30      ! left
rect%(2) = rect%(0)+20  ! bottom
rect%(3) = rect%(1)+70  ! right
OK} = Tool NewControl (w],@rect%(0),"OK",TRUE,0,0,1,0,0,0)
```

Figure 21-4. Show buttons

```
rect%(1) = rect%(3)+30  ! Left is 30 pixels from OK button
rect%(3) = rect%(1)+70  ! Right is left + 70
Cancel} = Tool NewControl(w],@rect%(0),"Cancel",TRUE,0,0,1,0,0,0)
DO
    IF MOUSEB~ THEN
    h = MOUSEH
    v = MOUSEV
    IF Tool TestControl (OK},h,v) = 10 THEN
        CALL WaitButtonUp
        IF Tool TestControl (OK},MOUSEH,MOUSEV) = 10 THEN
            PRINT "Then let's eat!"
            EXIT DO
            ENDIF
        ENDIF
    IF Tool TestControl (Cancel},h,v) = 10 THEN
        CALL WaitButtonUp
        IF Tool TestControl (Cancel},MOUSEH,MOUSEV) = 10 THEN
            PRINT "No food for you, then!"
            EXIT DO
            ENDIF
        ENDIF
    SOUND
    CALL WaitButtonUp
    ENDIF
LOOP
ToolBox DisposeControl(OK})
ToolBox DisposeControl(Cancel})
ERASE RECT 0,130;200,200  ! Erase "Ready to eat?"
END PROGRAM
SUB WaitButtonUp
DO
    IF NOT MOUSEB~ THEN EXIT DO
LOOP
END SUB
```

Figure 21-5. Show buttons *(continued)*

After displaying the question "Ready to eat?" and setting up the rectangle dimensions, the program uses two NewControl calls to create buttons labeled "OK" and "Cancel." The program saves the control handles in variables named *OK}* and *Cancel}*, respectively.

Both the OK and Cancel buttons are created with an initial value of 0, or off.

The program's main DO loop waits until the mouse button is pressed. Then it uses the toolbox function TestControl to test whether the cursor is in the OK button or the Cancel button. If the button was pressed inside the OK or Cancel button, the program waits until the mouse button is released and then calls TestControl again. If the cursor was released inside the same button, the program prints an appropriate message and exits from the DO loop.

If the cursor has moved out of the button in which the mouse was first pressed or the mouse was pressed outside the two controls, the program sounds a beep, waits for the mouse to be released, and goes through the loop again. After it exits from the DO loop, the program calls DisposeControl to dispose of each control before ending.

The program in Figure 21-5 shows one way to operate a group of radio buttons. Because this program will have three controls to handle, it uses an array named *rBtns}* to store the control handles. Another array, *c$*, contains the control titles. After the arrays are set up, the FOR/NEXT loop calculates the location rectangle from the loop's index and uses NewControl to create each of the three radio buttons. Each of the radio buttons is created with a control value of 0, off.

```
! Radio Buttons
DIM rect%(3), rBtns}(3), c$(3)
w]= OutputWindow]
c$(1) = 'Yes'
c$(2) = 'No'
c$(3) = 'Maybe'
FOR i = 1 TO 3
rect%(0) = 50+20*i  ! top
rect%(1) = 30  ! left
rect%(2) = rect%(0)+20  ! bottom
rect%(3) = rect%(1)+80  ! right
rBtns}(i)=Tool NewControl (w],@rect%(0),c$(i),TRUE,0,0,1,2,0,0)
```

Figure 21-5. Radio buttons

```
NEXT i
DO
    IF MOUSEB~ THEN
    FOR i = 1 TO 3
        IF Tool TestControl(rBtns}(i),MOUSEH,MOUSEV)=11 THEN
            CALL WaitButtonUp
            IF Tool TestControl(rBtns}(i),MOUSEH,MOUSEV)=11 THEN
                FOR j = 1 TO 3
                    ToolBox SetCtlValue(rBtns}(j),0)
                NEXT i
                ToolBox SetCtlValue (rBtns}(i),1)
            ENDIF
            EXIT FOR
        ENDIF
    NEXT i
    CALL WaitButtonUp
    ENDIF
LOOP
END PROGRAM
SUB WaitButtonUp
DO
    IF NOT MOUSEB~ THEN EXIT DO
LOOP
END SUB
```

Figure 21-5. Radio buttons (*continued*)

The program's main DO loop once again keeps circling until the mouse button is pressed. When the button is pressed, a FOR/NEXT loop uses TestControl on each radio button in succession. If TestControl indicates that the mouse button was pressed in that control, the control is tested again when the mouse button is released. If the cursor is still in the same control, the program calls SetCtlValue twice, once in a loop that turns all the radio buttons off and once to turn on the radio button that was selected.

You can avoid the need to use a FOR/NEXT loop to turn all the radio buttons off each time if you use a separate variable to keep track of which button is on. In that case you can simply call SetCtlValue to turn off the button that is on before you turn on the new button.

PRACTICE EXERCISES

1. How would you create a radio button titled "push me" in the boundary rectangle whose top left corner is at 20,20? Make it visible and 100 pixels wide.
2. What statements would you use to create a button labeled "Cancel" in the boundary rectangle whose top left corner is at 30,30? Make the button visible and 80 pixels wide.
3. Change the Cancel button in Exercise 2 so that it is only 60 pixels wide, and change its title to "Quit." Be sure to hide the button during the changes so you won't see it blinking.
4. Can you write a statement that increases a scroll bar's value by 1 if the mouse is located in the up arrow area of the scroll bar? Assume *scroll}* is the name of the scroll bar's handle.

Chapter 22
QuickDraw Graphics

The QuickDraw graphics routines produce the high-speed calculations and display of graphics information on the Macintosh screen. QuickDraw underlies almost everything that happens on the Macintosh. The QuickDraw routines total about 145 in number and occupy approximately one third of the machine's read-only memory.

Like all other Macintosh programs, BASIC uses QuickDraw extensively. BASIC's GPRINT command and all of the graphics commands in Chapter 16, for instance, rely heavily on QuickDraw. This chapter introduces a few of the QuickDraw routines that you can call directly from your BASIC programs. By no means does it describe more than just a tiny bit of QuickDraw. If you want to know more about QuickDraw, you may want to get a copy of *Inside Macintosh,* published by Apple Computer, Inc.

SETTING THE GRAPHICS PORT

Every QuickDraw activity takes place within what is known as a graphics port, or *grafport.* Each window has its own grafport. Once a particular grafport has been set, QuickDraw continues using that grafport until it is changed. Technically, the location of the graphics pen is measured from the upper left corner of the grafport, not of the window. If QuickDraw is not set to use the correct grafport, graphics output will probably appear in the wrong window.

The SetPort routine tells QuickDraw which grafport to use. SetPort takes one argument, a pointer to a window or to another type of grafport.

```
ToolBox SetPort (myWindow])
ToolBox SetPort (OutputWindow])
```

BASIC calls SetPort every time it needs to draw in a different window. The only time you need to call SetPort is when you have called NewWindow to create your own window. Because BASIC does not know about the window that you created, BASIC does not call SetPort for you. If you try to draw in the window without calling SetPort, the information will not appear in that window and may appear instead in another window.

When you call SetPort, you change an important part of the environment in which BASIC operates. To avoid causing problems, you should restore BASIC's grafport setting after you have finished your drawing. You can use the GetPort routine to obtain BASIC's grafport pointer before you use SetPort and then use SetPort after your drawing is done to restore BASIC's setting. GetPort requires a pointer to a pointer variable where it will store the grafport pointer.

```
ToolBox GetPort (@oldport])  ! Save BASIC's port
ToolBox SetPort (myWindow])
! Drawing commands here
ToolBox SetPort (oldport])  ! Restore BASIC's port
```

CLIPPING THE DRAWING AREA

The toolbox routine ClipRect "clips" your graphics output within the boundaries of a rectangle that you specify. Any graphics output drawn within the rectangle will be visible, but graphics outside the boundary rectangle will not be visible. ClipRect takes one argument, a pointer to the first element of the array that defines the boundary rectangle to which the output is to be clipped. You store the coordinates of the boundary rectangle in the array in the order top, left, bottom, right.

You can use ClipRect to help you draw unusual shapes. To draw half a circle, for instance, you can set ClipRect so only half the area drawn by PAINT OVAL will be visible. If you use ClipRect this way, you should reset ClipRect to the size of the window when you are finished so you won't interfere with BASIC's environment.

```
DIM rect%(3)
rect%(0) = 120  ! top
rect%(1) = 0    ! left
rect%(2) = 240  ! bottom
rect%(3) = 240  ! right
! Show drawing only in bottom half of screen
ToolBox ClipRect(@rect%(0))
rect%(0) = 0   ! new top
! Reset cliprect to whole window
ToolBox ClipRect(@rect%(0))
```

DRAWING IN YOUR OWN WINDOW

This section brings together the different toolbox calls you need to make if you want to create and draw in your own window, separate from your program's normal output window. The listing in Figure 22-1 shows the series of steps to follow to display information in your own window. After you create your window with the NewWindow routine (as described in Chapter 21), you need to call SetPort to point QuickDraw to the correct grafport. Then you should call ClipRect to set the clipping area equal to the visible

```
! ToolBox calls for drawing in your own window
DIM rect%(3)
rect%(0) = 50  ! top
rect%(1) = 50  ! left
rect%(2) = 300  ! bottom
rect%(3) = 300  ! right
w] = Tool NewWindow (0,0,@rect%(0),"",TRUE,0,-1,-1,FALSE,0,0)
ToolBox SetPort (w])
rect%(0) = 0  ! top
rect%(1) = 0  ! left
rect%(2) = 250  ! bottom
rect%(3) = 250  ! right
ToolBox GetPort (@oldport]) ! save BASIC's Grafport
ToolBox SetPort (w])
ToolBox ClipRect (@rect%(0)) ! set ClipRect to the whole window
! Draw in the window here
ToolBox DrawGrowIcon (w]) ! draw size box for type 0 window
! Clean up after we have finished drawing
ToolBox SetPort (oldport])    ! restore BASIC's Grafport
ToolBox DisposeWindow (w])  ! close the window
END PROGRAM
```

Figure 22-1. ToolBox calls for drawing in your own window

area of your window. This step is necessary because BASIC is not aware of your window and thus cannot set the ClipRect for you. If you do not set it, some of the information you display in your window could get clipped.

To display information in your own window, you can use direct calls to QuickDraw or any of BASIC's commands that rely heavily on QuickDraw. These include the GPRINT, PLOT, FRAME, ERASE, PAINT, and INVERT commands. You should avoid BASIC's PRINT, INPUT, and CLEARWINDOW commands because they are likely to interfere with the graphics environment you have established for your special window.

When you finish drawing in your window, remember to reset BASIC's grafport and close the window, as in Figure 22-1, when you are finished with it.

ARCS

The three shapes implemented in BASIC (rectangles, ovals, and rectangles with rounded corners) are not all the graphics shapes that QuickDraw provides. Another one that is fairly easy to use is the arc. An arc is a piece of an oval. QuickDraw provides routines that let you frame, erase, paint, and invert arcs. If you frame an arc, the graphics pen moves along the circumference of the oval for the prescribed distance. If you erase, paint, or invert an arc, the shape looks like a wedge or a piece of pie.

Figure 22-2 shows several examples of arcs. The first part of the shape's definition is the rectangle that surrounds the oval. You define the rectangle in an integer array just as you do for all other toolbox routines that use rectangles. Then you supply two angles to define the beginning and extent of the arc. The angles are measured in degrees, as shown in Figure 22-2.

A starting angle of 0 is straight up, 180 is straight down, 90 is horizontal to the right, and 270 is horizontal to the left. The corners are assumed to be midway between these points, even if the rectangle is not a square, and QuickDraw adjusts the angles accordingly. Thus, the upper right corner of the rectangle is labeled 45 degrees in Figure 22-2, even though the rectangle is not square and the angle does not look like a 45-degree angle.

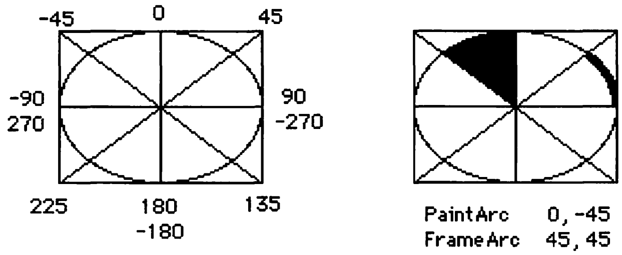


Figure 22-2. Illustrations of arcs

The second extra number you give to define the arc is the extent of the arc. The extent is not the ending point, but the angle of movement along the circumference of the arc from the starting point. The sign of the extent specifies the direction of the movement. A positive number indicates clockwise movement; a negative number indicates counterclockwise movement. An extent of 90 moves one-quarter of the way around the oval from the starting point. You can calculate the angle of the second end of the arc by adding the starting angle and the extent together.

You can frame, erase, paint, or invert an arc by calling the toolbox routine FrameArc, EraseArc, PaintArc, or InvertArc. Each routine takes three arguments—the boundary rectangle, the starting angle, and the angle of movement that defines the shape of the arc. Here are some examples of their use:

```
DIM rect%(3)
rect%(0) = 0  ! top
rect%(1) = 0  ! left
rect%(2) = 100  ! bottom
rect%(3) = 100  ! right
ToolBox FrameArc (@rect%(0),0,45)
ToolBox PaintArc (@rect%(0),90,-45)
ToolBox EraseArc (@rect%(0),180,-60)
ToolBox InvertArc (@rect%(0),270,-90)
```

PATTERNS

A pattern is a design that is defined in an area 8 pixels square. You can visualize the way a pattern is drawn into a window by imagining that the 8-pixel square is a floor tile. Once the first patterned tile is placed in the upper left corner of the output window, identical tiles are laid next to it to spread the pattern throughout the window.

The pattern definition contains eight rows with eight pixels each. You can store the pattern definition in an 8-element character array with each element of the array representing one row of the pattern. Figure 22-3 shows a magnified pattern definition.

To calculate the value of a row, add together the numbers beneath the columns that contain black pixels. For example, in

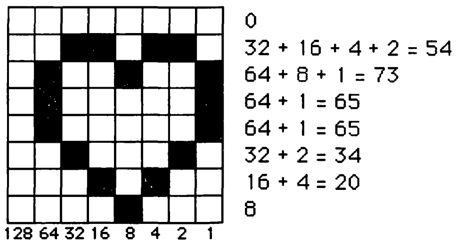


Figure 22-3. Calculating a pattern's values

Figure 22-3, the top row contains no black pixels, so the value of the first element of the character array is 0. The second row contains four black pixels. They are in the columns labeled 32, 16, 4, and 2. Adding those numbers together gives a row value of 54 for the second row.

Each pattern occupies eight bytes, so you always store a pattern in an array and pass it by reference when using it as an argument to a toolbox routine. The example program at the end of this chapter will calculate the values in the array for you.

Setting Your Own Patterns

The toolbox routine PenPat allows you to assign any pattern you want to the graphics pen. PenPat takes one argument, a pointer to the first element of an array that contains the pattern. Another toolbox routine, BackPat, allows you to set the background pattern in a grafport. The background pattern is plain white unless you change it. BackPat also takes a single argument, a pointer to the first element of an array containing the pattern definition.

```
DIM pat©(7)  ! 8 bytes for pattern
! Put values in pat©
ToolBox PenPat (@pat©(0))
ToolBox BackPat (@pat©(0))
```

Using a Pattern Without Changing the Pen

QuickDraw has a group of routines that let you use a new pattern without changing the pattern to which the graphics pen is set. The term for this activity is *fill.* A shape that is filled looks just like a shape that is painted. The only difference is that painting uses the pattern of the graphics pen, while filling uses a pattern you supply.

You can use toolbox routines to fill any of the standard shapes. FillRect fills a rectangle and FillOval fills an oval. The first argument for these routines is a pointer to an array that defines the shape's boundary rectangle. The second argument is a pointer to the first element of a character array that defines the pattern you want to use.

```
DIM rect%(3), pat©(7)
! Put values in rect% and pat©
ToolBox FillRect (@rect%(0),@pat©(0))
ToolBox FillOval (@rect%(0),@pat©(0))
```

In addition to the boundary rectangle, FillRoundRect requires two integers. After the pointer to the boundary rectangle, FillRoundRect requires the width and height of the rectangle that defines the roundness of the corners. Finally, you need to supply a pointer to the character array that defines the pattern you want to use.

```
DIM rect%(3), pat©(7)
! Put values in rect% and pat©
ToolBox FillRoundRect (@rect%(0),20,20,@pat©(0))
```

FillArc requires a pointer to the array that defines the arc's boundary rectangle, two integers representing the starting angle and the angle that defines the extent of the arc, and a pointer to the character array that defines the pattern you want to use.

```
DIM rect%(3), pat©(7)
! Put values in rect% and pat©
ToolBox FillArc (@rect%(0),0,90,@pat©(0))
```

The fill routines can be handy if you are already defining your own pattern. If you are using one of the standard patterns available in BASIC, however, you may find it just as easy to use SET PATTERN, PAINT, and then follow with PENNORMAL to reset the graphics pen after drawing.

PICTURES

A picture is really a record of operations performed by QuickDraw. Making a QuickDraw picture is like using a tape recorder. You turn the recorder on when you want to start recording, and you turn it off when you want to stop recording. You can play the recording back whenever you want to, and you can erase the tape to make room for something else.

You call OpenPicture to start recording QuickDraw commands as part of a picture, and you call ClosePicture to stop recording commands. DrawPicture redraws the picture, and KillPicture deletes the record of the picture and releases the memory it occupied for other uses.

OpenPicture is a function that returns a handle to the picture that is being defined. You use this handle with DrawPicture and KillPicture to specify what picture to use. When you call the function OpenPicture, you specify a rectangle that is to be the picture frame. QuickDraw records all of the actions that affect the area inside the picture frame and does not record actions outside the picture frame. When you later call DrawPicture to redraw the picture, you specify a new rectangle in which QuickDraw puts the picture. If the two rectangles are not the same size, QuickDraw automatically scales the picture from the size of the original frame to the size of the destination rectangle.

```
DIM rect%(3)
rect%(0) = 0  ! top
rect%(1) = 0  ! left
rect%(2) = 240  ! bottom
rect%(3) = 240  ! right
! Create your own window and set up here
!  (See Figure 22-1)
```

```
pic} = Tool OpenPicture (@rect%(0))
! Draw the picture here
ToolBox ClosePicture
! Now you can redraw the picture in any rectangle
ToolBox DrawPicture (pic})
! Delete the picture from memory when all done
ToolBox KillPicture (pic})
```

Pictures can be very handy when you want to draw something several times or in different sizes. However, there are limitations. QuickDraw can record only one picture at a time. Unfortunately, BASIC is recording a picture every time it puts graphics information in the output window. That means you cannot use the output window to record your own picture. You must create your own window to get a grafport in which you can record your picture without any interference from BASIC. Once you have called ClosePicture, however, you can safely play the picture back in any grafport, including the output window.

COLOR

QuickDraw contains two routines that allow you to specify colors for output devices such as color printers and color plotters. Each grafport has a background color, which is assumed to be white unless you change it. In addition, you can draw in one foreground color at a time in a grafport. QuickDraw can handle 32 different colors, but codes are predefined for only eight. The foreground color is black unless you change it.

You can use the ForeColor routine to change the foreground color and the BackColor routine to change the background color of the current grafport. Each routine takes two integers that are codes for the color you want. Table 22-1 lists the eight predefined colors and their codes. The color codes contain two integers each because the ForeColor and BackColor routines expect their argument to be a 4-byte long integer data type.

```
ToolBox BackColor (409,0)  ! blue background
ToolBox ForeColor (69,0)   ! draw in yellow
ToolBox ForeColor (205,0)  ! now draw in red
```

Table 22-1. Predefined Color Codes

Color	Code
White	30,0
Black	33,0
Yellow	69,0
Magenta (purple)	137,0
Red	205,0
Cyan (green blue)	273,0
Green	341,0
Blue	409,0

Even if you put color calls in your program, you will not be able to see the colors unless you have a plotter, printer, or other device that handles color output. When you are using a black and white device, every color except white appears black.

WIDTH OF A STRING

QuickDraw contains a reasonably broad selection of text-handling routines. You can call them directly, if you wish, but it is usually easier to use BASIC's GPRINT command to display text. You will, however, find one of the toolbox routines particularly handy if you need to center text in your output displays.

BASIC keeps track of the length of strings in characters. However, Monaco is the only standard Macintosh font in which all characters have the same width. All of the other fonts are proportional, with letters like "m" wider than letters like "i." To center text in a proportional font, you need to be able to measure the width of a string in pixels. The StringWidth function returns a string's width in pixels. The string to be measured is the function's only parameter. You can use any string or string expression up to a maximum string length of 255 characters.

```
width% = Tool StringWidth ('String to center')
PRINT width% ;' pixels wide.'
SET PENPOS (240-width%) DIV 2,30  ! Center it
GPRINT 'String to center'
```

The value StringWidth returns is the width of the string in the current font, size, and style. StringWidth adds up the widths of the characters in the string. If you call StringWidth after any SET FONT, SET FONTSIZE, or SET GTEXTFACE commands, the width you get from StringWidth will be the same as the string's width when you display it with GPRINT.

Finding Whether a Pixel Is Black or White

If you want to know whether a particular pixel is black or white, you can use the GetPixel function to find out. You give GetPixel the horizontal and vertical positions of the pixel whose color you want. GetPixel is a Boolean function. It returns true if the pixel is black and false if the pixel is white.

```
a~ = Tool GetPixel( 1,1)
IF Tool GetPixel( 1,1) THEN PRINT "It's black!"
```

GetPixel returns the value of the pixel as it actually looks on the screen. Do not expect GetPixel to return the value of the point in your window if the location of the pixel is outside the visible area of your current grafport or if the location of the pixel is covered by another window. Since your program does not always know whether you have moved another window on top of the output window, you may sometimes be surprised if you rely too heavily on GetPixel.

EXAMPLE PROGRAM

The Make Pattern program in Figure 22-4 displays patterns and the values that you use to define them when calling QuickDraw routines. You can use the program as a tool to help you design new patterns to use in your programs.

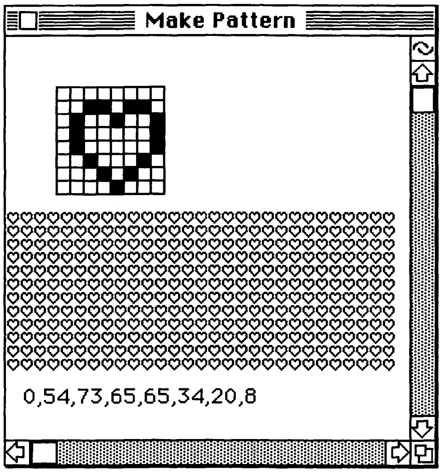
```
! Make Pattern
DIM big~(7,7), pat©(7)
l = 30  ! Left of magnified pattern
t = 30  ! Top of magnified pattern
FOR i = 9 TO 65 STEP 8
    FRAME RECT l,t; l+i,t+65
    FRAME RECT l,t; l+65,t+i
NEXT i
DO
    IF MOUSEB~ AND inrect~ THEN
        DO
            h = (MOUSEH-l) DIV 8
            v = (MOUSEV-t) DIV 8
            big~(v,h) = NOT big~(v,h)
            INVERT RECT l+8*h+1,t+8*v+1; l+8*h+8,t+8*v+8
            CALL DoPattern
            IF NOT MOUSEB~ THEN EXIT DO
        LOOP
    ENDIF
LOOP
END PROGRAM
SUB DoPattern
    ! Pack the pattern into pat©
    FOR row = 0 TO 7
        byte% = 0
        FOR col = 0 TO 7
            IF big~(row,col) THEN byte% = byte% + 2 ^ (7-col)
        NEXT col
        pat©(row) = byte%
    NEXT row
    ! Draw the pattern
    ToolBox PenPat (@pat©(0))
    PAINT RECT 0,104; 232,200
    ! Erase and redisplay pat© values
    ERASE RECT 0,201; 300,230
    SET PENPOS 10,220
    FOR row = 0 TO 6
        GPRINT pat©(row);",";
    NEXT row
    GPRINT pat©(7)  ! No comma for the last one
END SUB
FUNCTION InRect~
    InRect~ = TRUE
    IF (MOUSEH < l) OR (MOUSEH > l+63) THEN InRect~ = FALSE
    IF (MOUSEV < t) OR (MOUSEV > t+63) THEN InRect~ = FALSE
END FUNCTION
```

Figure 22-4. Make Pattern program

Clicking on a spot in the magnified pattern definition area at the top of the output window turns that spot on or off. After each change, the program displays the new pattern at normal size in the middle of the window. When it displays each pattern, the program also calculates and displays the eight values you can put into a character array to define the pattern from your BASIC programs. Figure 22-5 shows a sample of the program's output.

The program uses two arrays: *big~*, to hold the magnified version of the pattern, and *pat©*, to hold the final packed version used to set the pen pattern. The variables *l* and *t*, representing the left and top of the magnified pattern, are both set to 30. The two FRAME RECT statements inside the FOR/NEXT loop draw the squares in the magnified pattern area. Each pixel in the magnified pattern is a square, 8 normal pixels on a side.

The main loop of this program waits until the mouse button is pressed inside the magnified pattern definition area. When this happens, the program calculates the row and column in which the cursor is located by subtracting the location of the edge of the



```
0,54,73,65,65,34,20,8
```

Figure 22-5. Output from Make Pattern

magnified pattern and dividing by 8. The DIV operator truncates the result of the division, so the result is an integer from 0 to 7. The program then inverts the appropriate element of the *big~* array and uses an INVERT RECT statement to invert the square representing that pixel in the magnified display.

The subroutine DoPattern packs the information in *big~* into the pattern format in *pat©* and displays the new pattern and the values in its array. The nested FOR/NEXT loops at the beginning of the subroutine pack the eight items of information in *big~* for each row into a single byte for *pat©*. Once this is done for each of the eight rows, the program calls the toolbox routine PenPat to set the graphics pen to the new pattern and uses PAINT RECT to display an area filled with the pattern. If you put the rectangle's definition into an integer array, you could call FillRect instead of using the PenPat and PAINT RECT statements.

The final statements in the subroutine DoPattern print the values in *pat©*, the array that defines the pattern. When you find a pattern you like, you can use these eight numbers to define the pattern in your programs.

PRACTICE EXERCISES

1. How would you clip the drawing area so your program could draw only in the top 50 pixels of the window?
2. What statements would you use to draw a solid wedge that is the upper right corner of a circle contained in the rectangle 0,0;100,100?
3. Can you write statements to center the string "Title" in the leftmost 100 pixels of the output window?
4. What statements would you use in your program to set the graphics pen to the pattern displayed in Figure 22-5?

Chapter 23
Using Resources

Command:
- PERFORM

This chapter concludes the discussion of the undocumented features of Macintosh BASIC. It describes the toolbox routines that handle a second set of "hidden" files on the Macintosh. These hidden files contain packets of information called *resources*. Any kind of information can be filed as a resource, as long as it has a resource type and identification number. Common types of resources include fonts, menus, dialog box descriptions, and program code. You can retrieve, alter, and use these resources in your BASIC programs.

Two of the system resources that let you open a file and choose a name for a new file can be used easily in any BASIC program. The example programs later in this chapter demonstrate how to use these resources. Much of the other information in this chapter is of use primarily to advanced programmers and to others who are curious about the inner workings of the Macintosh.

This chapter also includes a description of how a BASIC program can call a program written in assembly language. If you want to create your own assembly language programs or create your own resources, you should obtain the Apple 68000 Development System. In addition to assembly language programming tools, that product includes a program that you can use to make new resources and put them in resource files.

RESOURCES AND RESOURCE FILES

The term *computer files* usually refers to the kinds of data file that were discussed in Chapters 12 and 13. These are the only kinds of file on most computers. The Macintosh, however, has a set of hidden files called *resource files.* Every file on the Macintosh has one part in the visible data file format and another part in the hidden resource format. The two parts of a file are sometimes called *forks.* Figure 23-1 illustrates the way a file is divided into a data fork and a resource fork.

You can use the commands described in Chapters 12 and 13 to read and write the data fork of a file. You use the toolbox routines described in this chapter to read and write the resource fork.

Identifying Resources

Almost anything can be a resource. All you have to do to specify a resource is give it a resource type and identification number. You

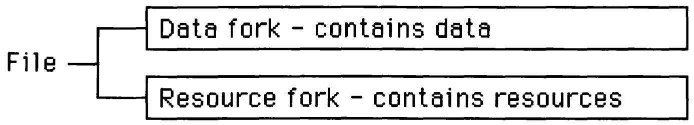


Figure 23-1. Data and resource forks of a file

can give it a name as well, if you wish. A resource type consists of four characters in which spaces and case are significant. Table 23-1 lists some of the most common resource types. Types of resources include such standard items as fonts, desk accessories, icons, strings, and segments of program code. You can define new resource types of your own as long as you do not duplicate the name of any existing resource type.

The resource identification number is an integer. The number must not duplicate the number of another resource of the same type. The range of resource numbers that is reserved for system resources varies from one resource type to another. To be safe, you should assign resource numbers higher than 256 to any resources you create.

Table 23-1. Common Resource Types

Type	Description
'ALRT'	Alert template
'CNTL'	Control template
'CODE'	Program code segment
'DITL'	Dialog or alert item list
'DLOG'	Dialog template
'DRVR'	Desk accessory or I/O driver
'FONT'	Font
'ICON'	Icon
'ICN#'	Icon list
'INTL'	International resource
'KEYC'	Keyboard configuration
'MENU'	Menu
'PACK'	Package
'PAT'	Pattern
'PAT#'	Pattern list
'PICT'	Picture
'STR'	String
'STR#'	String list
'WIND'	Window template

You can also assign a name to a resource. The name can be any legal string. Use of resource names is fairly common for fonts and desk accessories, but not very common for other types of resources. The toolbox routines that search for a resource by name do not distinguish between upper- and lowercase when looking for the resource name.

Handling Resource Files

Any special resources that a program needs are normally stored in the resource fork of that program's file. Standard resources such as fonts, desk accessories, cursors, and patterns are contained in the resource fork of the System file. While your BASIC program is running, two resource files are usually open. They are the System resource file and the resource fork of Macintosh BASIC's file.

When you request a resource, the toolbox routines search the open resource files beginning with the most recently opened file. If there is a resource in Macintosh BASIC's file with the same number as a system resource, the toolbox routines will use the one in BASIC's file instead of the one in the System file, since BASIC's file was opened after the System file.

You can open additional resource files, and you can also change the way in which the toolbox routines search resource files. To open a new resource file, use the toolbox function OpenResFile. This function takes the name of the file as its only argument and returns a reference number for the file. If the resource file is already open, OpenResFile only returns its reference number.

```
refnum% = Tool OpenResFile ('My resource file')
refnum2% = Tool OpenResFile ('Resource file 2')
```

To change the way the toolbox routines search files for a resource, you can use the toolbox routine UseResFile. UseResFile tells the toolbox routines to begin searching for resources with a particular file. UseResFile takes one argument, the reference number of the first file to be searched. The reference number of the System resource file is zero. Calling UseResFile does not change the order in which files are searched.

```
ToolBox UseResFile (refnum%)
ToolBox UseResFile (0)  ! Use System file only
```

Using a Resource Type as an Argument

To pass a resource type as an argument to a toolbox routine, you must pack the four characters into two integers. You use BASIC's ASC function to convert each character into a number, multiply the leftmost character for each integer by 256, and add the values of the two characters for each integer together as in the following example:

```
! Example for type 'FONT'
first2% = ASC('F')*256 + ASC('O')
second2% = ASC('N')*256 + ASC('T')
! As parameter use second2%,first2%
```

In the argument list to a toolbox routine you must first list the integer variable that contains the last two letters followed by the integer variable that contains the first two letters. This is required because BASIC does not have a special variable type to match the packed array of four characters required by the toolbox routines.

Getting a Resource

The toolbox functions GetResource and GetNamedResource provide ways for your program to get any available resource. You use GetResource if you want to search for a resource by its ID number and GetNamedResource if you want to look for it by name. Both functions require you to supply the resource type first and then the ID number or name. Both functions search the open resource files, load the resource into memory (under normal circumstances), and return a handle to the resource. You can use the handle to refer to the resource when you call other toolbox routines.

```
first2% = ASC('P')*256 + ASC('I')
second2% = ASC('C')*256 + ASC('T')
h} = Tool GetResource (second2%,first2%,350)  ! Gets PICT 350
h} = Tool GetNamedResource (second2%,first2%,'scene')
```

Listing Resource Types

The toolbox contains routines that let you find out what resources are present in the open resource files. CountTypes is a function that returns the number of different resource types present in the open resource files. CountTypes requires no arguments.

GetIndType is a toolbox routine that returns the name of the resource type specified by an index from 1 to CountTypes. The two toolbox calls are usually used in combination. GetIndType requires two arguments: a pointer to the first element of a 4-byte array where the resource type name will be stored, and the index of the resource type to be stored. The following example shows how you can use the two toolbox calls to display a list of all types of resources contained in the open resource files.

```
! List available resource types
DIM type©(3)
FOR count = 1 TO Tool CountTypes
    ToolBox GetIndType (@type©(0),count)
    string$ = ""
    FOR index = 0 TO 3
        string$ = string$ & CHR$(type©(index))
    NEXT index
    PRINT string$
NEXT count
```

Listing Individual Resources

The toolbox function CountResources returns the number of separate resources present of a given type. This function requires you to supply the resource type as an argument. The function GetIndResource returns a handle to the resource that matches the resource type and index number you supply as arguments. These two functions are usually used together. The following example uses CountResources and GetIndResource to fill an array with handles to all of the FONT resources available:

```
! Get handles to all available fonts
first2% = ASC('F')*256 + ASC('O')
second2% = ASC("N")*256 + ASC("T")
total = Tool CountResources (second2%,first2%)
```

```
DIM h}(total)  ! dimension array for the handles
FOR count = 1 TO total
    h}(count) = Tool GetIndResource (second2% ,first2% ,count)
NEXT count
```

Having a handle to a resource does not tell you very much about it. If you used GetIndResource with an index to get the handle, you usually want to know the resource's ID and name. You can use the toolbox routine GetResInfo to get them.

GetResInfo requires four arguments. First you supply the handle of the resource about which you want information. Then you supply a pointer to an integer variable into which GetResInfo will store the resource ID number. Finally you must supply pointers to arrays into which GetResInfo will store the resource type (4 bytes) and name (256 bytes). The example in Figure 23-2 lists all of the

```
! List available desk accessories
DIM type©(3), name©(255)
first2% = ASC('D')*256 + ASC('R')
second2% = ASC("V")*256 + ASC("R")
FOR count = 1 TO Tool CountResources (second2% ,first2% )
    h} = Tool GetIndResource (second2% ,first2% ,count)
    ToolBox GetResInfo (h},@ID% ,@type©(0),@name©(0))
    IF CHR$(name©(1)) <> "." THEN ! Not a desk accessory if it is "."
        ! Get resource type in a string
        rtype$ = ""
        FOR index = 0 TO 3
        rtype$ = rtype$ & CHR$(type©(index))
        NEXT index
        ! Get resource name in a string
        rname$ = ""
        FOR index = 1 TO name©(0)
            rname$ = rname$ & CHR$(name©(index))
        NEXT index
        PRINT rtype$;"  ";ID% ,rname$
    ENDIF
NEXT count
END PROGRAM
```

Figure 23-2. List available desk accessories

available desk accessories (a desk accessory is resource type DRVR with a name that does not start with a period).

RESOURCES YOU CAN USE

In many cases, you can shorten your programs by using resources. For instance, you can have most aspects of windows and menus defined in a resource. Instead of specifying the entire set of arguments every time that you create a window, you can use a toolbox function with a much shorter parameter list to get the window specifications from a resource. If the resource you need does not already exist in a resource file, you need to create it with a program from the Apple 68000 Development System.

Windows

The toolbox function GetNewWindow allows you to create a new window from a window template (type WIND) in a resource file. It requires fewer arguments than the NewWindow function described in Chapter 20. GetNewWindow requires the resource ID number of the WIND resource, two zeros to tell the toolbox routines to allocate their own storage area, and the number −1 twice to tell the toolbox routine that this new window will be in front of all others. After it creates the new window, GetNewWindow returns a window pointer.

```
w] = Tool GetNewWindow (300,0,0,-1,-1)
```

Menus

The toolbox function GetMenu creates a new menu from a MENU resource. You provide a single argument, the ID number of the MENU resource. GetMenu returns a handle to the new menu. If you have all of the individual items already defined in the resource, GetMenu gives you a menu that is ready to install in the menu bar.

```
myMenu} = Tool GetMenu (333)
ToolBox InsertMenu (myMenu},0)
ToolBox DrawMenuBar
```

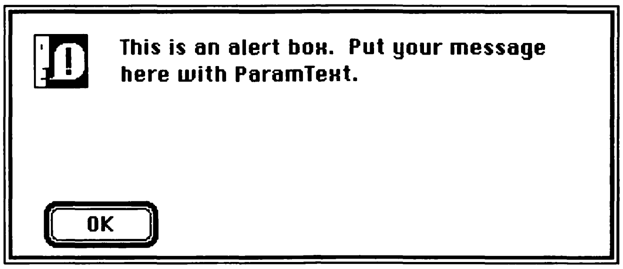
Pictures

You can get a picture (resource type PICT) from a resource file with GetResource or GetNamedResource and then give the resource handle to DrawPicture to draw the picture. This allows you to make the picture once and to read it from disk and redraw it quickly any time you run your program.

```
first2% = ASC("P")*256 + ASC("I")
second2% = ASC("C")*256 + ASC("T")
pic} = Tool GetResource (second2%,first2%,333)  ! Get PICT 333
ToolBox DrawPicture (pic})
```

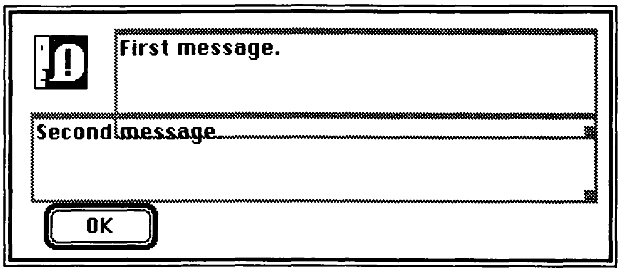
Alert Boxes

Macintosh programs often use alert boxes to tell you when something cannot be done or to alert you when you are about to do something that could cause you to lose information. Alert boxes can contain text, icons, and controls, as in this example:


This is an alert box. Put your message here with ParamText.

Because alert boxes can be very complex, the only reasonable way to use them is to have them stored as resources in a resource file. Luckily, two of the alert resources (type ALRT) in Macintosh

BASIC's resource file are general enough for you to use in your BASIC programs. This is a diagram of the resource ALRT 10:



The two areas outlined in gray are text display areas. You use the toolbox routine ParamText to set the text to be displayed in these areas. The gray outlines are not displayed. ParamText takes four strings as arguments, even if (as in this example) the alert does not have that many text display areas. If you do not want to display a message in one of the text areas, you can supply an empty string for that area.

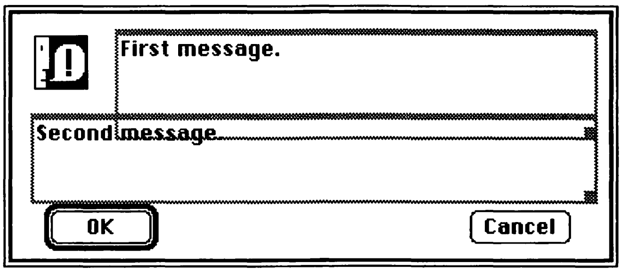
```
ToolBox ParamText ("First message.","Second message.","","")
ToolBox ParamText ("Only one message","","","")
a$ = "Give this message"
ToolBox ParamText (a$,"","","")
```

After you have used ParamText to set the text, you call the toolbox function Alert to display the alert box. This function requires three arguments: the number of the ALRT resource and two zeros. This sequence displays the alert shown previously:

```
! Display ALRT 10
ToolBox ParamText ("First message.","Second message.","","")
itemHit% = Tool Alert ( 10,0,0)   ! displays the alert ALRT 10
! itemHit% will always be 1 in this example
```

The Alert function displays the alert box and handles everything related to the operation of the alert. It does not return control to your program until the RETURN key is pressed or a button in the alert box is clicked. The function returns an integer that tells you which button was clicked. In this first example, there is only one button (the OK button) and the function Alert always returns the number 1. When an alert box contains more than one button, pressing the RETURN key has the same effect as clicking on the button that is boldly outlined.

Because the ALRT 10 box just shown contains only one button, it is useful only to deliver a message. If you need an answer to a question, you need an alert box with two buttons. The ALRT 1 box in BASIC's resource file looks like the ALRT 10 box with the addition of a second button, labeled Cancel.


First message.
Second message.

The routine to display the ALRT 1 box is similar to the code for ALRT 10:

```
! Display ALRT 1
ToolBox ParamText ( "First message.","Second message.","","")
itemHit% = Tool Alert ( 1,0,0)      ! displays the alert ALRT 1
IF itemHit% = 1 THEN PRINT "Selected 'OK' or pressed RETURN"
IF itemHit% = 2 THEN PRINT "Selected 'Cancel'"
```

This alert box contains two buttons. Your program can find out which button was chosen by testing the number returned by the Alert function. The number is 1 for the OK button or the RETURN key and 2 for the Cancel button.

CALLING CODE FROM A RESOURCE FILE

Segments of program code can be filed as resources and called from your BASIC programs. The ability to execute code from a resource file makes BASIC a truly flexible and expandable language.

Calling an Assembly Language Routine

■ PERFORM

As it was described in Chapter 15, BASIC's PERFORM command executes another BASIC program and then returns to the next statement after the PERFORM statement. In addition to this, PERFORM has an extra, undocumented capability that lets you execute any assembly language subroutine that receives its parameters according to the toolbox parameter-passing conventions. Advanced programmers can use this feature of PERFORM to call assembly language subroutines written with the assembler in Apple's 68000 Development System.

The assembly language subroutine you want to execute should be located in a resource file. If the file is not already open, you can use OpenResFile to open it. Then you use GetResource with the subroutine's resource type and ID to get a handle to it. You execute the subroutine by using the command PERFORM followed by the resource handle you received from GetResource.

```
! Call Assembly Language Code
! For example, assume it is in resource CODE 333
t1% = ASC("C")*256 + ASC("O")
t2% = ASC("D")*256 + ASC("E")
h} = Tool GetResource(t2%,t1%,333)  ! reads code and returns handle
PERFORM h} (p1%,p2%)  ! Call the code, using two integer parameters
ToolBox ReleaseResource(h})  ! release code from memory
```

If the subroutine requires arguments, put them in parentheses just after the handle. Use the same parameter-passing procedures you use to pass parameters to toolbox routines. After the subroutine has been executed, you can call ReleaseResource with the handle as an argument to remove the subroutine from memory. Call ReleaseResource only if no other program uses the resource.

You do not get any help from BASIC in checking your arguments to the assembly language subroutine. As with most assembly language programming, you are on your own. If the arguments you supply occupy more or fewer bytes than required by the assembly language subroutine, your program will almost certainly stop with a fatal error. Other programming mistakes may cause fatal errors or more subtle problems.

Packages

Packages are sets of toolbox-like routines that are not in your machine's read-only memory. Instead, the code for these routines is located in the resource fork of the System file on your disk. Packages have the resource type PACK. Each package contains the code for several related routines.

Table 23-2 shows the standard list of packages and the types of routines contained in each. BASIC calls the Disk Initialization, Floating-Point Arithmetic, Transcendental Functions, and Binary-Decimal Conversion packages whenever they are needed. You should not need to call them directly. The Standard File package contains two routines, SFGetFile and SFPutFile, that you can use from BASIC. Example programs that use SFGetFile and SFPutFile are described in detail at the end of this chapter.

The International Utilities package contains utilities that convert the time and date to the proper format for each country. It also contains utilities that handle string comparisons according to each country's normal usage. When you use the OPTION COLLATE NATIVE command described in Chapter 8, BASIC uses the International Utilities package routines to perform string comparisons.

Table 23-2. The Macintosh System Packages

Resource ID	Description
'PACK' 2	Disk Initialization
'PACK' 3	Standard File
'PACK' 4	Floating-Point Arithmetic
'PACK' 5	Transcendental Functions
'PACK' 6	International Utilities
'PACK' 7	Binary-Decimal Conversion

Using the Packages From BASIC

To use a routine from one of the packages in your BASIC program, you have to load the package and then use PERFORM to execute the routine you want. Since BASIC does not recognize the names of the individual package routines, you need to supply an extra argument in the PERFORM statement to indicate which routine in the package you wish to select. You add this selector code to the end of the routine's normal argument list. Table 23-3 shows the selector codes for each of the routines in the Standard File and International packages.

Do not call ReleaseResource when you are finished using a package resource. You may not be the only one using the package. If you call ReleaseResource for a package BASIC had loaded into memory, a fatal error will result the next time BASIC tries to call a routine in that package.

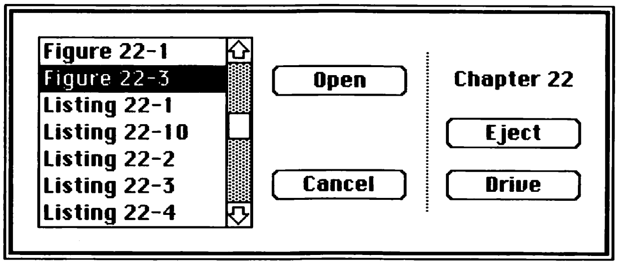
Table 23-3. Package Routine Selector Codes

Routine	Code
Standard File (PACK 3)	
SFPutFile	1
SFGetFile	2
International (PACK 6)	
IUDateString	0
IUTimeString	2
IUMetric	4
IUGetIntl	6
IUSetIntl	8
IUMagString	10
IUMagIDString	12
IUDatePString	14
IUTimePString	16

EXAMPLE PROGRAMS

This chapter's example programs show you how to use two of the most useful Macintosh resources, SFGetFile and SFPutFile, which control the dialog boxes to open and name files.

SFGetFile presents and manages the standard dialog box that every Macintosh application is supposed to present when you ask it to open a file.



You call SFGetFile by using GetResource to load the resource PACK 3 and calling the resource with selector code 2 at the end of your argument list. The program in Figure 23-3 is an example that uses SFGetFile.

This example program looks long, but it is not much more complicated than previous examples. It uses the GetResource function to load PACK 3, as described earlier in this chapter. The PERFORM statement in the middle of the program actually executes the SFGetFile code. The first two arguments are the horizontal and vertical coordinates of the pixel where the top left corner of the dialog box is to be located on the screen. The location is followed by an empty string and two zeros.

The next two arguments for SFGetFile are the number of file types you want the routine to select for display in the scrolling box and a pointer to an integer array that contains the packed file types. The packed file types must be stored in the array before you attempt to use SFGetFile. This example tells SFGetFile to display files of two types, TEXT and BINY. If you specify −1 for the number of file types, SFGetFile displays all files on the disk regardless of their file types.

```
! Call SFGetFile
DIM reply©(73)    ! result goes here
DIM TList%(3)     ! list of two types
! Store file types to be selected ('TEXT' and 'BINY')
TList%(0) = ASC("T")*256+ASC("E")
TList%(1) = ASC("X")*256+ASC("T")
TList%(2) = ASC("B")*256+ASC("I")
TList%(3) = ASC("N")*256+ASC("Y")
! Load PACK 3
t1% = ASC("P")*256+ASC("A")
t2% = ASC("C")*256+ASC("K")
h} = Tool GetResource(t2%,t1%,3)  ! reads PACK 3 code into h}
! Call SFGetFile
PERFORM h} (100,80,"",0,0,-1,@TList%(0),0,0,@reply©(0),2)
IF reply©(0) = 0 THEN END PROGRAM   ! Cancel was selected
volume = reply©(6)*256 + reply©(7)
IF volume > 32767 THEN volume = volume -65536
SETVOL -v
filename$ = ""
FOR i=11 TO 10+reply©(10)  ! reply©(10) is length
    filename$ = filename$ & CHR$(reply©(i))
NEXT i
PRINT "File name: "; filename$
OPEN #3: filename$
! Read the file and process the information
CLOSE
END PROGRAM
```

Figure 23-3. Call SFGetFile

You should always use two zeros after the pointer to the list of file types and follow the zeros with a pointer to an array that contains at least 74 bytes. SFGetFile stores a reply record in this array that contains information you will need. The last argument, the number 2, is the selector code that tells PACK 3 to execute SFGetFile.

Table 23-4 lists the items that SFGetFile stores in the reply record. The 0th element of the array is a Boolean variable. If the value of this element is 0, the Boolean variable is false, meaning that the Cancel button in the SFGetFile dialog box was pressed. In

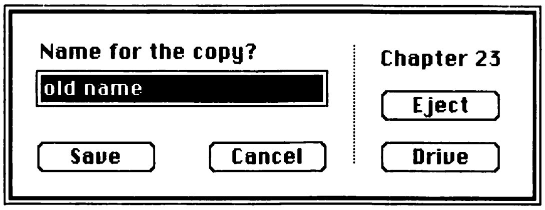
Table 23-4. SFGetFile Reply Record

Start	Number Of Bytes	Data Type	Description
0	1	Boolean	True (>0) if OK, False if Cancel
1	1	Boolean	Not used
2	4	4 characters	File type
6	2	Integer	Volume reference number
8	2	Integer	File's version number (usually 0)
10	1	Number	Length of file name
11	63	Characters	File name

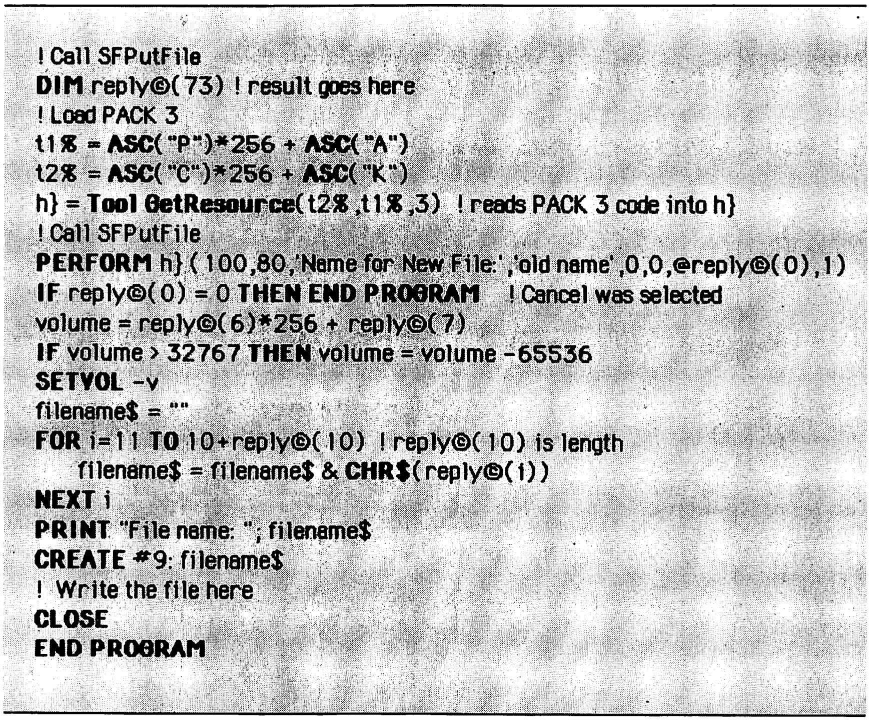
this case, your program should not open a file. If the 0th element is greater than 0, the Open button was pressed and your program can open a file. The example program ends with an END PROGRAM statement if the Cancel button was pressed.

The length of the file name is contained in the tenth byte of the reply array. The FOR/NEXT loop at the end of the example program in Figure 23-3 shows how you can assemble the file name. Once you have the name of the chosen file, your program can open the file and do its work.

SFPutFile presents the "Save As" dialog box that lets you specify a new file's name.



The steps you use to call SFPutFile are similar to those used for



```
! Call SFPutFile
DIM reply©(73) ! result goes here
! Load PACK 3
t1% = ASC("P")*256 + ASC("A")
t2% = ASC("C")*256 + ASC("K")
h} = Tool GetResource(t2%,t1%,3) ! reads PACK 3 code into h}
! Call SFPutFile
PERFORM h}( 100,80,'Name for New File:','old name',0,0,@reply©(0),1)
IF reply©(0) = 0 THEN END PROGRAM  ! Cancel was selected
volume = reply©(6)*256 + reply©(7)
IF volume > 32767 THEN volume = volume -65536
SETVOL -v
filename$ = ""
FOR i=11 TO 10+reply©(10) ! reply©(10) is length
    filename$ = filename$ & CHR$(reply©(i))
NEXT i
PRINT "File name: "; filename$
CREATE #9: filename$
! Write the file here
CLOSE
END PROGRAM
```

Figure 23-4. Call SFPutFile

SFGetFile, but a little bit simpler. Figure 23-4 is an example program that shows how to call SFPutFile. Once again, you use GetResource to give you a handle to PACK 3. Then you call it in a PERFORM statement with a selector code of 1 to specify SFPutFile.

The first two arguments for SFPutFile are the horizontal and vertical coordinates where you want the dialog box located on the screen. Then you supply two strings: a prompt line for the dialog box and the file name that is to be displayed when the dialog box first appears. After two required zeros, the program once again supplies a pointer to a 74-byte array for a reply record. The final 1 is the SFPutFile selector code.

SFPutFile fills most of the reply record, but it does not store the file type information. If the first byte of the reply record is zero, the Cancel button was selected and the program ends without creating and writing a file.

PRACTICE EXERCISES

1. Can you write a short program that opens a resource file named "MoreResources" and then lists for each type of resource the number of available resources of that type?
2. How would you display an alert box that warns "There is not enough room on the disk to save that file"? That type of alert does not require a choice, so it does not need any buttons other than the OK button.
3. How would you display an alert box with the message "The percentages add to less than 100%. Do you want to continue the calculations anyway?"
4. Assume your program has its own menu and one of the items on the menu lets you rename a file. If the present name of the file is in the variable *name$*, how might you present a dialog box to get the new name for the file?

Part five
Programming Style

Chapter 24
Professional Polish

Throughout this book, you have been both the programmer and the program user. Now it is time to separate those two roles. This chapter discusses some of the special considerations involved in giving your programs a professional polish.

Most discussions of good programming practices focus on the goal of making things easy for programmers. Those practices include using variable and subroutine names that are self-explanatory, structuring a program so it is easy to follow, using comments to explain parts of the program where the code may be unclear, and using modular code that you can debug once and then use again in other programs. All of those practices are important, particularly when more than one programmer is working on the same program.

This chapter, however, is about two additional goals that come to the fore when you look at a program from the point of view of the person who uses it. The first goal is naturalness of use. The second goal is speed.

NATURALNESS

There is no reason why anyone should have to change work habits to match a machine. It is much better for designers of the machines and of the programs that run on machines to design their products to match the habits and styles of people.

In the early days of the computer industry, people expected to spend hours learning how to use a computer program. Programs often required arcane codes and provided cryptic messages. Computer programs were expected to take a person through a problem or process from beginning to end in the most "logical" fashion. What was logical to the computer programmer who designed the computer program was not necessarily logical to the person using the program.

It may have been acceptable for computer programs to behave that way when access to computers was limited to a small cadre of the technical elite. Learning all those things was just part of the price of admission to the elite. But as powerful computers like the Macintosh become widely available, that kind of program behavior becomes totally unacceptable.

The design of the Macintosh and its user interface is based on years of research to determine the most natural ways for computers to interact with people. You can take advantage of that research by using the standard Macintosh user interface—including the mouse, windows, menus, controls, alert boxes—in your programs.

The best way to learn how to use the parts of the Macintosh user interface in your programs is to study the way they are used in programs like MacWrite and MacPaint. If your program looks and works like those programs, you have probably used the elements of the interface correctly. If you plan to write programs for the commercial market, you might want to read Apple Computer's publication, *Inside Macintosh,* which includes a detailed specification for the use of each element of the Macintosh user interface.

The overall effect you should try to achieve in your program's user interface is to make everything seem natural, with no surprises. The program's appearance and behavior should be consistent with the way people normally work and with other Macintosh programs. If you take the trouble to use the mouse, menus, controls, windows, and other elements of the Macintosh user interface properly, you will be rewarded with programs that take almost no time to learn and are easy (even fun) to use.

INPUT CHECKING

When you are writing a program just for yourself, you do not always need extensive input checking. After all, you know what input you wrote the program to handle, so you are not very likely to try to type things the program will not accept. Besides, you are a programmer, so if the program stops with some error, you can always look at the program, fix it, and then restart.

One of the steps involved in giving a program professional polish is making certain that the program will be able to handle any type of input correctly, no matter how unexpected the input is. Two techniques are available: you can use controls to limit the input to things the program can handle, or you can allow any input and have the program present an error message if it cannot handle the input.

Your program can avoid error message situations by providing gentle guidance. If, for example, the only possible answers to a question are yes and no, the program should offer buttons reading "Yes" and "No." The buttons serve two purposes: their appearance tells the person using the program that a choice needs to be made, and they limit the range of choices to the two the program is prepared to handle. Most problems with input arise in programs that require all input to be typed on the keyboard. You can avoid these problems (and make your program easier to use) by using the Macintosh controls.

Whenever it must receive input from the keyboard, your program needs to check that input very carefully. If a number needs to be less than 100, the program should check that the number *is* below 100. When the program receives a ZIP code or telephone number, it should check to see that it contains the correct number of digits. If the input is incorrect, the program should respond with a message that explains what is needed.

The messages your program displays when it receives incorrect input should clearly tell the operator what is expected. Once computer programs gave cryptic messages like "Illegal input" or "BDOS error on A." That kind of message causes puzzlement and frustration. Instead of "Illegal input," your message should clearly say "Please type a number" or "Expected a number" if that is what you mean.

TESTING

Test your program completely. Try every path. Watch others use it. Ask other people to help you test it. After you fix any problems the testing uncovers, test it again.

Test for errors in the user interface as well as for more obvious errors like incorrect answers. Look for places in the program where people may not know what to do next. Does the program make a person do routine tasks that could be done automatically by the program? Are there any shortcuts you could provide to reduce the number of steps a person has to take to get results?

Once your program works for the easy cases, try some unexpected things. Your testing goal is to eliminate potential problems. If your program is used by a large number of people, it will eventually receive almost every possible combination of instructions. You cannot try every one during testing. You can, though, deliberately try unusual combinations and extreme cases to make certain your program handles them correctly.

As the programmer, you know your program better than anyone else. If you don't know what it will do in a particular situation, then you need to find out. Try all the unusual cases. If the program takes input from the keyboard, see what happens if you press the RETURN key without typing anything. Try pressing unusual keys at random to see if the program handles them correctly. If some of the keyboard input is supposed to be numbers, go through your entire program at least once, typing letters at every prompt to make certain the program handles all the errors properly.

When your program performs calculations, test the calculation routines to be sure they work with unusual data. What happens with a very large number or a very small number? Does the program work with positive numbers but fail if you give it a negative number? If it expects an integer, what happens if you give it a fraction? Does the program give the correct results with zero and plus and minus infinity? Are inappropriate input values stopped with an appropriate corrective message?

PROGRAM SEGMENTATION

If your program is very large, you may have to break it into pieces (called segments) to fit it into the machine. Try to find logical places to split the program where it shifts from one activity to another. Many programs follow a sequence of setup, data acquisition, calculation, and displaying results. The place where the program switches from one of those activities to the next is a logical place to divide the program.

When you split a program, you should add a PROGRAM statement (described in Chapter 15) at the beginning of each part of the program and save the parts in separate disk files. Each part of your original program is now a separate program, which can be executed with BASIC's PERFORM command.

SPEED, TIMING, AND RHYTHM

A program that is slow can be almost as frustrating as a program with an old-fashioned user interface. This section discusses a few techniques you can use to identify the slow portions of your program and to increase their speed. It also discusses two subtle aspects of program speed: modifying the order in which the program does things (timing), and matching the time during which the program does things to the way people work (rhythm).

If you think your program operates too slowly, the first thing to do is to identify the places in the program that you think may cause the problem. Run your program all the way through, making notes every time you think something happens too slowly. If you have trouble identifying the part of the code that causes the slow response, using the debugger's Trace command may help you find the problem area.

Now look at the slow parts of your program. If the slowness is caused by file operations or drawing graphics, there may not be a lot you can do. Large graphics do take longer to draw than small graphics. Be sure you are drawing only what you need.

One of the common causes of slowness that you can often remedy is loops. A loop that executes unnecessary statements a hundred or more times can slow a program. Check the loop to make sure every statement in the loop has to be there. If you can, move some of the statements in the loop to outside the loop, where they will be executed only once.

If reducing the contents of loops does not provide all the speed improvement you need, you may need to rewrite some of the program. BASIC takes slightly less time to reference a simple variable than it does to reference an array variable. The first 64 variables that your program uses (including labels and defined functions) are referenced slightly faster than other variables. Computations take less time with integers than with reals, and similarly, take less time with extended-precision than with double-precision variables.

You can make additional speed improvements by using global variables instead of parameters in calls to subroutines and defined functions. Subroutines and functions without parameters run faster than those with parameters. FOR/NEXT and DO loop structures run faster than IF/THEN and SELECT CASE structures. Multiplication and the SQR function are both much faster than exponentiation.

If you want to compare the speed of two different statements to add your own findings to the speed guidelines listed here, a standard way to do the comparison is to put one of the statements you want to compare inside a FOR/NEXT loop that will execute the statement 5000 or 10,000 times. Then time the program. Now replace the statement inside the FOR/NEXT loop with the statement you want to compare it with, and time it again. The statement that was in the version that ran in the fastest time is the one you want to use in time-sensitive parts of your programs.

Timing refers to when your program does things. If your program seems to respond too slowly, you might want to examine the way it is being used. If you can locate any spots where people tend to pause (perhaps to read a screenful of information the program has just displayed or to listen to music or sound effects), those spots are places where the program can do time-consuming operations without any delays being noticed.

Many rough spots that sem to be caused by speed problems can be cured by improving program timing. Consider, for example, a program that needs to write the results of its calculations to a disk file and also display the results in the program's output window. If the program writes the results to the file before putting them in the output window, the person who wants to see those results has to wait for the file operation. If the program displays the results in the output window first, the person is busy reading the results and may not even notice that the file operations are taking place.

If your program uses the PERFORM statement to execute other programs, you need to make certain that the delay while your program loads a new segment occurs at a time that is as unobtrusive as possible. You may need to change the place where you originally split the program in order to obtain smooth, unobtrusive performance.

The ultimate timing goal is to fine-tune until the program performs in a rhythm that feels natural to the person using it. Good rhythm is seldom noticed. Bad rhythm, however, can ruin an otherwise wonderful program.

As a hypothetical example of bad rhythm, imagine a program that requires you to type a long list of paired words. For some reason, the program takes longer to process the first word in each pair than it takes to process the second word. Most people would find the rhythm of this program unnatural, because it pauses in the middle rather than at the end of each pair of words.

A program with an extremely unnatural rhythm like the one in this example is likely to make the user frustrated, tense, and nervous. The person may not realize the exact cause of this agitated state, but will probably associate the negative experience with the program or the computer and refuse to use them again.

The only sure way to identify an unnatural rhythm in a program is to use it. Different people may have slightly different expectations, so it is a good idea to watch other people use the program as well. You should make notes of places where people seem to be frustrated or have to wait for the program, and then ask them about those places afterwards. With a little care, you can make certain that your program has a natural and pleasing rhythm.

PROFESSIONAL APPEARANCE

Why not make your program look as good as possible? If the internal workings of your program are truly professional, it is worth the time to polish the program's appearance as well. This means making sure all the spelling and grammar in the displays and messages are correct, arranging output attractively, and centering text in the window or over a column if it looks better that way.

It also means making judicious use of graphics and sound if they are appropriate to the program.

DOCUMENTATION

If your program will be used by other people, you need to provide some written documentation. The written documentation serves as an introduction to the program and its capabilities and as a reference while the program is being used. If your program is so self-explanatory that it does not need any pictures with the written documentation, you may be able to provide the documentation as a text file on the program disk.

You may have noticed that some people learn more quickly when they read material on a printed page, others learn more quickly by listening, and still others learn more quickly by doing. If the nature of your program makes it possible, you might consider offering all three methods for learning how to use your program. If you examine the Guided Tour materials that Apple Computer enclosed with your Macintosh, you will see that they have provided adequate help for all three ways of learning.

IMPONDERABLES

When everything else is done, why do people like some programs and dislike others? What gives one program a pleasing character? What makes other programs aggravating to use? If you find a way to get exact answers to these questions, you have a great future in product selection and marketing.

In the absence of certainty, you can do some experiments. Watch other people use your program. If they look puzzled or frustrated, note where they were in the program and what they were trying to do. Use the program yourself. If you find yourself waiting for the program or getting irritated, you have some work to do. Rewrite or redesign the parts of your program that seem to be troublesome. Test it again and again if necessary.

Once everything works correctly and you have eliminated trouble spots, look at your program as a whole. Does it present a consistent design? Do the parts fit together into a coherent whole? Even if it is not a game, is your program just a little bit fun to use? If so, you may have the makings of a great commercial product.

Appendix A Macintosh BASIC Statements and Functions

This appendix presents formal descriptions of the syntax of all statements and functions in Macintosh BASIC. In these descriptions the Macintosh BASIC keywords are in boldface type. Items you need to supply are in plain type. Items in square brackets are optional, while items that are not in square brackets are a required part of the syntax. For example, in the syntax description

GOSUB label[:]

the word GOSUB is a BASIC keyword, and you supply the label when you write the program statement. The colon is enclosed in square brackets, so its use is optional.

For many BASIC statements, you have a choice of words or syntax. The set of choices is enclosed in curly braces ({ }) and the choices are separated from each other by a vertical bar (|). The vertical bar means "or." For example, the syntax description

OPTION COLLATE {NATIVE | STANDARD}

indicates that you can use either **OPTION COLLATE NATIVE** or **OPTION COLLATE STANDARD.**

An ellipsis (. . .) just before a closing bracket or brace means that you can repeat the portion of the syntax that is inside the brackets or braces. The description

READ variable [, variable . . .]

indicates that a READ statement must always have one variable name and may have any number of additional variable names preceded by commas. The comma and variable name inside the square brackets must always be used together because there are no curly braces or vertical lines to indicate that a choice is allowed.

A slash (/) indicates the end of a statement in a multiple-statement control structure. Usually, the end of a statement is the same as the end of a line. You can, however, use a colon to end a statement if you want to put more than one statement on a line. The syntax description

DO / [statements /] **LOOP**

indicates that you have the option of including additional statements between the DO statement and the LOOP statement.

Any punctuation other than square brackets ([]), curly braces ({}), a vertical bar (|), an ellipsis (. . .), or a slash (/) is a part of the punctuation of the BASIC statement or function.

Descriptive names are used in the syntax descriptions for the items you must supply. A *literal* is an actual value, not the name of a variable. A string literal should include matching quotation marks at the beginning and end of the string value.

Variable means the name of a variable, not its value. *Numvar* is an abbreviation for numeric variable. *IntegerArray* means the name of an array of integers. *Handle* means the name of a variable of type handle and *pointer* means the name of a variable of type pointer.

An *expression* is any combination of values, variables, functions, and operators that results in a single value when it is evaluated. *Numexpr* is an abbreviation for numeric expression. The words *left, top, bottom,* and *right* refer to numeric expressions unless the syntax description specifically says they are numeric variables. The word *element* means a numeric expression that evaluates to the index of an array element. The word *channel* means a numeric expression that evaluates to the number of a communications channel.

The word *statement* means a single program statement, and the word *statements* means one or more program statements.

Comments, labels, function names, subroutine names, and toolbox names are part of your program statements. You cannot use variables or expressions for them. On the other hand, you can use string literals, string variables, or any legal string expressions for file names, device names, and volume names.

MACINTOSH BASIC STATEMENTS

The following is an alphabetic list of Macintosh BASIC statements.

ASK CURPOS # channel, numvar

ASK DOCUMENT numvarLeft, numvarBottom; numvarRight, numvarTop

ASK ENVIRONMENT numvar

ASK EOF # channel, numvar

ASK EXCEPTION {numexpr | **INVALID** | **UNDERFLOW** | **OVERFLOW** | **DIVBYZERO** | **INEXACT**} BooleanVariable

ASK FONT numvar

ASK FONTSIZE numvar

ASK GTEXTFACE numvar

ASK GTEXTMODE numvar

ASK HALT {numexpr | **INVALID** | **UNDERFLOW** | **OVERFLOW** | **DIVBYZERO** | **INEXACT**} BooleanVariable

ASK HPOS numvar

ASK HPOS # channel, numvar

ASK LOCATION numvarLeft, numvarBottom; numvarRight, numvarTop

ASK OUTPUT numvarLeft, numvarBottom; numvarRight, numvarTop

ASK PATTERN numvar

ASK PEN numvar1, numvar2

ASK PENMODE numvar

ASK PENPOS numvar1, numvar2

ASK PENSIZE numvar1, numvar2

ASK PICSIZE numvar

ASK PRECISION numvar

ASK ROUND numvar

ASK SCALE numvarLeft, numvarBottom; numvarRight, numvarTop

ASK SHOWDIGITS numvar

ASK TABWIDTH numvar

ASK VPOS numvar

BTNWAIT

CALL subroutineName [({expression | variable} [, {expression | variable} ...])]

CLEARWINDOW

CLOSE [# channel]

CREATE # channel: filename [, {**APPEND** | **OUTIN**}] [, {**BINY** | **DATA** | **TEXT**}] [, {**RECSIZE** numexpr | **SEQUENTIAL** | **STREAM**}]

DATA literal[, literal ...]

DEF functionName [(variable [, variable ...])]

DELETE filename

DEVCONTROL # channel: @ integerArray (element)

DEVSTATUS # channel: @ integerArray (element)

DIM array (numexpr1 [, numexpr2 ...]) [, array (numexpr3 [, numexpr4 ...]) ...]

DO / [statements /] **LOOP**

DOCUMENT PRINT

EJECT {numexpr | volumename}

END [{**FUNCTION** | **MAIN** | **PROGRAM** | **SELECT** | **SUB** | **WHEN**}]

ERASE { {**OVAL** | **RECT**} left,top; right,bottom | **ROUNDRECT** left,top; right,bottom **WITH** numexpr1,numexpr2}

EXIT [{**DO** | **FOR** | **FUNCTION** | **PROGRAM** | **SUB**}]

FOR numvar = numexpr1 **TO** numexpr2 [**STEP** numexpr3] / [statements /] **NEXT** numvar

FRAME { {**OVAL** | **RECT**} left,top; right,bottom | **ROUNDRECT** left,top; right,bottom **WITH** numexpr1,numexpr2}

FUNCTION functionName [(variable [, variable ...])] / [statements /] functionName = expression / [statements /] **END FUNCTION**

GETFILEINFO filename, @ integerArray (element)

GETVOLINFO volumename, @ integerArray (element)

GOSUB label[:]

GOTO label[:]

GPRINT [**AT** numexpr1,numexpr2;] [expression [{; | ,} [expression] ...]]

GTEXTNORMAL

IF BooleanExpression **THEN** statement [**ELSE** statement]

IF BooleanExpression **THEN** / statements / [**ELSE** / statements /] **ENDIF**

IGNORE WHEN { **ERR** | **KBD** | **MENU** handle | **WINDOW** pointer }

INPUT [stringLiteral {; | ,}] variable [, variable . . .]

INPUT # channel [, {**BEGIN** | **NEXT** | **RECORD** numexpr | **SAME**}] [, {**IF EOF~** | **IF EOR~** | **IF MISSING~**} **THEN** statement]: variable [, variable . . .]

INVERT { {**OVAL** | **RECT**} left,top; right,bottom | **ROUNDRECT** left,top; right,bottom **WITH** numexpr1,numexpr2}

[**LET**] variable = expression

LINE INPUT [stringLiteral {; | ,}] variable

LINE INPUT # channel [, {**BEGIN** | **END** | **NEXT** | **RECORD** numexpr | **SAME**}] [, {**IF EOF~** | **IF MISSING~** } **THEN** statement]: variable

LOCK filename

NEXT numvar

OPEN # channel: {filename | deviceName} [, {**APPEND** | **INPUT** | **OUTIN**}] [, {**BINY** | **DATA** | **TEXT**}] [, {**RECSIZE** numexpr | **SEQUENTIAL** | **STREAM**}]

OPTION COLLATE {**NATIVE** | **STANDARD**}

PAINT { {**OVAL** | **RECT**} left,top; right,bottom | **ROUNDRECT** left,top; right,bottom **WITH** numexpr1,numexpr2}

PENNORMAL

PERFORM progname[({expression | @variable} [, {expression | @variable} . . .])]

PERFORM handle [({expression | @variable} [, {expression | @variable} . . .])]

PLOT [numexpr1, numexpr2 [; [numexpr3, numexpr4] . . .]]

POP

PRINT [expression [{; | ,} [expression] . . .]]

PRINT # channel [, {**BEGIN** | **END** | **NEXT** | **RECORD** numexpr | **SAME**}] : [expression [{; | ,} [expression] . . .]]

PROCENTRY numvar

PROCEXIT numexpr

PROGRAM programName [([@] variable [, [@]variable ...])]

RANDOMIZE

READ variable [, variable ...]

READ # channel [, {**BEGIN** | **END** | **NEXT** | **RECORD** numexpr | **SAME**}] [, {**IF EOF~** | **IF EOR~** | **IF MISSING~**} **THEN** statement]: variable [, variable ...]

REM comments

RENAME filename1, filename2

RESTORE [label[:]]

RETURN

REWRITE # channel [, {**BEGIN** | **END** | **NEXT** | **RECORD** numexpr | **SAME**}] [, {**IF EOF~** | **IF MISSING~**} **THEN** statement]: [expression [{; | ,} [expression] ...]]

SELECT [**CASE**] expression / [**CASE** {[[**IS**] relationalOperator] literal | literal **TO** literal} [, {[[**IS**] relationalOperator] literal | literal **TO** literal} ...] / statements /] [**CASE ELSE** / statements /] **END SELECT**

SET CURPOS # channel, numexpr

SET DOCUMENT {**TOWINDOW** | left, bottom; right, top}

SET ENVIRONMENT numexpr

SET EOF # channel, numexpr

SET EXCEPTION {numexpr | **INVALID** | **UNDERFLOW** | **OVERFLOW** | **DIVBYZERO** | **INEXACT**} BooleanExpression

SET FONT numexpr

SET FONTSIZE numexpr

SET GTEXTFACE numexpr

SET GTEXTMODE numexpr

SET HALT {numexpr | **INVALID** | **UNDERFLOW** | **OVERFLOW** | **DIVBYZERO** | **INEXACT**} BooleanExpression

SET HPOS numexpr

SET HPOS # channel, numexpr

SET LOCATION left, bottom; right, top

SET OUTPUT {**TOSCREEN** | left, bottom; right, top}

SET PATTERN {numexpr | **BLACK** | **DKGRAY** | **GRAY** | **LTGRAY** | **WHITE**}

SET PEN numexpr1, numexpr2

SET PENMODE numexpr

SET PENPOS numexpr1, numexpr2

SET PENSIZE numexpr1, numexpr2

SET PICSIZE numexpr

SET PRECISION {numexpr | **EXTPRECISION** | **DBLPRECISION** | **SGLPRECISION**}

SET ROUND {numexpr | **DOWNWARD** | **TONEAREST** | **TOWARDZERO** | **UPWARD**}

SET SCALE left, bottom; right, top

SET SHOWDIGITS numexpr

SET TABWIDTH numexpr

SET VPOS numexpr

SETFILEINFO filename, @ integerArray (element)

SETVOL {numexpr | volumename}

SOUND numexpr1,numexpr2,numexpr3 [; numexpr1,numexpr2,numexpr3 . . .]

SOUND numexpr @integerArray (element) [; numexpr @integerArray (element) . . .]

STOP

STOPSOUND

SUB subroutineName [(variable [,variable . . .])] / **END SUB**

TOOLBOX toolboxSubroutineNameWithParameters

UNDIM array ([, . . .]) [, array([, . . .]) . . .]

UNLOCK filename

WHEN ERR / statements / **END WHEN**

WHEN KBD / statements / **END WHEN**

WHEN MENU handle / statements / **END WHEN**

WHEN WINDOW pointer / statements / **END WHEN**

WRITE # channel [, {**BEGIN** | **END** | **NEXT** | **RECORD** numexpr | **SAME**}] [, **IF THERE~ THEN** statement]: [expression [{; | ,} [expression] . . .]]

MACINTOSH BASIC FUNCTIONS

The following is a list of functions available in Macintosh BASIC:

ABS (numexpr)

ADDRESS] ({ handle | pointer | stringexpr | numexpr })

ANNUITY (numexpr1, numexpr2)

ASC (stringExpression)

ATEOF~ (# channel)

ATN (numexpr)

CHR$ (numexpr)

CLASSCOMP (numexpr)

CLASSDOUBLE (numexpr)

CLASSEXTENDED (numexpr)

CLASSSINGLE (numexpr)

COMPOUND (numexpr1, numexpr2)

COPYSIGN (numexpr1, numexpr2)

COS (numexpr)

DATE$

DOWNSHIFT$ (stringExpression)

ERR

EXP (numexpr)

EXP2 (numexpr)

EXPM1 (numexpr)

FORMAT$ (stringExpression; expression [, expression . . .])

FREE

GETFILENAME$ (numexpr)

GETVOLNAME$ (numexpr)

HIGHWORD ({pointer | numexpr})

INDIRECT] ({handle | pointer | stringexpr})

INKEY$

INT (numexpr)

KBD

LEFT$ (stringExpression, numexpr)

LEN (stringExpression)

LOG (numexpr)

LOG2 (numexpr)

LOGB (numexpr)

LOGP1 (numexpr)

LOWWORD ({pointer | numexpr})

MENU}

MENUID

MENUITEM

MID$ (stringExpression, numexpr [, numexpr2])

MOUSEB

MOUSEB~

MOUSEH

MOUSEV

NEXTDOUBLE (numexpr1, numexpr2)

NEXTEXTENDED (numexpr1, numexpr2)

NEXTSINGLE (numexpr1, numexpr2)

OUTPUTWINDOW]

PI

RANDOMX (numvar)

RELATION (numexpr1, numexpr2)

REMAINDER (numexpr1, numexpr2)

RIGHT$ (stringExpression, numexpr)

RINT (numexpr)

RND [(numexpr)]

SCALB (numexpr1, numexpr2)

SGN ({numexpr | BooleanExpression})

SIGNNUM (numexpr)

SIN (numexpr)

SOUNDOVER~

SQR (numexpr)

STR$ (numexpr)

TAB (numexpr)

TAN (numexpr)

TICKCOUNT

TIME$

TONES (numexpr1)

TOOL toolboxFunctionNameWithParameters

TRUNC (numexpr)

TYP (# channel)

UPSHIFT$ (stringExpression)

VAL (stringExpression)

VALPOINTER ({pointer | numexpr})

Appendix B
Error Messages and Codes

There are three different kinds of error messages you may encounter while you are working in Macintosh BASIC. The first kind of message occurs just after you have entered a new program line. If BASIC detects an error in the line, it displays a message to help you correct the error. Those messages are self-explanatory and are not listed in this appendix.

The second type of error message you may encounter is a Macintosh BASIC run-time error message. If it encounters a condition it cannot handle, BASIC displays one of these messages while it is executing your program.

The final type of message you may encounter is a Macintosh system error. When one of these occurs, all you receive is a number. The last part of this appendix explains the meaning of the most common system error numbers.

MACINTOSH BASIC RUN-TIME ERROR MESSAGES

You can trap run-time errors using the WHEN ERR statement described in Chapter 15. The system function ERR returns a numeric code corresponding to the appropriate message when a run-time error has occurred. The run-time messages and codes follow.

98	File not open for output
99	Bad master directory block
100	File system error
101	External File System Error
102	Not a MAC diskette
103	No such drive
104	Volume already on-line
105	File is locked
106	Volume not on-line
107	(no message)
108	Refnum error
109	Error in user parameter list
110	File already open
111	Filename already exists
112	Can't delete an open file
113	Volume is locked
114	File is locked
115	Disk is write protected
116	File not found
117	Too many files open
118	Memory full
119	Tried to position to before start of file (r/w)
120	End of file
121	File not open
122	Bad file name
123	Disk I/O error
124	No such volume
125	Disk full
126	Directory full
127	Channel already exists
128	Channel not in range 0..32767
129	Array is too small
130	File is not a **Data** file

131 No such channel
132 Index must be >0
133 **Record** only works with **Relative** files
134 **Stream** files may not be positioned
135 Data exceeds record length
136 Not enough values in record
137 **Rewrite** must be used to write an existing record
138 Record is empty
139 Channel 0 implies a **Text** file
140 Last output was not finished
141 Index must be >= 0
142 This call does not work with channel 0
143 Print/Input are used with **Text** files
144 This call doesn't work with a **Stream** file
145 I/O System Error
146 Driver I/O Error
147 Read/Write are used with **Data** or **Biny** files
148 **Recsize** must be >0
149 Position exceeds record length
154 Undefined label, **Function**, or **Sub**
155 Illegal quantity
156 Syntax error
157 Undimensioned array reference
158 Dimension too big
159 Negative subscripts not allowed
160 Subscript out-of-bounds
161 Type mismatch
162 **Next** without **For**
163 **Loop** without **Do**
164 **Endif** not found
165 **Loop** not found
166 Integer overflow
167 **Return** without **Gosub**
168 Not enough memory for that operation
169 Deleting text only allowed on the line containing the **Input** prompt
170 Parameters don't match
171 Missing **End Select** statement
172 Couldn't find a **Case** that matched
173 Missing **End When** statement

174 Couldn't find matching **When**
175 Too many arguments for this toolbox routine
176 Not enough arguments for this toolbox routine
177 Something wrong with arguments calling this toolbox routine
178 **For** without **Next**
179 Already a **Dim** for this array
180 Can't assign string to this type of variable
181 Not enough values for **Input** list
182 Expected a number
183 Expected a Boolean
184 Too many values for **Input** list
185 Out of **DATA** to **READ**
186 You must move the insertion point back to the **Input** prompt line
187 Floating point halt
188 **End Sub** or **End Function** missing
189 Can't use parenthesis when assigning to a function
190 Bad range in **Set Location**
191 Low on memory. Please close some windows or Quit.
192 Bad range in **Set Output**
193 Can't assign function result here
194 Cannot run without listing

MACINTOSH SYSTEM ERRORS

There is almost never anything you can do to correct a Macintosh system error. When you receive a message reporting a system error, you should save a copy of your program and exit from BASIC. The condition that caused the system error may have damaged the copy of Macintosh BASIC or the Finder in your machine's memory. To avoid complications, you should restart by booting from a start-up disk. Even though there is rarely anything you can do to recover from these errors, explanations of the most common system error codes are included here to satisfy your curiosity. Some software errors can result in the display of incorrect system error ID numbers.

01 Bus error—an internal communication error in the machine. This should never happen on a Macintosh, but might occur rarely when running MacWorks on a Lisa.

02 Illegal address — the central processor chip received a word or long-word reference to an odd-numbered address, which is not permitted.

03 Illegal instruction — the central processor chip received an instruction it did not recognize.

04 Division by zero — the central processor chip executed a divide instruction (DIVS or DIVU) with a divisor of zero.

05 Check trap — the central processor chip executed a "check register against bounds" (CHK) instruction that found an improper value.

06 Overflow trap — the central processor chip executed a "trap on overflow" (TRAPV) instruction that detected numerical overflow (a number too large).

07 Privilege violation — the central processor chip executed a "return from exception" (RTE) or other inappropriate instruction.

08 Trace mode — the trace bit in the central processor chip's status register is set.

09 Line 1010 trap — the 1010 trap dispatcher portion of the system software used for Macintosh toolbox calls has been damaged.

10 Line 1111 trap — usually means that a breakpoint left by a systems programmer was reached.

11 Miscellaneous hardware exception — the central processor chip detected an error condition not covered by error numbers 01 through 10.

12 Unimplemented core routine — a trap number was encountered that does not correspond to a toolbox routine.

13 Uninstalled interrupt — an interrupt vector table entry is 0 when it should not be.

14 IO core — an error occurred while processing input or output.

15 Segment loader error — a call to GetResource to read a segment of BASIC into memory failed (this usually means you are out of room in memory).

16 Floating point error—the halt bit on a floating point environment word was set.

17 Package 0 not present—the resource PACK 0 is not on the disk or there is not enough room for it in memory.

18 Package 1 not present—the resource PACK 1 is not on the disk or there is not enough room for it in memory.

19 Package 2 not present—the resource PACK 2 is not on the disk or there is not enough room for it in memory.

20 Package 3 not present—the resource PACK 3 is not on the disk or there is not enough room for it in memory.

21 Package 4 not present—the resource PACK 4 is not on the disk or there is not enough room for it in memory.

22 Package 5 not present—the resource PACK 5 is not on the disk or there is not enough room for it in memory.

23 Package 6 not present—the resource PACK 6 is not on the disk or there is not enough room for it in memory.

24 Package 7 not present—the resource PACK 7 is not on the disk or there is not enough room for it in memory.

25 Out of memory—there is no room for any more program instructions or data in your machine's random-access memory.

26 Can't launch file—a call to GetResource to read segment 0 into memory failed. The file probably does not contain an executable program.

27 File system map has been damaged—the file system map contains a logical block number that is too big or too small to be a correct logical block on this volume.

28 Stack has moved into application heap—the two storage areas used by BASIC in its internal operations have collided (out of memory).

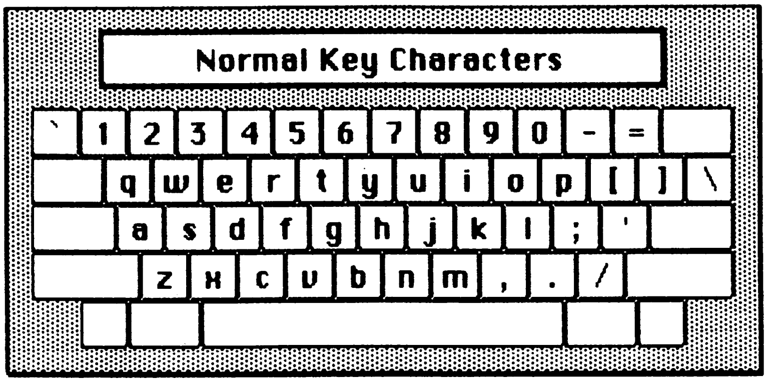
Appendix C

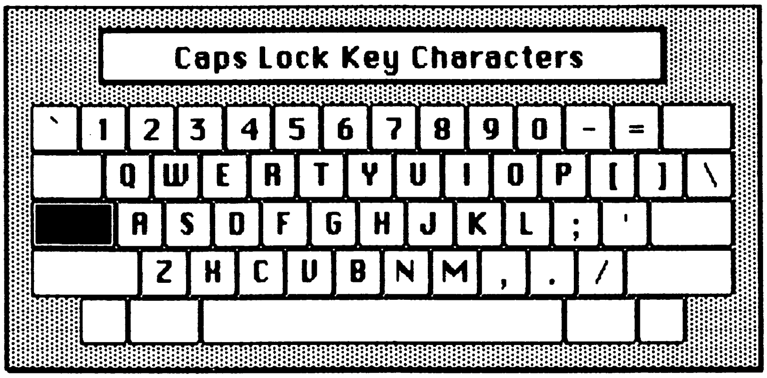
ASCII Codes And Keyboard Characters

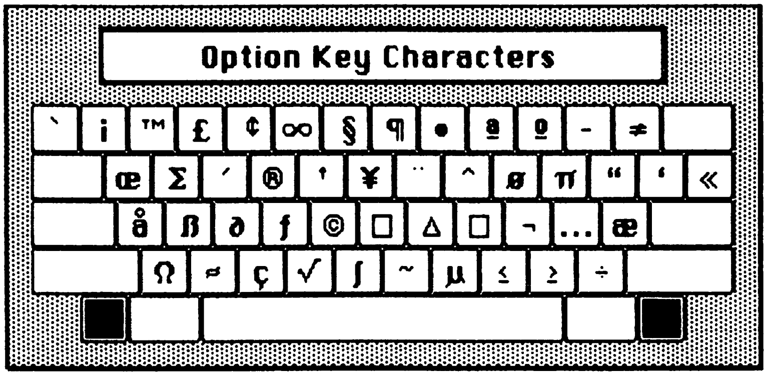
This appendix shows the character that is printed for each ASCII code and the arrangement of the characters on the Macintosh keyboard. The characters are shown in the Chicago 12 font, which has the most complete character set. Some of the more unusual characters may not be present in every font and every font size. One character, ASCII 217, is different in every font and size. You can type this character from the keyboard with the combination OPTION-SHIFT-~.

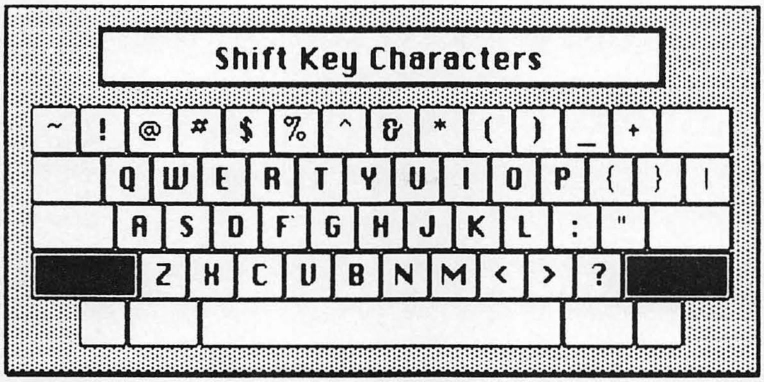
000		032		064	@	096	`
001	□	033	!	065	A	097	a
002	□	034	"	066	B	098	b
003	□	035	#	067	C	099	c
004	□	036	$	068	D	100	d
005	□	037	%	069	E	101	e
006	□	038	&	070	F	102	f
007	□	039	'	071	G	103	g
008	□	040	(	072	H	104	h
009		041	)	073	I	105	i
010	□	042	*	074	J	106	j
011	□	043	+	075	K	107	k
012	□	044	,	076	L	108	l
013		045	-	077	M	109	m
014	□	046	.	078	N	110	n
015	□	047	/	079	O	111	o
016	□	048	0	080	P	112	p
017	⌘	049	1	081	Q	113	q
018	✓	050	2	082	R	114	r
019	◆	051	3	083	S	115	s
020		052	4	084	T	116	t
021	□	053	5	085	U	117	u
022	□	054	6	086	V	118	v
023	□	055	7	087	W	119	w
024	□	056	8	088	X	120	x
025	□	057	9	089	Y	121	y
026	□	058	:	090	Z	122	z
027	□	059	;	091	[	123	{
028	□	060	<	092	\	124	|
029	□	061	=	093	]	125	}
030	□	062	>	094	^	126	~
031	□	063	?	095	_	127	

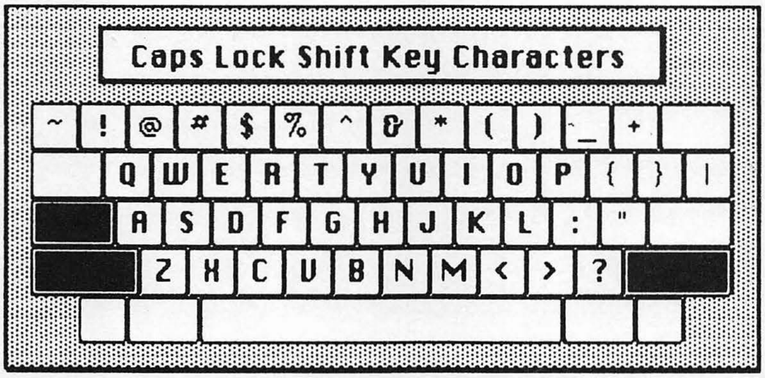
128	Ä	160	†	192	¿	224	□
129	Å	161	°	193	¡	225	□
130	Ç	162	¢	194	¬	226	□
131	É	163	£	195	√	227	□
132	Ñ	164	§	196	ƒ	228	□
133	Ö	165	•	197	≈	229	□
134	Ü	166	¶	198	∆	230	□
135	á	167	ß	199	«	231	□
136	à	168	®	200	»	232	□
137	â	169	©	201	…	233	□
138	ä	170	™	202		234	□
139	ã	171	´	203	À	235	□
140	å	172	¨	204	Ã	236	□
141	ç	173	≠	205	Õ	237	□
142	é	174	Æ	206	Œ	238	□
143	è	175	Ø	207	œ	239	□
144	ê	176	∞	208	–	240	□
145	ë	177	±	209	—	241	□
146	í	178	≤	210	“	242	□
147	ì	179	≥	211	”	243	□
148	î	180	¥	212	‘	244	□
149	ï	181	µ	213	’	245	□
150	ñ	182	∂	214	÷	246	□
151	ó	183	Σ	215	◊	247	□
152	ò	184	Π	216	ÿ	248	□
153	ô	185	π	217	□	249	□
154	ö	186	∫	218	□	250	□
155	õ	187	ª	219	□	251	□
156	ú	188	º	220	□	252	□
157	ù	189	Ω	221	□	253	□
158	û	190	æ	222	□	254	□
159	ü	191	ø	223	□	255	□

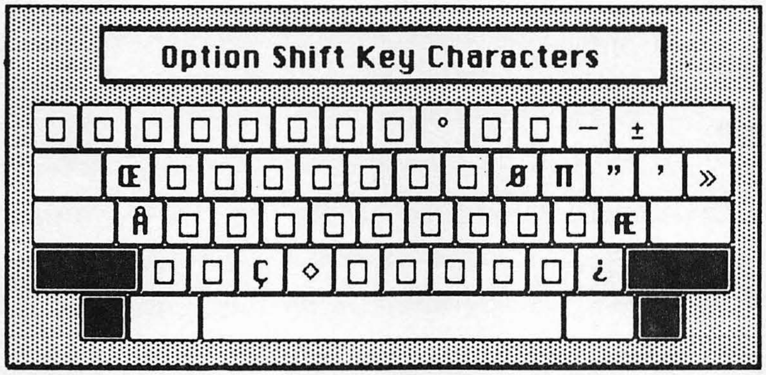

Normal Key Characters
` 1 2 3 4 5 6 7 8 9 0 - =
q w e r t y u i o p [ ] \
a s d f g h j k l ; '
z x c v b n m , . /


Caps Lock Key Characters
` 1 2 3 4 5 6 7 8 9 0 - =
Q W E R T Y U I O P [ ] \
A S D F G H J K L ; '
Z X C V B N M , . /


Option Key Characters
` ¡ ™ £ ¢ ∞ § ¶ • ª º – ≠
œ Σ ´ ® † ¥ ¨ ˆ ø π " ' «
å ß ∂ ƒ © □ Δ □ ¬ … æ
Ω ≈ ç √ ∫ ~ µ ≤ ≥ ÷


Shift Key Characters
~ ! @ # $ % ^ & * ( ) _ +
Q W E R T Y U I O P { } |
A S D F G H J K L : "
Z X C V B N M < > ?


Caps Lock Shift Key Characters
~ ! @ # $ % ^ & * ( ) _ +
Q W E R T Y U I O P { } |
A S D F G H J K L : "
Z X C V B N M < > ?


Option Shift Key Characters
□ □ □ □ □ □ □ □ ° □ □ – ±
Œ □ □ □ □ □ □ □ Ø ∏ ” ’ »
Å □ □ □ □ □ □ □ □ □ Æ
□ □ Ç ◊ □ □ □ □ □ ¿

Appendix D
Toolbox Routines Accessible From BASIC

Macintosh BASIC allows you to call a large number of the toolbox routines built into the machine's read-only memory. This appendix shows sample calls to the routines BASIC recognizes. Only the routines preceded by an asterisk (*) are described in this book.

Experienced programmers should refer to Apple Computer's *Inside Macintosh* for descriptions of all of the toolbox routines. The general procedures for calling toolbox routines are described in Chapter 19 of this book. The dimensions of the data types used in the example calls are appName©(31), bitMap©(13), cursor%(33), dateTimeRec%(6), eventRecord%(7), finalTicks%(1), FontInfo%(3), GrafPort%(52), offset%(1), point%(1), pattern©(7), penState©(17), rect%(3), reply©(73), ResType©(3), seconds%(1), string©(255).

QUICKDRAW

GrafPort Routines

ToolBox OpenPort (GrafPort])

ToolBox InitPort (GrafPort])

ToolBox ClosePort (GrafPort])

* **ToolBox SetPort** (GrafPort])

* **ToolBox GetPort** (@GrafPort%(0))

ToolBox GrafDevice (device%)

ToolBox SetPortBits (@bitMap©(0))

ToolBox PortSize (width%, height%)

ToolBox MovePortTo (leftGlobal%, topGlobal%)

ToolBox SetOrigin (h%, v%)

ToolBox SetClip (Region})

ToolBox GetClip (Region})

* **ToolBox ClipRect** (@rect%(0))

* **ToolBox BackPat** (@pattern©(0))

Cursor Handling

ToolBox InitCursor

ToolBox SetCursor (@cursor%(0))

ToolBox HideCursor

ToolBox ShowCursor

ToolBox ObscureCursor

Pen and Line Drawing

ToolBox HidePen

ToolBox ShowPen

ToolBox GetPenState (@penState©(0))

ToolBox SetPenState (@penState©(0))

* **ToolBox PenPat** (@pattern©(0))

ToolBox MoveTo (h%, v%)

ToolBox Move (dh%, dv%)

ToolBox LineTo (h%, v%)

ToolBox Line (dh%, dv%)

Text Drawing

ToolBox TextFont (fontnum%)

ToolBox TextFace (style%)

ToolBox TextMode (mode%)

ToolBox TextSize (fontsize%)

ToolBox SpaceExtra (extra%)

ToolBox DrawChar (char©)

ToolBox DrawString (string$)

ToolBox DrawText (@text©(0), firstByte%, byteCount%)

width% = **Tool CharWidth** (char©)

* width% = **Tool StringWidth** (string$)

width% = **Tool TextWidth** (@text©(0), firstByte%, byteCount%)

ToolBox GetFontInfo (@FontInfo%(0))

Drawing in Color

* **ToolBox ForeColor** (color%, 0)

* **ToolBox BackColor** (color%, 0)

ToolBox ColorBit (whichBit%)

Calculations With Rectangles

ToolBox SetRect (@rect%(0), left%, top%, right%, bottom%)

ToolBox OffsetRect (@rect%(0), dh%, dv%)

ToolBox InsetRect (@rect%(0), dh%, dv%)

ans~ = **Tool SectRect** (@sourceRectA%(0), @sourceRectB%(0), @destRect%(0))

ToolBox UnionRect (@sourceRectA%(0), @sourceRectB%(0), @destRect%(0))

ans~ = **Tool PtInRect** (pt.h%, pt.v%, @rect%(0))

ToolBox Pt2Rect (ptA.h%, ptA.v%, ptB.h%, ptB.v%, @destRect%(0))

ToolBox PtToAngle (@rect%(0), pt.h%, pt.v%, @angle%)

ans~ = **Tool EqualRect** (@rectA%(0), @rectB%(0))

ans~ = **Tool EmptyRect** (@rect%(0))

Graphics Operations on Shapes

* **ToolBox FillRect** (@rect%(0), @pattern©(0))
* **ToolBox FillOval** (@rect%(0), @pattern©(0))
* **ToolBox FillRoundRect** (@rect%(0), ovalWidth%, ovalHeight%, @pattern©(0))
* **ToolBox FrameArc** (@rect%(0), startAngle%, arcAngle%)
* **ToolBox PaintArc** (@rect%(0), startAngle%, arcAngle%)
* **ToolBox EraseArc** (@rect%(0), startAngle%, arcAngle%)
* **ToolBox InvertArc** (@rect%(0), startAngle%, arcAngle%)
* **ToolBox FillArc** (@rect%(0), startAngle%, arcAngle%, @pattern©(0))

Calculations With Regions

Region} = **Tool NewRgn**

ToolBox DisposeRgn (Region})

ToolBox CopyRgn (sourceRegion}, destRegion})

ToolBox SetEmptyRgn (Region})

ToolBox SetRectRgn (Region}, left%, top%,right%, bottom%)

ToolBox RectRgn (Region}, @rect%(0))

ToolBox OpenRgn

ToolBox CloseRgn (Region})

ToolBox OffsetRgn (Region}, dh%, dv%)

ToolBox InsetRgn (Region}, dh%, dv%)

ToolBox SectRgn (sourceRegionA}, sourceRegionB}, destRegion})

ToolBox UnionRgn (sourceRegionA}, sourceRegionB}, destRegion})

ToolBox DiffRgn (sourceRegionA}, sourceRegionB}, destRegion})

ToolBox XORRgn (sourceRegionA}, sourceRegionB}, destRegion})

ans~ = **Tool PtInRgn** (pt.h%, pt.v%, Region})

ans~ = **Tool RectInRgn** (@rect%(0), Region})

ans~ = **Tool EqualRgn** (Region1}, Region2})

ans~ = **Tool EmptyRgn** (Region})

Graphics Operations on Regions

ToolBox FrameRgn (Region})

ToolBox PaintRgn (Region})

ToolBox EraseRgn (Region})

ToolBox InvertRgn (Region})

ToolBox FillRgn (Region}, @pattern©(0))

Bit Transfer Operations

ToolBox ScrollRect (@rect%(0), dh%, dv%, updateRegion})

ToolBox CopyBits (@srcBitmap©(0), @destBitmap©(0), @srcRect%(0), @destRect%(0), mode%, maskRgn})

Pictures

* Picture} = **Tool OpenPicture** (@picframeRect%(0))

ToolBox PicComment (kind%, dataSize%, Data})

* **ToolBox ClosePicture**

* **ToolBox DrawPicture** (Picture}, @destRect%(0))

* **ToolBox KillPicture** (Picture})

Calculations With Polygons

Polygon} = **Tool OpenPoly**

ToolBox ClosePoly

ToolBox KillPoly (Polygon})

ToolBox OffsetPoly (Polygon}, dh%, dv%)

Graphics Operations on Polygons

ToolBox FramePoly (Polygon})

ToolBox PaintPoly (Polygon})

ToolBox ErasePoly (Polygon})

ToolBox InvertPoly (Polygon})

ToolBox FillPoly (Polygon}, @pattern©(0))

Calculations With Points

ToolBox AddPt (sourcePt.h%, sourcePt.v%, @destPoint%(0))

ToolBox SubPt (sourcePt.h%, sourcePt.v%, @destPoint%(0))

ToolBox SetPt (@point%(0), h%, v%)

ans~ = **Tool EqualPt** (ptA.h%, ptA.v%, ptB.h%, ptB.v%)

ToolBox LocalToGlobal (@point%(0))

ToolBox GlobalToLocal (@point%(0))

Miscellaneous Utilities

num% = **Tool Random**

* ans~ = **Tool GetPixel (h%, v%)**

ToolBox StuffHex (@array©(0), hexstring$)

ToolBox ScalePt (@point%(0), @sourceRect%(0), @destRect%(0))

ToolBox MapPt (@point%(0), @sourceRect%(0), @destRect%(0))

ToolBox MapRect (@rRect%(0), @sourceRect%(0), @destRect%(0))

ToolBox MapRgn (Region}, @sourceRect%(0), @destRect%(0))

ToolBox MapPoly (Polygon}, @sourceRect%(0), @destRect%(0))

RESOURCE MANAGER

Opening and Closing Resource Files

ToolBox CreateResFile (filename$)

* refnum% = **Tool OpenResFile** (filename$)

ToolBox CloseResFile (refnum%)

Checking for Errors

num% = **Tool ResError**

Setting the Current Resource File

refnum% = **Tool CurResFile**

refnum% = **Tool HomeResFile**

* **ToolBox UseResFile** (refnum%)

Getting Resource Types

* num% = **Tool CountTypes**

* **ToolBox GetIndType** (@ResType©(0), index%)

Getting and Disposing of Resources

ToolBox SetResLoad (load~)

* num% = **Tool CountResources** (ResTypeLast2%, ResTypeFirst2%)

* Resource} = **Tool GetIndResource** (ResTypeLast2%, ResTypeFirst2%, index%)

* Resource} = **Tool GetResource** (ResTypeLast2%, ResTypeFirst2%, ResID%)

* Resource} = **Tool GetNamedResource** (ResTypeLast2%, ResTypeFirst2%, name$)

ToolBox LoadResource (Resource})

* **ToolBox ReleaseResource** (Resource})

ToolBox DetachResource (Resource})

Getting Resource Information

ResID% = **Tool UniqueID** (ResTypeLast2%, ResTypeFirst2%)

* **ToolBox GetResInfo** (Resource}, @ResID%, @ResType©(0), @nameString©(0))

attrs% = **Tool GetResAttrs** (Resource})

Modifying Resources

ToolBox SetResInfo (Resource})

ToolBox SetResAttrs (Resource}, attrs%)

ToolBox ChangedResource (Resource})

ToolBox AddResource (Data}, ResTypeLast2%, ResTypeFirst2%, ResID%, name$)

ToolBox RmveResource (Resource})

ToolBox AddReference (Resource}, ResID%, name$)

ToolBox RmveReference (Resource})

ToolBox UpdateResFile (refnum%)

ToolBox WriteResource (Resource})

ToolBox SetResPurge (install~)

Advanced Routines

attrs% = **Tool GetResFileAttrs** (refnum%)

ToolBox SetResFileAttrs (refnum%, attrs%)

WINDOW MANAGER

Initialization and Allocation

* W] = **Tool NewWindow** (0, 0, @rect%(0), title$, visible~, procID%, −1, −1, goAway~, 0, 0)

ToolBox CloseWindow (W])

* W] = **Tool GetNewWindow** (ResID%, 0, 0, −1, −1)

* **ToolBox DisposeWindow** (W])

Window Display

* **ToolBox SetWTitle** (W], title$)

* **ToolBox GetWTitle** (W], @string©(0))

ToolBox SelectWindow (W])

ToolBox HideWindow (W])

ToolBox ShowWindow (W])

ToolBox HiliteWindow (W], Hilite~)

ToolBox BringToFront (W])

ToolBox SendBehind (W], behindW])

* W] = **Tool FrontWindow**

* **ToolBox DrawGrowIcon** (W])

Mouse Location

num% = **Tool FindWindow** (pt.h%, pt.v%, @W])

Window Movement and Sizing

* **ToolBox MoveWindow** (W], h%, v%, front~)
* **ToolBox SizeWindow** (W], width%, height%, fUpdate~)

Update Region Maintenance

ToolBox InvalRect (@badRect%(0))

ToolBox ValidRect (@goodRect%(0))

ToolBox InvalRgn (badRegion})

ToolBox ValidRgn (goodRegion})

Miscellaneous Utilities

ToolBox SetWRefCon (W], p])

p] = **Tool GetWRefCon** (W])

ToolBox SetWindowPic (W], Pic})

Pic} = **Tool GetWindowPic** (W])

longint] = **Tool PinRect** (@rect%(0), pt.h%, pt.v%)

MENU MANAGER

Initialization and Allocation

* M} = **Tool NewMenu** (newmenuResID%, title$)
* M} = **Tool GetMenu** (menuResID%)
* **ToolBox DisposeMenu** (M})
* **ToolBox AppendMenu** (M}, itemName$)

ToolBox AddResMenu (M}, ResTypeLast2%, ResTypeFirst2%)

ToolBox InsertResMenu (M}, ResTypeLast2%, ResTypeFirst2%, afterItem%)

Forming the Menu Bar

* **ToolBox InsertMenu** (M}, beforeID%)

* **ToolBox DrawMenuBar**

* **ToolBox DeleteMenu** (menuResID%)

ToolBox ClearMenuBar

MenuBar} = **Tool GetNewMBar** (menubarResID%)

MenuBar} = **Tool GetMenuBar**

ToolBox SetMenuBar (MenuBar})

Choosing From a Menu

longint] = **Tool MenuSelect** (pt.h%, pt.v%)

longint] = **Tool MenuKey** (char©)

* **ToolBox HiliteMenu** (menuResID%)

Controlling Appearance of Items

* **ToolBox SetItem** (M}, item%, string$)

* **ToolBox GetItem** (M}, item%, @string©(0))

* **ToolBox DisableItem** (M}, item%)

* **ToolBox EnableItem** (M}, item%)

* **ToolBox CheckItem** (M}, item%, checked~)

ToolBox SetItemIcon (M}, item%, iconResID%)

ToolBox GetItemIcon (M}, item%, @iconResID%)

* **ToolBox SetItemStyle** (M}, item%, style%)

* **ToolBox GetItemStyle** (M}, item%, @style%)

ToolBox SetItemMark (M}, item%, char©)

ToolBox GetItemMark (M}, item%, @char©)

Miscellaneous Utilities

ToolBox SetMenuFlash (M}, item%)

ToolBox CalcMenuSize (M})

num% = **Tool CountMItems** (M})

* M} = **Tool GetMHandle** (menuResID%)

ToolBox FlashMenuBar (menuResID%)

CONTROL MANAGER

Initialization and Allocation

* Control} = **Tool NewControl** (W], @rect%(0), title$, visible~, value%, min%, max%, procID%, 0, 0)

Control} = **Tool GetNewControl** (ctrlResID%, W])

* **ToolBox DisposeControl** (Control})

Control Display

* **ToolBox SetCTitle** (Control}, title$)

* **ToolBox GetCTitle** (Control}, @string©(0))

* **ToolBox HideControl** (Control})

* **ToolBox ShowControl** (Control})

ToolBox DrawControls (W])

ToolBox HiliteControl (Control})

Mouse Location

* num% = **Tool TestControl** (Control}, pt.h%, pt.v%)

num% = **Tool FindControl** (pt.h%, pt.v%, W], @Control})

num% = **Tool TrackControl** (Control}, startPt.h%, startPt.v%, 0, 0)

Control Movement and Sizing

* **ToolBox MoveControl** (Control}, hloc%, vloc%)

ToolBox DragControl (Control}, startPt.h%, startPt.v%, @limitRect%(0), @slopRect%(0), axis%)

* **ToolBox SizeControl** (Control}, width%, height%)

Control Setting and Range

* **ToolBox SetCtlValue** (Control}, value%)

* **ToolBox SetCtlMin** (Control}, minValue%)

* **ToolBox SetCtlMax** (Control}, maxValue%)

* value% = **Tool GetCtlValue** (Control})

* minValue% = **Tool GetCtlMin** (Control})

* maxValue% = **Tool GetCtlMax** (Control})

Miscellaneous Utilities

ToolBox SetCRefCon (Control}, p])

p] = **Tool GetCRefCon** (Control})

DIALOG MANAGER

* itemHit% = **Tool Alert** (alertResID%, 0, 0)

itemHit% = **Tool CautionAlert** (alertResID%, 0, 0)

ToolBox CloseDialog (D])

hit~ = **Tool DialogSelect** (@eventRecord%(0), @D], @itemHit%)

ToolBox DisposDialog (D])

ToolBox GetDItem (D],item%, @type%, @itemHdl}, @rect%(0))

ToolBox GetIText (itemHdl}, @string©(0))

D] = **Tool GetNewDialog** (dialogResID%, 0, 0, −1, −1)

ans~ = **Tool IsDialogEvent** (@eventRecord%(0))

ToolBox ModalDialog (0, 0, @itemHit%)

D] = **Tool NewDialog** (0, 0, @rect%(0), title$, visible~, procID%, −1, −1, goAway~, 0, 0, items})

itemHit% = **Tool NoteAlert** (alertResID%, 0, 0)

* **ToolBox ParamText** (string0$, string1$, string2$, string3$)

ToolBox SellText (D], item%, startSel%, endSel%)

ToolBox SetDItem (D], item%, type%, itemHdl}, @rect%(0))

ToolBox SetlText (itemHdl}, string$)

itemHit% = **Tool StopAlert** (alertResID%, 0, 0)

DESK MANAGER

ToolBox CloseDeskAcc (accRefnum%)

accRefnum% = **Tool OpenDeskAcc** (accName$)

ToolBox SystemClick (@eventRecord%(0), W])

done~ = **Tool SystemEdit** (editCmd%)

ToolBox SystemTask

EVENT MANAGER

down~ = **Tool Button**

avail~ = **Tool EventAvail** (eventMask%, @eventRecord%(0))

ToolBox FlushEvents (eventMask%, stopMask%)

ToolBox GetMouse (@point%(0))

myEvent~ = **Tool GetNextEvent** (eventMask%, @eventRecord%(0))

down~ = **Tool StillDown**

ticks] = **Tool TickCount**

down~ = **Tool WaitMouseUp**

FONT MANAGER

ToolBox GetFNum (fontName$, @fontNum%)

ToolBox GetFontName (fontNum%, @string©(0))

real~ = **Tool RealFont** (fontNum%, fontSize%)

MEMORY MANAGER

ToolBox BlockMove (source], dest], countlo%, counthi%)

ToolBox DisposHandle (handle})

ToolBox DisposPtr (pointer])

size] = **Tool GetHandleSize** (handle})

size] = **Tool GetPtrSize** (pointer])

ToolBox HLock (handle})

ToolBox HUnLock (handle})

ToolBox HPurge (handle})

ToolBox HNoPurge (handle})

ToolBox MoreMasters

hdl} = **Tool NewHandle** (sizeLo%, sizeHi%)

ptr] = **Tool NewPtr** (sizeLo%, sizeHi%)

ToolBox SetHandleSize (hdl}, sizeLo%, sizeHi%)

ToolBox SetPtrSize (ptr}, sizeLo%, sizeHi%)

OPERATING SYSTEM UTILITIES

ToolBox Delay (numTicksLo%, numTicksHi%, @finalTicks%(0))

errorNum% = **Tool ReadDateTime** (@seconds%(0))

* **ToolBox Secs2Date** (secsLo%, secsHi%, @dateTimeRec%(0))

ToolBox SysBeep (duration%)

ToolBox UprString (@string©(0), marks~)

ToolBox Date2Secs (@dateTimeRec%(0), @seconds%(0))

SCRAP MANAGER

longint] = **Tool GetScrap** (hDest}, resTypeLast2%, resTypeFirst2%, @offset%(0))

pScrapStuff] = **Tool InfoScrap**

longint] = **Tool LoadScrap**

longint] = **Tool UnloadScrap**

longint] = **Tool PutScrap** (lengthLo%, lengthHi%, resTypeLast2%, resTypeFirst2%, @text©(0))

longint] = **Tool ZeroScrap**

SEGMENT LOADER

ToolBox ExitToShell

ToolBox UnloadSeg (routineAddr])

ToolBox GetAppParms (@appName©(0), @apRefNum%, @apParam})

TEXT EDIT

ToolBox TEActivate (hTE})

ToolBox TECalText (hTE})

ToolBox TEClick (pt.h%, pt.v%, extend~, hTE})

ToolBox TECopy (hTE})

ToolBox TECut (hTE})

ToolBox TEDeactivate (hTE})

ToolBox TEDelete (hTE})

ToolBox TEDispose (hTE})

CharsHandle} = **Tool TEGetText** (hTE})

ToolBox TEIdle (hTE})

ToolBox TEInsert (@text©(0), lengthLo%, lengthHi%,hTE})

ToolBox TEKey (char©, hTE})

hTE} = **Tool TENew** (@destRect%(0), @viewRect%(0))

ToolBox TEPaste (hTE})

ToolBox TEScroll (dh%, dv%, hTE})

ToolBox TESetJust (j%, hTE})

ToolBox TESetSelect (selStartLo%, selStartHi%, selEndLo%, selEndHi%, hTE})

ToolBox TESetText (@text©(0), lengthLo%, lengthHi%, hTE})

ToolBox TEUpdate (@updateRect%(0), hTE})

ToolBox TextBox (@text)(0), lengthLo%, lengthHi%, @boxRect%(0), j%)

TOOLBOX UTILITIES

longint] = **Tool BitAnd** (long1Lo%, long1Hi%, long2Lo%, long2Hi%)

longint] = **Tool BitOr** (long1Lo%, long1Hi%, long2Lo%, long2Hi%)

longint] = **Tool BitXor** (long1Lo%, long1Hi%, long2Lo%, long2Hi%)

longint] = **Tool BitNot** (longintLo%, longintHi%)

longint] = **Tool BitShift** (longintLo%, longintHi%, count%)

set~ = **Tool BitTst** (@byte©, bitnumLo%, bitnumHi%)

ToolBox BitSet (@byte©, bitnumLo%, bitnumHi%)

ToolBox BitClr (@byte©, bitnumLo%, bitnumHi%)

fixed] = **Tool FixRatio** (numerator%, denominator%)

fixed] = **Tool FixMul** (aLo%, aHi%, bLo%, bHi%)

num% = **Tool FixRound** (xLo%, xHi%)

CursHandle} = **Tool GetCursor** (cursorResID%)

IconHandle} = **Tool GetIcon** (iconResID%)

PatHandle} = **Tool GetPattern** (patternResID%)

StringHandle} = **Tool GetString** (stringResID%)

longint] = **Tool Munger** (h}, offsetLo%, offsetHi%, @text1©(0), len1Lo%, len1Hi%, @text2©(0), len2Lo%, len2Hi%)

StringHandle} = **Tool NewString** (string$)

ToolBox PlotIcon (@rect%(0), IconHandle})

ToolBox SetString (StringHandle}, string$)

ToolBox PackBits (@srcPtr], @destPtr], srcBytes%)

ToolBox UnpackBits (@srcPtr], @destPtr], dstBytes%)

ROUTINES IN PACKAGES

PACK 3 (load PACK 3 first)

* SFPutFile - **PERFORM** h} (pt.h%, pt.v%, prompt$, origName$, 0, 0, @reply©(0), 1)
* SFGetFile - **PERFORM** h} (pt.h%, pt.v%, "", 0, 0, numTypes%, @types©(0), 0, 0, @reply©(0), 2)

PACK 6 (load PACK 6 first)

IUDateString - **PERFORM** h} (loword%, hiword%, form%, @string©(0), 0)

IUTimeString - **PERFORM** h} (loword%, hiword%, inclSeconds~, @string©(0), 2)

IUSetIntl - **PERFORM** h} (refnum%, ID%, data}, 8)

IUDatePString - **PERFORM** h} (loword%, hiword%, form%, @string©(0), parm}, 14)

IUTimePString - **PERFORM** h} (loword%, hiword%, seconds~, @string©(0), parm}, 16)

Appendix E

Solutions to Practice Exercises

This appendix presents solutions to the practice exercises that appear at the end of many chapters. When an exercise involves writing a program, only one solution will be shown, even though many different answers are possible.

Chapter 3

1. Numeric variables: c and f. String variables: b. Name a contains an illegal space, d contains an arithmetic operator (the minus sign), e contains the # sign, and g does not start with a letter. Numbers and periods are allowed in variable names as long as they are not the first character. b is the only string variable name because it is the only one that ends with a dollar sign.
2. Amount = 9, size = 4, number = 8, and rate = 27.

3.
```
! Greeter
INPUT "Please type your name: "; name$
PRINT "Well, hello, "; name$
END PROGRAM
```

4.
```
! Convert meters to inches
PRINT "METERS TO INCHES CONVERTER"
PRINT
INPUT 'How many meters? '; meters
inches = 39.37 * meters
PRINT
PRINT meters; " meters contain "; inches; " inches."
END PROGRAM
```

Chapter 4

The practice exercises in Chapter 4 use the editing, search, and replace commands. They do not lend themselves to written solutions.

Chapter 5

1. The program does not print anything. It goes into an infinite loop branching from a to c to b to a.

2.
```
v~ = 4 * 2 > 5 OR 3 ^ 2 + 3 ≤ 7 AND NOT (3 = 7 / 2)
v~ = 4 * 2 > 5 OR 6 + 3 ≤ 7 AND NOT (3 = 7 / 2)
v~ = 8 > 5 OR 6 + 3 ≤ 7 AND NOT (3 = 3.5)
v~ = 8 > 5 OR 6 + 3 ≤ 7 AND NOT FALSE
v~ = 8 > 5 OR 9 ≤ 7 AND TRUE
v~ = TRUE OR FALSE AND TRUE
v~ = TRUE OR FALSE
v~ = TRUE
```

3.
```
IF a=6 THEN
    IF b=g THEN
        x=8
    ELSE
```

```
        x=5
    ENDIF
ELSE
    x=1
ENDIF
```

4.
```
SELECT CASE i
    CASE 1
        b = 3
    CASE 2
        b = 5
    CASE < 0
        b = 0
    CASE 4, 6
        b = 7
    CASE ELSE
        b = 10
END SELECT
```

Chapter 6

1. The program will exit from the DO loop and print "Hello" when a number between 1 and 5 is typed.

2. The loop will be executed 11 times, for the following values of i: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100.

3.
```
FOR i = 3 TO 37 STEP 2
    PRINT i
NEXT i
```

4.
```
FOR count = 1 TO 10
    GOSUB Print.it
NEXT count
END PROGRAM
Print.it:
    PRINT "This is a test."
    RETURN
```

Chapter 7

1. (a) 5; (b) −12.

2. (a) 3; (b) 3; (c) 4; (d) −4; (e) −3; (f) −4.

3.
```
! Print payment table
FOR int.rate = 9 TO 12 STEP .5
    PRINT int.rate, 5000/ANNUITY(int.rate/100/12,4*12)
NEXT int.rate
END PROGRAM
```

4.
```
! Wait 90 ticks
tstart = TICKCOUNT
tend = tstart + 90
DO ! Wait for TICKCOUNT
    IF TICKCOUNT >= tend THEN EXIT DO
LOOP
```

5.
```
! Random subroutine
Makernd:
DO
    a = RND (6)  ! range of 6
    a = INT (a)   ! make integer
    IF a <> 6 THEN EXIT DO
LOOP
a = a - 10
RETURN
```

Chapter 8

1. (a) 'abcd'; (b) 'ananas'; (c) 'pol'; (d) 'anana split'.

2. (a) 'Politics'; (b) 'spectrum'.

3.
```
DO
    LINE INPUT a$
    a = VAL (a$)
    IF STR$(a) = a$ THEN EXIT DO
LOOP
```

4.
```
! Get keyboard character
Getchar:
    DO
        c$ = INKEY$
        IF c$ <> "" THEN EXIT DO
    LOOP
    RETURN
```

5.
```
! Alarm clock
PRINT "What time should I ring?"
INPUT "Please use Macintosh format: ";t$
    ! This is the right place in the
    ! program to check for correct input.
DO
    IF LEFT$(t$,5) = LEFT$(TIME$,5) THEN
        IF RIGHT$(TIME$,2) = RIGHT$(t$,2) THEN EXIT DO
    ENDIF
LOOP
PRINT "RING!"
END PROGRAM
```

Chapter 9

1. (a) 45 double-precision real numbers; (b) 9 Booleans; (c) 901 times 3 = 2703 extended-precision reals; (d) 100 short integers; (e) 30 strings.

2. When a$( ) is copied into q$( ), the dimension of q$ changes to 3.

3. All of them: (a) tries to put a number into a string variable; (b) tries to copy arrays with a different number of dimensions; (c) tries to put a number into a Boolean variable; (d) tries to put a string into a numeric variable (the last character of the name is what counts); and (e) tries to put a number larger than 32767 into a short integer variable.

4. The fourth element of the DATA statement is a string, but the READ statement tries to put it into a numeric variable.

Chapter 10

1.
```
SET VPOS 3  ! For PRINT command
SET PENPOS 7,44  ! For GPRINT with 12-point type
```

2.
```
ASK TABWIDTH oldtab%  ! Save old value
SET TABWIDTH 20
PRINT 1,2,3,4,5
SET TABWIDTH oldtab%  ! Restore old value
```

3.
```
SET FONT 2  ! New York
SET FONTSIZE 24 ! 24-point
SET GTEXTFACE 8 ! Outline type
SET PENPOS 7,50 ! Picks a spot
GPRINT "Hello"
```

4.
```
PRINT FORMAT$("$##,###|###.##";amount)
```

Chapter 11

1.
```
! Convert degrees Celsius to Fahrenheit
DO
    INPUT "Degrees Celsius: "; degrees
    PRINT "That's "; Fahrenheit%(degrees); " Fahrenheit."
LOOP
END PROGRAM
DEF Fahrenheit%(x) = 9 * x / 5 + 32
```

2.
```
! Find the smaller of two integers
FUNCTION Min%(a%,b%)
IF a% < b% THEN
        Min% = a%
    ELSE
        Min% = b%
    ENDIF
END FUNCTION
```

3.
```
FUNCTION Comma$(string$)
zstr$ = ""  ! Start with empty string
FOR zchar = 1 TO LEN(string$)
    zch$ = MID$(string$,zchar,1)
    IF zch$ = '.' THEN zch$ = ','
    zstr$ = zstr$ & zch$
NEXT zchar
Comma$ = zstr$
END FUNCTION
```

4.
```
FUNCTION Stars$(length%)
IF length% < 1 THEN
        Stars$ = ""
    ELSE
        Stars$ = Stars$(length% - 1) & '*'
ENDIF
END FUNCTION
```

Chapter 12

1.
```
OPEN #1: "gift list", APPEND
```

2.
```
CREATE #1: "newfile", RECSIZE 30, OUTIN
```

3.
```
READ #3, RECORD 23: integer%, string$
```

4.
```
DO
IF TYP(#12) = 2 THEN EXIT DO
ASK CURPOS #12, record
SET CURPOS #12, record+1    ! Try next record
LOOP
READ #12: string$
```

Chapter 13

1.
```
! Display List of Visible Files
DIM a%(23)    ! 48 bytes
count = 1
```

```
DO
    file$ = GetFileName$(count)
    IF file$ = "" THEN EXIT DO
    GETFILEINFO file$, @a%(0)
    IF (Num2(4) DIV 16384) MOD 2 <> 1 THEN
        ! If it's not 1, file is visible
        PRINT file$
        ENDIF
    count = count + 1
LOOP
END PROGRAM
FUNCTION Num2(first)
    IF a%(first) < 0 THEN
        Num2 = a%(first) + 65536
    ELSE Num2 = a%(first)
    ENDIF
END FUNCTION
```

2.
```
DIM a%(23)    ! 48 bytes
GETFILEINFO "System", @a%(0)
folder% = a%(7)    ! Save system folder number
GETFILEINFO "Macintosh BASIC", @a%(0)
a%(7) = folder%    ! Set to system folder
SETFILEINFO "Macintosh BASIC", @a%(0)
END PROGRAM
```

3.
```
! Set up 300 baud modem
DIM setup%(1)
OPEN #4: ".AIN", OUTIN
setup%(0) = 8  ! always 8
    ! 300 baud, 8 data bits, 1 stop bit
setup%(0) = 380 + 3072 + 16384
DEVCONTROL #4: @ setup%(0)
    ! Now you can use the modem here
CLOSE #4  ! Close before quitting
END PROGRAM
```

4.
```
! Print List of Files on Disk
OPEN #5: ".Printer", APPEND
count = 1
DO
```

```
        file$ = GetFileName$(count)
        IF file$ = "" THEN EXIT DO
        PRINT #5: file$
        count = count + 1
    LOOP
    CLOSE #5
    END PROGRAM
```

Chapter 15

1.
```
CALL Average( 3,9,answer )
PRINT answer
END PROGRAM
SUB Average( a,b,result)
result = (a+b)/2
END SUB
```

2.
```
PERFORM Average( 3,9,@answer )
PRINT answer
END PROGRAM

! Separate program in a separate file:
PROGRAM Average( a,b,@result)
result = (a+b)/2
END PROGRAM
```

3.
```
WHEN ERR
IF ERR = 102 THEN PRINT "Is that a Lisa diskette?"
END WHEN
```

4.
```
WHEN KBD
IF KBD = ASC( "?") THEN PRINT "See page 40 in the manual"
END WHEN
```

Chapter 16

1.
```
SET PENSIZE 5,5
PLOT 30,30; 90,12
```

2.
```
SET PATTERN 3    ! Gray
SET PENSIZE 6,6
FRAME OVAL 10,10; 80,200
```

3.
```
PAINT RECT 10,10; 200,200
ERASE OVAL 30,30; 180,180
! INVERT could be used instead of ERASE here.
```

4.
```
ERASE RECT 10,10; 200,90
```

Chapter 17

1.
```
DO
    IF MOUSEB~ THEN
        PLOT MOUSEH, MOUSEV
        DO ! Wait for button to come up
            IF NOT MOUSEB~ THEN EXIT DO
        LOOP
    ENDIF
LOOP
```

2.
```
DO
    IF MOUSEB~ THEN
        IF MOUSEH > 0 AND MOUSEV > 0 THEN
            PLOT MOUSEH, MOUSEV; ! Draw line
            DO ! Wait for button to come up
                IF NOT MOUSEB~ THEN EXIT DO
            LOOP
        ENDIF
    ENDIF
LOOP
```

3.
```
CALL WaitforUser
PRINT "Out of the subroutine."
END PROGRAM
SUB WaitforUser
DO
    IF LEN(INKEY$) > 0 OR MOUSEB~ THEN EXIT SUB
LOOP
END SUB
```

4.
```
DO
    IF MOUSEB~ THEN
        PLOT MOUSEH, MOUSEV;
    ELSE
        PLOT   ! Button is up, so end the line.
    ENDIF
LOOP
```

Chapter 18

1.
```
WHEN KBD
key = KBD  ! Get it once so it can't change
IF key<ASC('0') OR key>ASC('9') THEN SOUND
END WHEN
```

2.
```
SOUND TONES(7), 127, 30
```

3.
```
SOUND 9 @trips%(4)  ! triplet is 3 integers
```

4.
```
STOPSOUND     ! Stop all existing sound
SOUND 0,0,60 ! One second of silence
```

Chapter 19

1. (a) 11 elements times 2 bytes each = 22; (b) 9 elements times 1 byte each = 9; (c) 4 elements times 8 bytes each = 32.

2. (a) integer is 2 bytes; (b) point is 2 integers for 4 bytes; (c) str255 is 256 bytes; (d) rect is 4 integers for 8 bytes.

3. The integer array should have 4 elements for a rect, so it should be dimensioned at 3 or greater. Integers occupy 2 bytes, so a dimension of 127 (128 elements times 2 = 256 bytes) is big enough for a str255 value. Each element of a character array occupies 1 byte, so a character array needs to be dimensioned at 255 or greater to have 256 bytes available for a str255 value.

Chapter 20

1. **ToolBox SizeWindow** (**OutputWindow]**,400,300,**TRUE**)

2.
```
DIM rect%(3)
rect%(0) = 100
rect%(1) = 100
rect%(2) = 300
rect%(3) = 300
w] = Tool NewWindow (0,0,@rect%(0),"",TRUE,1,-1,-1,FALSE,0,0)
```

3.
```
Applemenu} = Tool GetMenu(1)
ToolBox SetItem (Applemenu},1,'About My Program')
```

4.
```
myMenu} = Tool NewMenu (98,'Choice')
ToolBox AppendMenu (myMenu},'Yes/Y;No;(-;Maybe')
ToolBox InsertMenu (myMenu},0)
ToolBox DrawMenuBar
```

Chapter 21

1.
```
DIM rect%(3)
rect%(0) = 20  ! top
rect%(1) = 20  ! left
rect%(2) = rect%(0) + 20   ! bottom
rect%(3) = rect%(1) + 100  ! right
cntl} = Tool NewControl (OutputWindow],@rect%(0),'push me',TRUE,0,0,1,2,0,0)
```

2.
```
DIM rect%(3)
rect%(0) = 30  ! top
rect%(1) = 30  ! left
rect%(2) = rect%(0) + 20  ! bottom
rect%(3) = rect%(1) + 80  ! right
cntl} = Tool NewControl (OutputWindow],@rect%(0),'Cancel',TRUE,0,0,1,0,0,0)
```

3.
```
ToolBox HideControl (cntl})
ToolBox SizeControl (cntl},60,20)
ToolBox SetCTitle (cntl},'Quit')
ToolBox ShowControl (cntl})
```

4.
```
IF Tool TestControl(scroll},MOUSEH,MOUSEV) = 20 THEN
Toolbox SetCtlValue (scroll},Tool GetCtlValue(scroll})+1)
ENDIF
```

Chapter 22

1.
```
DIM rect%(3)
rect%(0) = 0  ! top
rect%(1) = 0  ! left
rect%(2) = 50   ! bottom
rect%(3) = 240 ! right
ToolBox ClipRect (@rect%(0))
```

2.
```
DIM rect%(3)
rect%(0) = 0  ! top
rect%(1) = 0  ! left
rect%(2) = 100  ! bottom
rect%(3) = 100  ! right
ToolBox PaintArc (@rect%(0),0,90)
```

3.
```
width% = Tool StringWidth ('Title')
GPRINT AT (100-width%) DIV 2,30;'Title'
```

4.
```
DIM pat©(7)
DATA 0,54,73,65,65,34,20,8
FOR count = 0 TO 7
    READ pat©(count)
NEXT count
ToolBox PenPat (@pat©(0))
```

Chapter 23

1.
```
DIM type©(3)
refnum% = Tool OpenResFile ("MoreResources")
FOR count = 1 TO Tool CountTypes
    ToolBox GetIndType (@type©(0),count)
    type$ = ""
    FOR index = 0 TO 3
        type$ = type$ & CHR$(type©(index))
    NEXT index
    t1% = type©(0)*256 + type©(1)
    t2% = type©(2)*256 + type©(3)
    PRINT type$,Tool CountResources(t2%,t1%)
NEXT count
END PROGRAM
```

2.
```
s$ = "There is not enough room on the disk to save that file."
ToolBox ParamText (s$,"","","")
itemHit% = Tool Alert (10,0,0)   ! displays the alert ALRT 10
```

3.
```
s1$ = "The percentages add to less than 100%."
s2$ = "Do you want to continue the calculations anyway?"
ToolBox ParamText (s1$,s2$,"","")
itemHit% = Tool Alert (1,0,0)     ! displays the alert ALRT 1
IF itemHit% = 2 THEN GOSUB GetData  ! Cancel was selected
```

4.
```
! Call SFPutFile
DIM reply©(73) ! result goes here
name$ = "test"
! Load PACK 3
t1% = ASC("P")*256 + ASC("A")
t2% = ASC("C")*256 + ASC("K")
h} = Tool GetResource(t2%,t1%,3)  ! reads PACK 3 code into h}
! Call SFPutFile
PERFORM h} (100,80,'Rename it to:',name$,0,0,@reply©(0),1)
IF reply©(0) = 0 THEN END PROGRAM   ! Cancel was selected
length = reply©(10)
newname$ = ""
FOR i=11 TO 10+length
    newname$ = newname$ & CHR$(reply©(i))
NEXT i
RENAME name$, newname$
END PROGRAM
```

Index

Using Macintosh™ BASIC

Apple Computer, Inc., has now developed a new version of the BASIC programming language uniquely suited to the remarkable Macintosh™ system. Author Richard Norling shares with you his keen insights into the many capabilities of Macintosh BASIC. Thorough descriptions of all Macintosh BASIC statements, functions, and operations are presented, with special emphasis given to graphics and sound. You'll learn how to create Macintosh windows and menus, how to program the mouse, and how to use Macintosh's toolbox commands. Clear and concise, Using Macintosh™ BASIC will enable you to complete your programming projects with confidence.

Richard Norling has spent more than twenty years working with computers and computer languages. Currently, he is president of Language Systems Corporation, a company that specializes in the development of computer software. Norling has written a variety of systems and application software and is co-author of a new statistics package for the Apple® Macintosh™ computer.

McGraw Hill

ISBN 0-07-881157-0