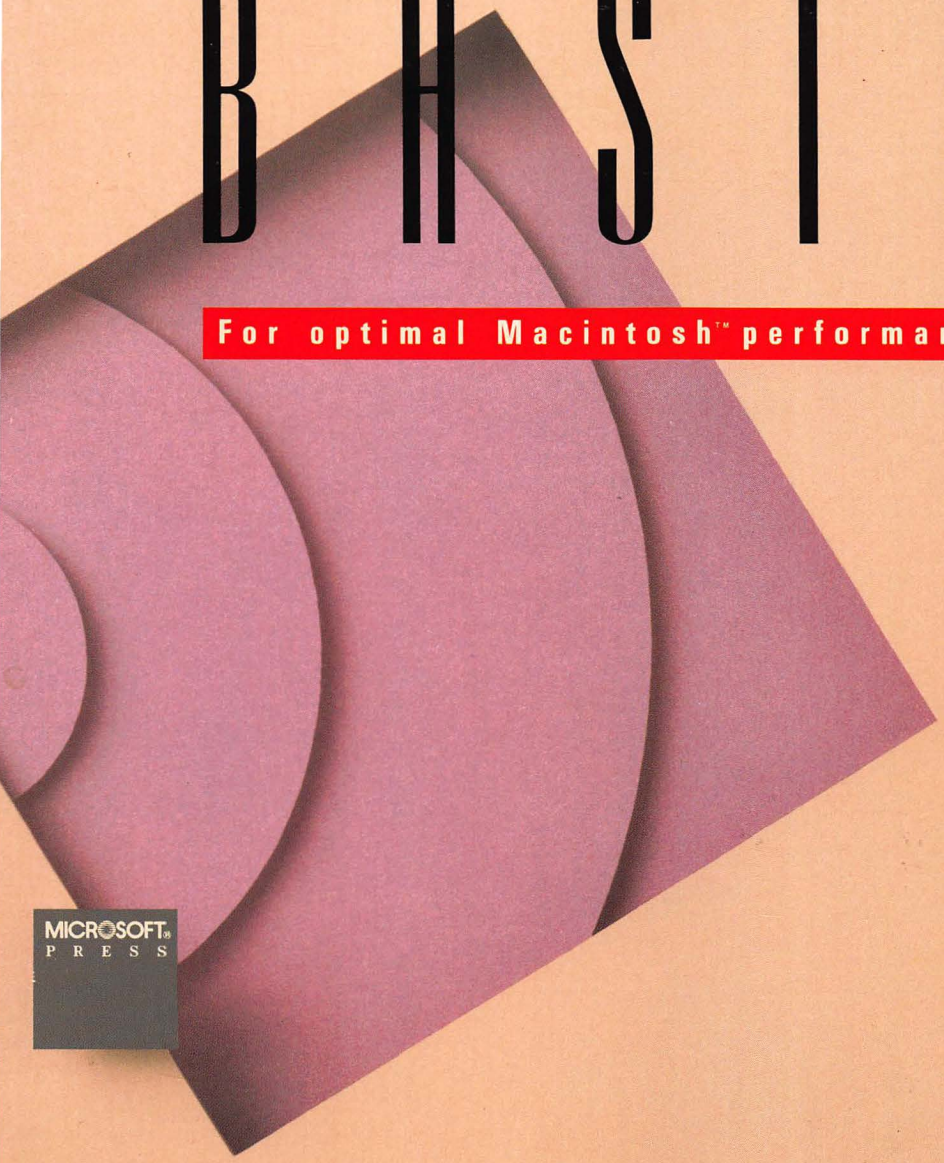
Creative Programming in

MICROSOFT® BASIC

For optimal Macintosh™ performance



MICROSOFT® PRESS

STEVE LAMBERT

Creative Programming in

Microsoft® BASIC

For Optimal
Macintosh™
Performance

Creative Programming in Microsoft® BASIC

For Optimal Macintosh™ Performance

Steve Lambert

MICROSOFT PRESS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
10700 Northup Way, Box 97200, Bellevue, Washington 98009

Library of Congress Cataloging in Publication Data
Lambert, Steve, 1945–
Creative programming in Microsoft BASIC for
optimal Macintosh performance.
Includes index.
1. Macintosh (Computer)—Programming. 2. BASIC
(Computer program language) I. Title.
QA76.8.M3L36 1985 005.265 85-18952
ISBN 0-914845-57-8

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 FGFG 8 9 0 9 8 7 6 5

Distributed to the book trade in the United States by Harper and Row.

Distributed to the book trade in Canada by General Publishing Company, Ltd.

Distributed to the book trade outside the United States and Canada
by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N. Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging in Publication Data available

Contents

Preface

I have been studying Microsoft BASIC, on and off, for about eight years—ever since I assembled my first kit-computer and discovered that if I wanted software for it, I had to write it myself. The availability of computer programs has improved since 1977, but every now and then I still want a program that just doesn't exist, and I dig out my BASIC books, refresh my memory, and become a programmer for a week or so. I have a pretty good collection of books about BASIC, having been through eight or nine computers with five different operating systems since that first kit. And because there are versions of Microsoft BASIC that will take advantage of the special features of almost every computer on the market, the purchase of each computer requires a new BASIC and a new book.

If I have come up with a few truths in these years of learning and relearning BASIC, they are that complex programs are no more difficult to write than simple programs—they are just longer—and the fastest way to learn how to program is by studying other people's programs and figuring out how they work (or why they don't). I have tried to apply these truths to this book about Microsoft BASIC for the Macintosh. The sample programs are not short, simple examples demonstrating features you could easily understand by reading the manual. They are, for the most part, programs that serve a useful purpose or demonstrate techniques that can be applied to such programs. I have tried to point out the pitfalls and "undocumented features" that I stumbled in or over, and explain the reasons for my approach to different problems.

The power and speed of new versions of BASIC have kept pace with the power and speed of newly released computers. If this trend continues—and there is no reason to believe that it won't—it will soon be feasible to write commercial-application,

entertainment, and utility software in BASIC, thereby re-establishing computer programming as a cottage industry open to anyone with a personal computer and the ability to reason clearly.

For the part-time programmer, it is easy to fall into the trap of using the old familiar commands that have faithfully followed you from version to version of BASIC. I hope this book encourages you to explore new territory, expand your horizons, and take advantage of the tremendous potential of the power offered by the team of Microsoft BASIC and the Macintosh.

Acknowledgments

As with every book that manages to make the journey from a mere idea to a finished product in the hands of a reader, this book was produced through the cooperative efforts of many people. The people in the editorial, technical review, and production departments at Microsoft Press have my unqualified respect for their ability to pull all the pieces together into one cohesive package. My particular thanks go to Managing Editor Joyce Cox, who took personal responsibility for editing the manuscript, to Technical Review Manager Barry Preppernau, who provided the backgammon game in Chapter 15 and reviewed all programs for reliability, and to Technical Reviewer Chris Matthews, who built and tested an HBC-1, and helped with the technical aspects of explaining analog-to-digital conversion.

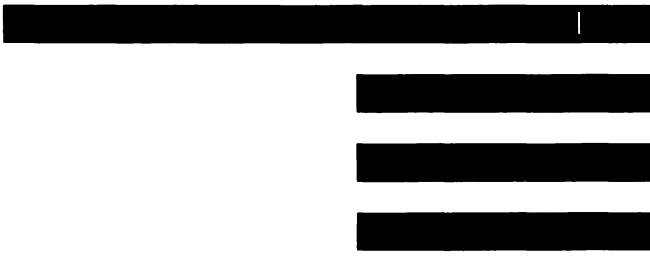
Special thanks go to two people who don't work for Microsoft Press, but who contributed greatly to this book:

John Socha, fellow author and programmer, critically reviewed many of the programs. He forced me to clean up my programming style, taught me much about both Microsoft BASIC and the Macintosh, and was always available to help me through the rough spots.

Gordon Mills, a linear field engineer for Texas Instruments, devoted several hundred hours of his own time to the task of designing the HBC-1, and to the even greater task of explaining its theory of operation to me.

SECTION I

Introduction



An Introduction to BASIC

Chapter 1

When Apple created the Macintosh, it totally broke away from the orderly evolutionary path followed by previous computers. The Mac is not only easy to use, it is friendly to the point of being fresh and has the potential for amazing power. Version 2.0 of Microsoft BASIC made most of this power available to the average programmer, and the enhancements provided by version 2.1 and the BASIC compiler continue to increase the pleasure of programming on the Macintosh.

Although this book is not a primer on either BASIC or the Macintosh, the chapters in this section give a short explanation of how the two fit together, and may help those readers familiar with traditional versions of BASIC on traditional machines to understand the power available through this new partnership.

The BASICs

People purchase the BASIC programming language for various reasons. Some have access to programs that are written in BASIC and need the language to run them. Others have specific problems and think they can create a program in BASIC that will solve them. A few intend to write programs for commercial distribution. But I imagine that the vast majority of the people who purchase BASIC do so out of a vague feeling that it is simply part of owning a computer. The computer is a mysterious device, and programming it occasionally is a responsibility much like the weekly winding of the grandfather clock, required to keep it running smoothly. Even after they discover their clock to be self-winding, people often continue to write programs in BASIC just because they find the exercise to be an entertaining form of mental gymnastics: Writing and debugging a complex BASIC program can generate all the excitement and pleasure of an adventure game you would pay to play in an arcade.

Microsoft BASIC is the most popular of all microcomputer BASICs, with versions that run on almost every brand of computer. Most of these versions are reasonably compatible, the primary differences being in commands that take advantage of special characteristics of individual machines. The additional scope of BASIC commands available on the Macintosh because of its high-resolution screen display and its ability to rapidly manipulate graphic images makes this version of BASIC a particularly challenging and enjoyable language in which to work. Some examples of these special commands are those that allow you to use the mouse, to create windows, and to manipulate graphics. In the version of BASIC for the IBM PC, the commands for dealing with different colors fall into this category. These special commands are obviously not transportable between machines that don't have like capabilities. Most other commands, however, behave exactly the same in the Macintosh version as they do in any other version.

If you have Microsoft BASIC programs that run on other machines, the communication program we develop in Chapter 12 will allow you to transfer them to your Macintosh, and you should be able to run them directly. Programs written in other brands of BASIC can also be translated to Microsoft BASIC for the Macintosh: It is simply a matter of sitting down with the documentation for each brand, finding the commands that aren't directly compatible, and replacing them with the equivalent Microsoft BASIC commands.

BASIC differences

The programs you create in versions 2.0 and above of Microsoft BASIC for the Mac will have a different appearance from programs in other versions with which you may have worked. Figure 1-1, which shows the same program listing in version 2.0 and in a traditional BASIC, illustrates the most obvious visual differences.

The listing on the top is in the traditional format. The lines are numbered in ascending order and the characters all appear in the style in which you typed them. The program listing on the bottom accomplishes the same task, but was modified to take advantage of BASIC for the Mac. And what a difference in appearance.

First, the line numbers are gone, replaced by an occasional label. Since the normal flow of a BASIC program is from the first line to the last, the only time you have to provide a number or label for a line is when the normal flow is to be diverted to that line from elsewhere in the program. A label can be indented, but it must be the first

```
1000 ' Communication Loop
1010 WHILE true
1020    IF pauseFlag THEN GOTO 1500
1030    IF (LOC(1) = 0) AND stopFlag THEN PRINT #1, xON$ : stopFlag = false
1040    WHILE LOC(1) = 0                          'nothing waiting to come in
1050       GOSUB 2000                              'send key typed to file #1
1060       IF (sendFlag OR viewFlag) AND NOT waitFlag THEN GOSUB 2500
1070       IF endViewFlag THEN GOSUB 3000
1080    WEND
1090 WEND
```

```
CommLoop:
  WHILE true
    IF pauseFlag THEN GOTO CommSkip
    IF (LOC(1) = 0) AND stopFlag THEN PRINT #1, xON$; : stopFlag = false
    WHILE LOC(1) = 0                                    'nothing waiting to come in
      SendKey                                           'send key typed to file #1
      IF (sendFlag OR viewFlag) AND NOT waitFlag THEN CALL SendLine
      IF endViewFlag THEN GOSUB EndFile
    WEND
  WEND
```

Figure 1-1. Traditional BASIC versus Microsoft BASIC for the Mac

item printed on the line and must be followed by a colon. It can be practically any combination of letters, numbers, and periods, as long as it starts with a letter, is no more than 40 characters long, and contains no spaces. Multiple words can be run together with the initial letter of each capitalized or with a period separating them to make them more readable. This flexibility allows you to use labels to specify the purpose of a line as well as its location: *AccountBalance:*, *InvestmentCredit:*, *MouseAction:*, *Dial.The.Phone:*, and so on. Since a label can appear on a line by itself, it can also be used to separate and identify subroutines and other program segments.

The second obvious feature of a version 2.0 program listing is that all the reserved words—BASIC statements and functions—are in a boldface type style. This happens automatically: When you press Return after typing a line, BASIC picks out all the reserved words and converts them to this style. This helps you easily locate many typos, since a misspelled reserved word will stay as it was typed. This feature also helps you avoid the accidental use of an obscure reserved word as a variable name: When the variable you just typed springs boldly back at you, you can change it on the spot, rather than waiting until you have a syntax error while running the program.

Interpreter versus compiler

Microsoft BASIC comes in two forms: as an interpreter and as a compiler. The difference between the two is the manner in which they translate high-level BASIC commands into the low-level language understood by the computer. Interpreters translate a program one line at a time, executing each translated line before translating the next. Compilers translate the entire program, and then run it. Microsoft BASIC 2.0 and 2.1 are interpreters. Microsoft is developing a BASIC compiler for the Macintosh.

There are advantages to each style of translation. With an interpreter you can run a program as soon as you finish typing it. Then, if you want to make changes, you can edit it and immediately run it again. With a compiler, there is an extra step in the middle—the compilation—which can take quite a bit of time. However, a compiled program runs substantially faster than an interpreted version of the same program, and doesn't require that BASIC be loaded first.

Interpreters are usually considered most convenient when writing and debugging a program. Compilers are most convenient when running a finished program. Compiled languages are especially appreciated by programmers offering their work for sale, as they don't have to depend on the purchaser having the appropriate version of BASIC in order to run the program. The ideal situation is a version of BASIC that can run your program interpretively until all debugging is complete, and then compile it. This level of convenience is approached by the combination of BASICs available from Microsoft.

Since the compiler will compile a program written with the interpreter, the decision to compile a program can be made retroactively and has no particular influence on how the program is written. And since this book is about writing programs, it is based on the interpretive versions of BASIC—specifically, on versions 2.0 and above of Microsoft BASIC for the Macintosh.

Generic syntax

The commands available to you in BASIC often have options that aren't necessarily obvious from the context in which the command appears in a program. So that you will be aware of all the possible options that might be available, as I introduce

each new BASIC command, I will also list its generic syntax using the same format as is used in the Microsoft BASIC manual. For example:

INPUT$(X[,[#]*filenumber*])

Significant aspects of this format are:

Feature	Meaning
CAPS	Capitalized words (longer than one character) must be typed letter for letter as shown, although you don't have to capitalize them; BASIC automatically does this for you
italics	Italicized words represent program-specific variables to be supplied by the user
X	A single capitalized letter, or a letter followed by a string-specifier (such as X$) also represents a program-specific variable that the user supplies
( )	Parentheses are part of the command and must be typed in
[ ]	Square brackets indicate optional parameters; they are not typed in
. . .	Ellipses indicate that the preceding item may be repeated any number of times

Programming style

One of the things about programming that makes it enjoyable for me is the fact that there are many ways to get the job done, and the method is rarely as important as the result. There is good programming style, and there are techniques that are generally considered "proper." These contribute to the speed and efficiency with which you write programs, and the ease with which their operation can be understood by others, but the ultimate test of whether or not a program is "good" is how well it does the job. Each programmer is free to develop his or her own style and structure.

My programming *technique,* with anything other than the simplest of programs, is to sit down first with a pencil and paper and list the things the program will do and the order in which they will be done. My approach is far too primitive to be glorified with the title "flow chart"; it is more a simple sketch. I also sketch where I think objects should appear on the screen at different points in the program (we will develop a grid in the next section that will help in this task).

My programming *style* is a little more involved. As I said earlier, programming style is pretty much up to the programmer and can vary from program to program—unless you are writing a book about programming and need to make it easy for your readers to move from one program to another without having to reorient themselves after each move. Here are some pointers about the style I have chosen for this book.

Comments

Most of the programs in this book are heavily commented. If you choose to type these programs into your Macintosh and run them, there is really no point in typing the comments as you can always refer to the book. However, if you modify a program, you should add a comment of your own to explain each modification so that you, or someone else, will understand its purpose when it is stumbled across in the future.

BASIC recognizes two types of comment, set off either by REM or by a single quote mark. I use only the single quote mark, but use it in several distinct ways (Figure 1-2). Comments that apply to a whole section of the program are typed flush left, and have at least one line of open space above. Comments that apply only to the line that follows them are indented as far as that line. Comments that are on their own lines are set off by the single quote mark followed by an asterisk. Short comments tacked onto the end of a command line are set off by only the single quote mark.

```
'*
'* Define variables.
'*
top = 20                                    'top of output window
left = 20                                   'left side
bottom = 300                                'bottom
right = 500                                 'right side

'*
'* Open window for display.
'*
WINDOW 1, , (0, 20) - (512, 342), 3
TEXTFONT 4                                  'Monaco font -- monospaced
TEXTSIZE 6                                  'make it small, to get numbers in
```

Figure 1-2. Samples of comments

You will probably notice that the short comments to the right of the command lines in my programs are aligned with the right edge of the display. This is not a new feature of BASIC, but was done to increase the readability of the program listings by separating the comments from the commands as much as possible. (It also produces a better balanced and therefore more pleasing page.)

One other comment on comments. The maximum size program that BASIC can load is determined by the amount of random access memory (RAM) available in your Macintosh. This is not usually a significant factor with a 512K Mac, which has about 370K available for your program, but a 128K Macintosh only has about 20K available for your program and its variables. Often, removing the comments will reduce the size of an unloadable program enough so that it can be loaded. One of the utility programs developed in Section VI performs this task for you.

Labels

I use labels only where necessary: that is, when the normal program flow is diverted to the label. In other words, I don't use labels simply as comments; I use them in place of traditional line numbers. Labels (at least in this book) have initial caps and describe the purpose of the section they label. Statements can follow labels on the same line, but I consistently put labels on their own line.

Variable names

I have adopted the convention of using initial lowercase letters for variable names. If the name is composed of multiple words run together, I capitalize the first letter of the second and all following words. Space permitting, variable names are long enough to describe what they represent. The maximum length and components of a variable name are the same as for a label: 1 to 40 letters, numbers, or periods.

Programmers accustomed to other versions of BASIC may feel that long variable names and labels are an extravagance they will pay for with slower-running programs. This is not the case in BASIC versions 2.1 and above, which tokenize variable and label names. This means that they replace the variable name with a symbol (token), which is stored in a list along with the name it replaced. The only time your meaningful names and labels are used is when they are displayed on the screen for you to read; BASIC itself works strictly with the tokens.

There is an interesting side effect to this method of dealing with variable names. BASIC stores the variable names with the exact combination of upper- and lowercase characters you type, but recognizes the same combination of characters, however they are typed, as that variable. BASIC stores the name only once, and then uses that name each time it has to display the variable. Each time you type the name, the stored version is updated with the specific combination of upper- and lowercase characters you type. As a result, each time the screen is refreshed—that is, each time a line is redrawn because you have edited it, or because the screen has scrolled—all occurrences of the name are updated to match the way you last typed it.

Indents

Labels and major comments are printed flush left. The body of the program is indented four spaces, and the body of each FOR...NEXT and WHILE...WEND loop is indented an additional four spaces from its beginning level.

Spaces

BASIC has very little use for spaces; they are needed only to make the program more readable for people. But since people, as well as a few computers, will be reading these programs, I separate almost everything with spaces. An advantage of following this practice, even if you aren't producing programs for public consumption, is that it is easier to automatically search for and replace a word if it is always set off by spaces.

GOTO statements

The use of GOTO statements is generally frowned on by people who teach BASIC programming. The justification for this attitude is that GOTO statements, improperly used, can make it very difficult for a person reading the program to follow its logical flow. The effect of too many GOTOs is often called "spaghetti logic." The proper use of subroutines and subprograms usually eliminates the need for most GOTO statements.

My primary use of GOTOs is to jump around several lines, or return to the beginning of a loop. The purist would avoid even these uses, possibly through the use of an IF...THEN...ELSE statement, or a WHILE...WEND loop. Purity, however, has never been one of my major vices.

Programs on disk

The purpose of the programs in this book is to help you understand how BASIC commands are used to create useful programs. You can learn about programming by simply reading the program listings and their explanations, but it is much more effective if you also run the programs, study what they do, and then make changes to test your understanding. However, typing the longer programs is a rather tedious task, so if you would like to run the programs in this book without having to type them, you can purchase the Companion Disk to *Creative Programming in Microsoft BASIC*, which contains the programs exactly as they appear in the book. The disk also contains additional information about the construction of the HBC-1 analog-to-digital converter described in Chapter 18 and Appendix C.

You can order the disk with the order card bound into this book, or by sending your name and address, along with $19.95 (plus $1.00 for postage and handling), for each disk. U.S. funds only, please. California residents must add 6.5% sales tax and Washington state residents 8.1% sales tax. Payment must be made by check or credit card. Include your MasterCard, VISA, or American Express Card number, along with the expiration date, with your order. Send your order to: Microsoft Press, Attn: LDSK, 10700 Northup Way, Bellevue, WA 98004. Please allow four weeks for delivery.

I have made every attempt to ensure that the programs on the Companion Disk are "bug-free." However, if you should discover something I missed, you can drop me a line at: 15 Central Way, #280, Kirkland, WA 98033. I will include the correction on future disks.

Now that you have seen a few of the ways Microsoft BASIC for the Mac differs from other versions of BASIC, and I've told you about the way I format my own BASIC programs, let's move on to Chapter 2 for a look at some of the unique aspects of the Macintosh.

The Macintosh BASIC Environment

Chapter 2

When Apple released the Macintosh, it stressed the new computer's "user friendliness" and ease of operation. It billed the Mac as the machine for "the rest of us"—for the people who have no desire to become programmers in order to use a computer. The Mac has lived up to this billing, but with the development of programming languages that allow access to its power, it has also become a delight for programmers, from neophyte to expert. This chapter provides an overview of the Macintosh's features as they apply to Microsoft BASIC, for those users who are not familiar with this combination.

First let's take a quick tour of the BASIC work environment—the screenful of windows and menus that Apple calls a desktop. If you have seen other Macintosh applications, the items on the desktop shown in Figure 2-1 should be familiar.

At the top is the menu bar, displaying the titles that, with a press of the mouse button, drop down into full command menus. You choose a command by dragging down the list and releasing the button while the pointer is over the desired command.

Three windows occupy the remainder of the screen: the List, Output (labeled Untitled in Figure 2-1), and Command windows. From the time you load the BASIC interpreter, it is in one of three modes of operation: edit, program execution, or command. Each mode is associated with a different window on the Macintosh desktop. The edit mode is used to create and modify programs in the List window. The program-execution mode is used to run these programs, displaying the results in the Output window. The command mode allows you to enter commands directly into the Command window and have them immediately executed. We will look at each of these windows a little more closely in a few moments.

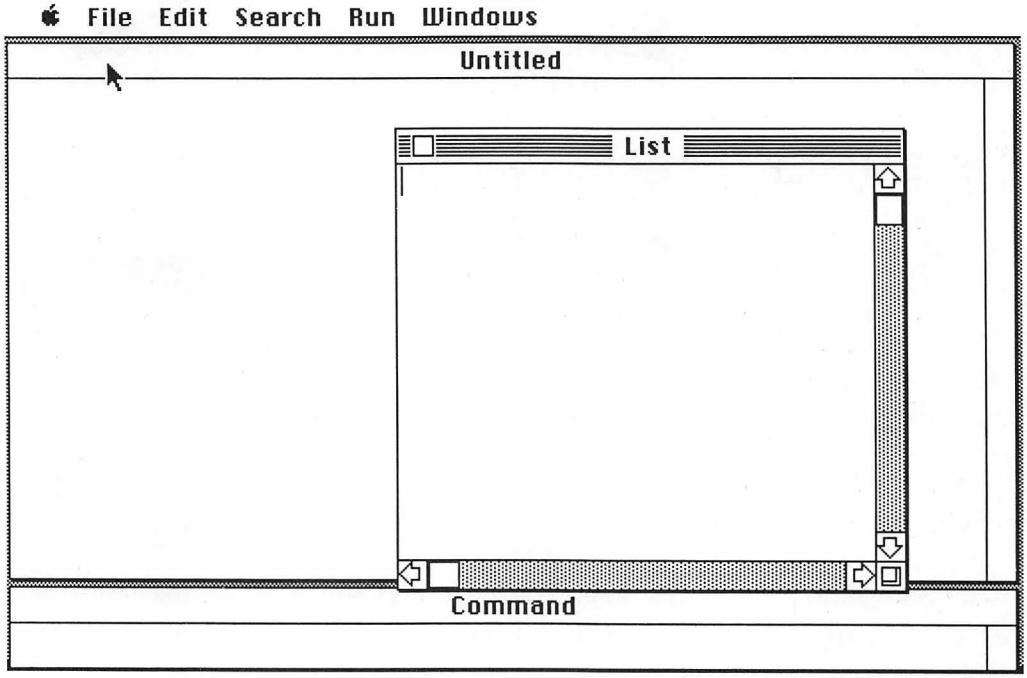


Figure 2-1. The BASIC desktop

You have undoubtedly used other Macintosh applications in which you could open and close windows, use the mouse to change their size and location, click buttons in them to make selections, and so on. So you won't be too amazed to hear that BASIC makes use of all these same features. What may come as a pleasant surprise, though, is the fact that BASIC also allows you to integrate these features into the programs you write. Let's take a closer look at what we have here.

The Macintosh screen

The first time you saw the Macintosh in operation, you were undoubtedly impressed by the sharpness of the image on its screen. This crispness is due to the high resolution and small size of the Macintosh screen compared with most other computer screens. The Macintosh always operates in a graphic mode (as opposed to a text mode), creating images on its screen by turning on or off little dots called pixels. Section II explains how you can use BASIC to control the condition of these pixels, either

individually or in groups, to create and move images, including windows, pushbuttons, and pictures you bring in from MacPaint.

The mouse and pointer

As you know, moving the mouse controls where the pointer points. Clicking the mouse button, either once or rapidly several times, selects an object for some action or gets the action going. Holding the button down while moving the mouse controls other activities, perhaps moving a window or drawing a line. Each of these mouse events is constantly monitored by the Macintosh and information about them is made available to you through BASIC. You can tell where the mouse is now and where it was when it was last clicked, double clicked, or even triple clicked. You can identify the beginning and ending points of a drag, as well as the ID number of any button, box, or window clicked. You can even change the shape of the pointer or hide it away when it isn't needed. You will be familiar with most of these techniques by the time you get through Section III.

The standard windows

BASIC itself uses four windows to manage your creative efforts while you are writing a program. Three of these windows—List, Command, and Output—were shown in the initial desktop displayed earlier; the fourth window—a second List window—is used with the first List window to simultaneously list different parts of the same program.

The Command window

When you load the BASIC interpreter, the Command window is active and BASIC is in the immediate mode, waiting for you to type a command. Anything you type appears in the Command window. When you press Return, BASIC assumes the contents of this window to be a command and attempts to execute it. After the command is executed, it is discarded; to repeat the command, you have to enter it again. You can enter multiple commands in the Command window by separating them with colons. If you enter more text than will fit on one line of the Command window, the additional text is automatically wrapped around to the next line—up to a total of about 250 characters.

If you exceed this limit, an error message is displayed when you press Return, and your commands are not executed.

NOTE: The Cut, Copy, and Paste commands function in the Command window just as they do elsewhere. If you think you might want to repeat a command, or change it slightly and try it again, copy it to the Clipboard before pressing Return. After the command has been executed, press Command-V to paste a copy from the Clipboard back into the Command window.

You enter commands in the Command window to test or change the value of variables while debugging a program, to get the results of a single calculation rapidly, or to try out a short string of commands before using it in a program. To create a program composed of more than about three lines, however, you must use the List window. If the List window is visible, simply click in it to enter the edit mode. If the List window is not visible, choose Show List from the Windows menu or type List in the Command window and press Return.

The List window

BASIC's List window is essentially a specialized word-processing program, designed to help you write and modify programs. It allows you to use the Macintosh's standard Cut, Copy, and Paste commands to edit text, and has word-processing features such as Find and Replace.

The title bar, scroll bars, and size box in this window function just as they do in similar windows in other Macintosh applications: You can drag the window by its title bar, scroll to different spots in the listing with the scroll bars, and change the size of the window with the size box.

There is a shortcut for changing the size of any BASIC window that has a title bar: You can double-click the title bar. BASIC remembers two sizes for each window, and when you double-click, it switches the window to the other size.

The second List window

The second List window, which is identical to the first, can be used to display a different section of the program than the one in the first window. This is very handy when you are reorganizing a program by cutting and pasting segments within it.

The Output window

The Output window is automatically opened by BASIC to display the results of the program you run in the List Window or the commands you enter in the Command window. BASIC designates the Output window as window #1, and counts it as one of four output windows you can have open in your programs. You can use BASIC commands to create another window as #1, in which case the stock Output window is replaced with the one of your own design.

Creating your own windows

The windows discussed so far are displayed by the BASIC interpreter. The programs you write in BASIC can also display up to four windows on the screen at a time, and you can tailor the size, shape, and style of these windows to your needs. Typically, they are used to gather input, display the result of whatever task your program is performing, or alert the user to pertinent points along the way.

Most of the housekeeping tasks associated with displaying windows on your screen are taken care of by the Macintosh. For example, if you create a window that can be dragged with the mouse, or expanded and contracted, the Macintosh steps in on your behalf when the user attempts one of these operations. Just as with significant events in the life of your mouse, the Macintosh traps significant window events, stores information about them, and passes it on to you if you request it. Information stored includes the size of each window, the currently active window, the number of the most recently pressed button or most recently used edit field, and whether a window has had a previously covered area exposed, and therefore needs to be refreshed. You'll find out more about trapping these events in Section II.

Dialog boxes

A dialog box is simply a window that has been put to a special purpose. If you are near your Macintosh now, you can have a look at a dialog box by choosing from the menu any item with an ellipsis after it. Choosing Open... from the File menu, for example, produces a dialog box similar to the one shown in Figure 2-2. This dialog box is used to gather the information needed to carry out the Open... command. It offers items you can scroll through and buttons you can click; other dialog boxes might have edit fields into which you can type a word or phrase.

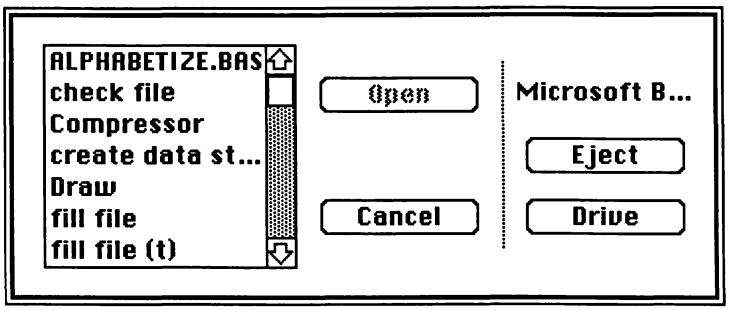


Figure 2-2. The Open dialog box

You can build your own dialog boxes in BASIC or you can use a few of the stock ones offered by the Macintosh. As an example, the dialog box you produced by choosing Open... from the File menu could also be produced from within a BASIC program with the FILES$(1) function, which you'll learn about in Section II. Not only does this function produce the dialog box, it also retrieves the user's selection. Isn't that easier than the old *LINE INPUT "Type the name of the file (filename.ext) to open";f$* routine? And selecting from a list in a dialog box solves the problem of whether or not the user's entry is correctly typed and spelled.

Menus

The menu bar across the top of the screen has six items on it. The Apple icon at the left end heads a menu that is common to most Macintosh applications; the remaining five menus contain some items that are similar to menu items in other applications, but most are unique to BASIC. If you place the pointer over one of the menu titles and press the mouse button, that entire menu drops down. If you drag the pointer down the menu, the commands that are currently available become highlighted when the pointer is over them. Releasing the mouse button while a menu item is highlighted chooses that item, causing whatever action it controls to take place.

The commands available from the BASIC menu bar are grouped by function and arranged beneath titles that more or less describe the function. Most commands in the File and Run menus serve the same purpose as similarly named commands that you would type in other versions of BASIC. Figure 2-3 illustrates the relationship between menu items, their keyboard equivalents (typed in the Command window) and standard BASIC commands.

Menu	Item	Keyboard Equivalent	BASIC on IBM PC
File	New	New	New
	Open...	Load	Load
	Close	Window Close #	n/a (not the same as Close (*filename*))
	Save	n/a	n/a
	Save As...	Save	Save
	Print...	Llist (*List, LPT1:Prompt* is exact replacement)	Llist
	Quit	System	System
Edit	Cut	n/a	n/a
	Copy	n/a	n/a
	Paste	n/a	n/a
Search	Find...	n/a	n/a
	Find Next	n/a	n/a
	Find Selected Text	n/a	n/a
	Find Label	n/a	n/a
	Find the Cursor	n/a	n/a
	Replace...	n/a	n/a
Run	Start	Run	Run
	Stop	Stop	Stop
	Continue	Cont	Cont
	Suspend	n/a	n/a
	Trace On/Off	Tron/Troff	Tron/Troff
	Step	n/a	n/a
Window	Show Command	n/a	n/a
	Show List	List	n/a
	Show Second List	n/a	n/a
	Show Output	Window #1	n/a

Figure 2-3. Macintosh menu options and their keyboard equivalents

The individual menu commands are explained in detail in the Microsoft BASIC manual and in introductory books on BASIC. Here is an overview of the commands and a brief description of those that either aren't obvious or don't relate directly to traditional BASIC commands.

The commands that you can select by dragging through the File menu are used primarily to move programs in and out of BASIC.

Command	Action
New	Clears memory before you type new program
Open	Brings program in from disk
Close	Closes active window on screen; not same as BASIC statement that closes file
Save	Updates program that you have already named
Save As...	Assigns name to program currently in memory; can also be used to change name of program or save copy on different disk
Print...	Sends program listing to printer; standard Macintosh Print dialog boxes appear, allowing you to set format
Quit	Returns you to Macintosh desktop (the Finder), which appears when you turn on machine and insert disk; if there are unsaved changes to program, you are prompted to save them

The Edit menu commands, used when typing or modifying a program in the List or Command window, are common to almost all Macintosh applications. Once accustomed to cutting, copying, and pasting, you will wonder how you ever got along without them.

Command	Action
Cut	Deletes currently selected text and replaces contents of Clipboard with deleted text
Copy	Replaces contents of Clipboard with copy of currently selected text, without deleting text
Paste	Puts copy of Clipboard contents at location currently selected in List or Command window

The Search menu commands are useful when editing and debugging a program, and are great when you decide to clean up your program by renaming variables or labels. With the exception of Find Label and Find the Cursor, these are standard word-processor features.

Command	Action
Find Label	Adds colon to selected (highlighted) program text, and searches for matching label
Find the Cursor	Scrolls List window to display section of program containing cursor

NOTE: If you do loose the cursor, which is possible when you scroll through a long listing, there is actually no need to resort to the Find the Cursor command: You can return to the cursor immediately, with no effect on the program, by typing any character followed by a Backspace (typing the character brings the section of the text containing the cursor onto the screen and enters the character; pressing the Backspace key deletes the character, leaving the text as it was).

The Run menu commands control the execution of your program. Start, Stop, and Continue are pretty obvious. The remainder cause the following actions:

Command	Action
Suspend	Causes execution of program to pause until any key is pressed (other than Command-S, which invokes Suspend)
Trace On	When you select Trace On, menu item changes to Trace Off. Each statement is framed in List window as it is executed and result is displayed in Output window, allowing you to watch cause and effect to discover where things are going wrong
Step	Essentially same as Trace command, except it executes only one statement and then returns to immediate mode, allowing you to test or alter variables before continuing to next step

The Windows menu commands all bring hidden or closed windows to the surface and make them active.

Keyboard command shortcuts

The most-often-used menu commands can be chosen by holding down the Command (⌘) key and pressing a letter that represents the command. Where available, the keyboard shortcut appears to the right of the command on the menu. For convenience they're also listed here in the table on the following page.

Command	Keyboard shortcut
Cut	⌘ X
Copy	⌘ C
Paste	⌘ V
Find	⌘ F
Find Next	⌘ N
Start	⌘ R
Stop	⌘ .
Suspend	⌘ S
Step	⌘ T
Show List	⌘ L

Creating your own menus

Your BASIC programs can display up to ten drop-down menus (in addition to the menu beneath the Apple icon, over which you have no control), with up to 20 items on each. The custom menus you create can replace some or all of BASICs five stock menus. Imagine how much more convenient these are than the conventional hierarchical menus, where typing a response to one menu leads you to another menu, and another, and so on. Your program need only specify the items for each menu and what will be done when one is chosen: BASIC, or the Macintosh operating system, keeps track of whether the pointer is over an item and what is going on with the mouse button. We will start creating our own menus in Section III.

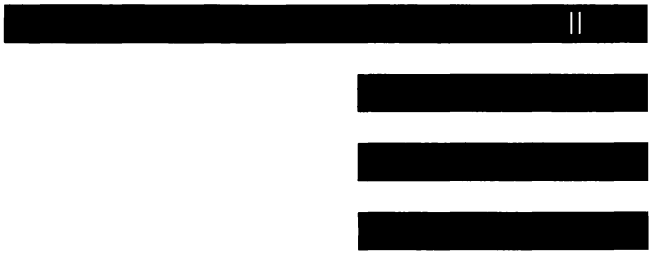
Memory Management

A stock Macintosh, as it comes from the factory, has either 128K or 512K of RAM. Because the Macintosh operating system requires a lot of memory, and applications typically gobble more, other companies offer upgrade kits to expand the memory to as much as 2M (2 million bytes). Additional memory will allow you to run larger programs, and sometimes increases the speed of the programs you could run before the addition; but efficient management of whatever memory you have will allow optimum performance from the Macintosh. We will look at specific memory-management techniques as we develop large programs that require them.

Now that we have the background information out of the way, let's dig into the fun stuff. The next section deals with how you can control the display of text and pictures on the Macintosh screen.

SECTION II

Graphics



Introduction to Graphics

Chapter 3

An obvious and very pleasant difference between using the Macintosh and using practically any other computer results from what is called the "user interface." This is computer jargon for the manner in which you relate to the machine: how you give commands and provide the information needed to produce the results you desire. As you undoubtedly realized the first time you used it, the Macintosh leans heavily on graphic images and the mouse to communicate with the user. You quickly learn to recognize the graphic icons representing the files on your disk, and making selections from drop-down menus and dialog boxes by clicking buttons with your mouse becomes second nature.

One of Apple's goals in developing the Macintosh was to create a user interface that would be easy for the average person to understand, and then to make the components of this interface readily available to companies creating application software for the Macintosh. As a result, most commercial Mac applications—spreadsheets, word processors, database managers, and so on—make use of a similar interface. Once you master one program, you can learn the others relatively quickly, since you already understand the basics.

As we saw in Chapter 2, versions 2.0 and later of Microsoft BASIC for the Macintosh provide access to interface features such as menus, multiple windows, dialog boxes, and buttons, allowing you to create programs with all the power and pizzazz of those produced by the professionals. They also allow you to store, display, move, and manipulate other kinds of visual information: pictures you create in BASIC and those you transfer in from other programs, such as MacPaint or Microsoft Chart. You will utilize these features most effectively if you understand how images are produced on the Macintosh screen.

Screen control

The portion of the screen that lights up to display an image is 6.75 inches wide by 4.5 inches high. The patterns that appear in this area, whether they represent letters of the alphabet or parts of a picture, are created by turning on or off individual elements in a grid of small rectangles called pixels. The pixels are neatly organized into 342 rows and 512 columns. This works out to about 5765 pixels per square inch (as opposed to 2144 pixels per square inch for an IBM PC in its highest resolution graphic mode, displayed on a typical 12 inch monitor). If you have used FatBits in MacPaint, you have already seen and controlled individual pixels. Figure 3-1 shows a portion of the screen, enlarged by FatBits.

Many of the commands available in BASIC allow you to manipulate individual pixels or groups of pixels to form lines, shapes, and patterns. In order to do this accurately, you must have a way to identify the location of the pixels you want to control.

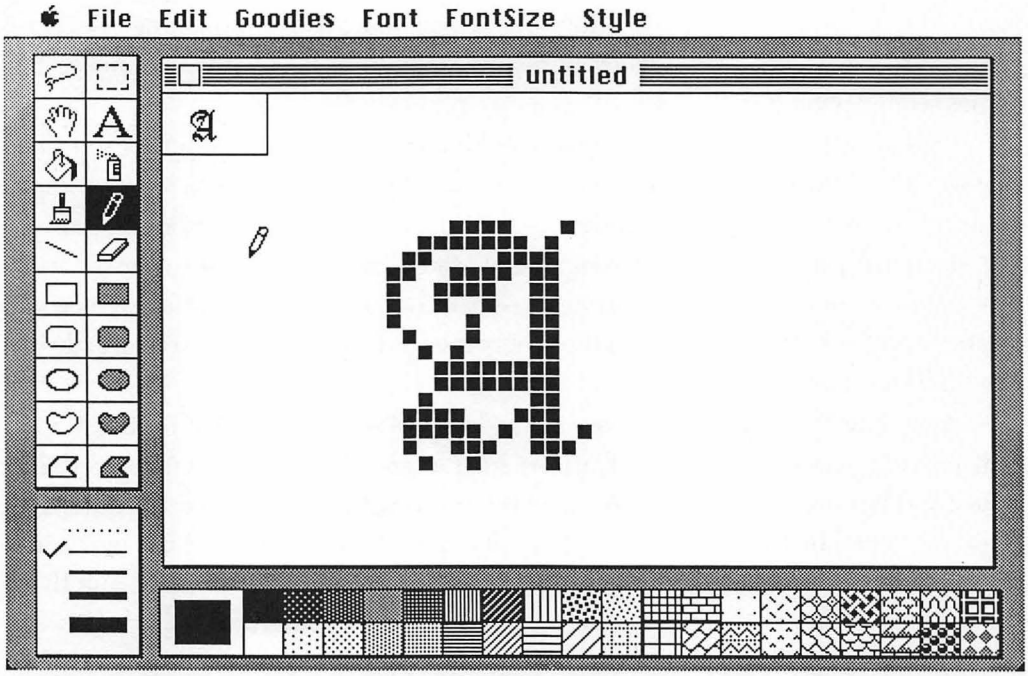


Figure 3-1. Pixels, as enlarged by FatBits

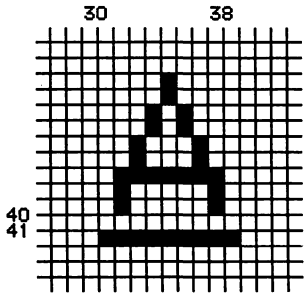


Figure 3-2. The display coordinate system

You do this on the Macintosh not by referencing the pixel itself, but by referencing an imaginary and infinitely small point at the upper left corner of the pixel. Let's have another look at some fatbits, this time with an imaginary grid laid over them, as shown in Figure 3-2.

This grid extends over the entire display area of the Macintosh screen, and forms the coordinate system used to specify where graphics and text are to be printed. The vertical lines are numbered consecutively from 0 to 512, and the horizontal from 0 to 342. You can specify a position at which something is to be displayed by specifying the coordinates of the point formed by the junction of a vertical and horizontal line on this grid, with the point (0,0) in the upper left corner of the screen and the point (512,342) in the lower right corner.

Pixel commands

The actual pixels affected by your command depend upon the type of command used. There are two classes of display commands: those provided directly by Apple and available to the programmer as built-in programs, known as ROM (read-only memory) calls, and those provided by Microsoft as BASIC statements. We will see examples of both in the programs we develop, so let's take a look at the differences now.

A pair of coordinates passed to a ROM call references the pixel directly below and to the right of the coordinate. The same coordinates used in a BASIC statement reference the pixel above and to the right. The difference between the two is most evident when printing text on the screen. The BASIC PRINT command that displayed the letter *A* in Figure 3-2 instructed the Macintosh to print the letter at location

(30,40). The character is nine pixels high and seven pixels wide (plus a pixel of white space on each side), and extends from grid location (30,40) *upward* and to the right. The ROM call that drew the line located one pixel below the letter instructed the Macintosh to draw a line from (30,41) to (38,41). The line fills the row of pixels hanging *below* grid line 41.

Multiple windows

The Macintosh allows you to display information in up to four windows at a time. To make the mechanics of this somewhat easier, each window has its own coordinate system, starting at (0,0) in its upper left corner. This means you can judge the placement of images using the borders of the window as guidelines, rather than having to pinpoint particular pixel locations within the screen as a whole. Also, as you move a window, the coordinates of images in it stay the same, since they are always relative to the corner of the window. We will play with this feature as we develop our first program in the next chapter. We will also experiment a little with where the different display commands place text and graphics.

Tracking the Mouse

Chapter 4

Although this first program, shown in Figure 4-1, is rather short, it introduces several interesting commands and will, I think, give you a better understanding of how the grid coordinate system is used to identify pixel locations. I'll explain how this program works in a moment. My practice throughout this book is to explain each command the first time I use it. (If you skip ahead and find that you've missed an explanation, you can either locate the command in the Index and return to the explanation, or look it up in Appendix A, which contains an alphabetical listing of all the Microsoft BASIC commands.) My explanations will probably be much easier to follow if you first load BASIC into your Mac, type into the List window each program as it appears in the text (less the comments), and run it. If you have experience with other versions of BASIC, you may want to simply glance through the explanation of each program, to pick up new commands.

After you have typed in the program in Figure 4-1 and chosen Start from the Run menu, position the pointer within the small window that is created on the screen and press the mouse button. The (x,y) coordinates that identify the grid location of the pixel at the head of the pointer (using ROM convention) are printed in the upper left corner of the window, as shown in Figure 4-2. Now drag the mouse and watch the coordinates change.

We will try a few other experiments with this display shortly, but first let's take a closer look at the individual lines in the program and consider what each command contributes to the final display.

```
'* Tracking the Mouse
'*

   '*
   '* Open two windows.
   '*
   WINDOW 1, "Background", (0, 38) - (512, 342), 1
   WINDOW 2, "Tracking the Mouse", (275, 150) - (475, 300), 1

   '*
   '* Print instructions.
   '*
   CALL MOVETO (2, 50)                                    'position to print onscreen
   PRINT "Position the pointer in this"
   PRINT "window, press the mouse "
   PRINT "button, and drag."
   PRINT "Triple click to quit."

   '*
   '* Wait for the mouse button and print coordinates.
   '*
   WHILE MOUSE(0) <> -3                                   'while no triple click
      WHILE MOUSE(0) = -1                                 'while button is down
         CALL MOVETO (10, 20)
         PRINT "x="; MOUSE(1), "y="; MOUSE(2)             'x and y coordinates
      WEND

      '*
      '* The button has been released. Go back and wait for it to be
      '* pressed again.
      '*
   WEND
   END
```

Figure 4-1. Mouse-tracking program listing

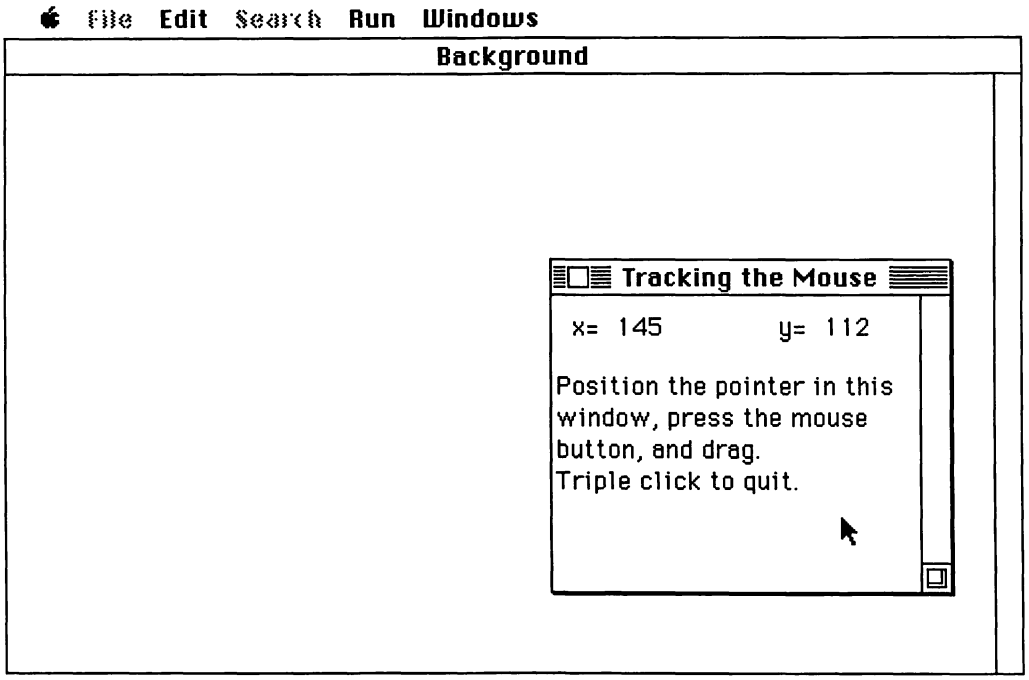


Figure 4-2. Tracking the mouse

The WINDOW command

The WINDOW command is a new one, even for those familiar with versions of BASIC for other machines. This program uses two windows: a background window that fills the entire screen, overlaid by a display window that starts rather small, but can be enlarged. (The idea of a background window is common to many of my programs: Its purpose is to cover the List and Command windows that BASIC normally displays so that the program output is more prominent on the screen.) Here are the statements that create these windows:

```
WINDOW 1, "Background", (0, 38) - (512, 342), 1
WINDOW 2, "Tracking the Mouse", (275, 150) - (475, 300), 1
```

As with many BASIC commands, there are several versions of the WINDOW command: four statements and one function. Here is the syntax for each version. First the statements:

WINDOW *ID*[, [*title*][, [*rectangle*] [, *type*]]]

WINDOW CLOSE *ID*

WINDOW OUTPUT *ID*

WINDOW OUTPUT *file #*

And here's the function:

WINDOW(*n*)

The four statements allow you to create windows, close existing windows, specify which will be the output window, and redirect the output from the screen to a file. The function returns information about the active and the current output windows.

The mouse-tracking program uses the first form of the WINDOW command to create two windows. The active window is the highlighted window in front of any other window on the desktop. In our program, window #2 is the active window, as it was the last window drawn and we haven't specifically designated another window as active. INPUT statements, MOUSE and DIALOG functions, and dialog event trapping (all of these subjects will be covered later) are monitored in the active window.

The current output window is the one in which print and graphic statements display their results. When a window is created, it automatically becomes *both* the active window and the current output window, but when more than one window is created (recall that you can have up to four), you can designate separate windows as active and current output.

In the WINDOW statement that creates it, our active window is identified as window #2 by the *ID* parameter and given the *title* Tracking the Mouse. The *rectangle* option determines where on the screen the window will appear, by specifying the (x,y) coordinates of its upper left and lower right corners: This output window stretches from (275,150) to (475,300). The *type* option specifies the window type, from the four possible types shown in Figure 4-3.

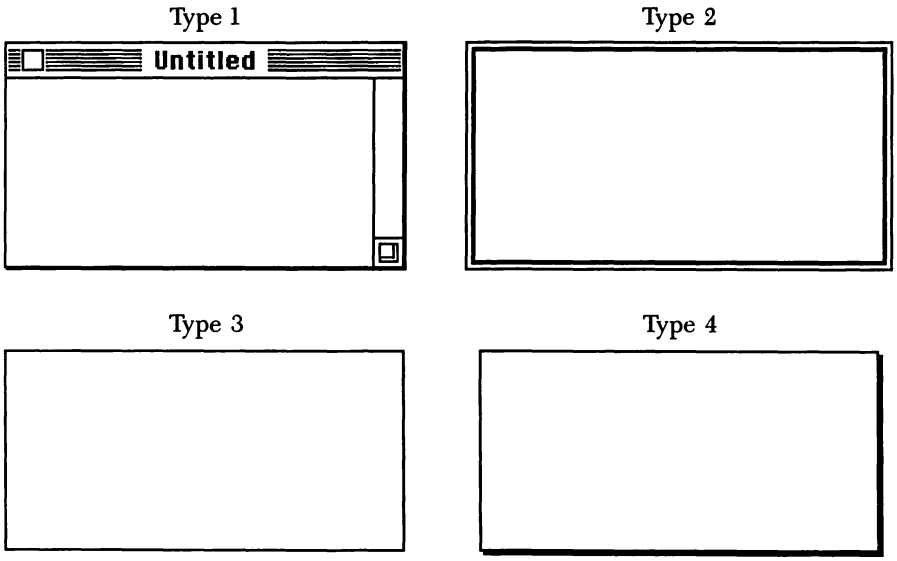


Figure 4-3. Four types of window

The integers 1 through 4 and −1 through −4 are used to specify these window types. Windows with negative type numbers, called modal windows, look just like their positive counterparts, but when a modal window is active, any attempt to select something outside it results in a beep. You can use this feature to force the person running the program to respond to a request for information, rather than select an item from a menu or wander off on some other tangent.

The CALL statement

The CALL statement diverts the flow of a program to a BASIC subprogram or, as in this program, to a machine-language routine.

```
CALL MOVETO (2, 50)
```

This particular machine-language routine—*MOVETO (x,y)*—is one of the group of predefined routines that Apple stored in the Macintosh ROM. These ROM calls are

among those provided to standardize the appearance of application programs (see Chapter 3). Calling these ROM routines from within BASIC is much easier than trying to accomplish the same task with a series of BASIC commands and, since they are written in machine language, they perform their tasks much faster than BASIC could.

The CALL statement is unusual, in that the statement name itself is optional. The two syntaxes of the CALL statement are:

CALL *name* [(*argument list*)]

name [*argument list*]

If you choose to drop CALL, notice that you must also drop the parentheses surrounding the list of arguments. I usually choose the shorter format.

The MOVETO statement

MOVETO positions the Mac's pen—the graphic point used for drawing lines, shapes, and text—at the pixel specified by the (x,y) coordinates given in the argument list: in this case (2,50). This is usually done in preparation for printing text or drawing a line or shape.

When running this program, you will notice that the origin of the coordinate system—that is, the point at which both x and y equal zero—is not at the same spot on the screen for every command that references coordinates. The WINDOW statement, for example, creates a window defined by (x,y) coordinates relative to the upper left corner of the *screen.* However, PRINT statements and other graphic commands (such as MOVETO) reference coordinates that are relative to the upper left corner of the *current output window.* We will experiment with this feature in just a moment.

The PRINT statement

The PRINT statement should be familiar to anyone who has used any version of BASIC. There is nothing new or different about the Macintosh version, which causes the expressions following it to be printed in the current output window, starting at the current pen location. The expressions to be printed can be either text or numbers. Those that are to be printed just as they appear in the PRINT statement, like the ones in our mouse-tracking program, are enclosed in quotation marks.

```
PRINT "Position the pointer in this"
PRINT "window, press the mouse "
PRINT "button, and drag."
PRINT "Triple click to quit."
```

Variables representing text or numbers are listed without quotes, and are automatically replaced by their actual values at the time of printing. We will see an example of this soon.

Multiple expressions, separated by semicolons or commas, can follow a single PRINT statement, as in this line from the mouse-tracking program:

```
PRINT "x="; MOUSE(1), "y="; MOUSE(2)
```

An expression following a semicolon is printed immediately adjacent to the previous expression; an expression following a comma is printed at the beginning of the next comma stop (the comma stop is a position set by the WIDTH statement, which defaults to the width of a string of 14 numbers in whichever font you are using). You can also terminate an entire PRINT statement with a semicolon or comma, causing the next PRINT statement to continue on the same line, after the current statement has been printed.

The WHILE...WEND statements

The WHILE...WEND statements are often referred to as a WHILE...WEND loop. The full syntax for these statements is:

WHILE *expression* [*statements*] WEND

When the program encounters a WHILE... statement, it checks to see if the expression after the WHILE is true. The first such expression evaluated in this program is the following:

```
WHILE MOUSE(0) <> -3
```

If the expression is not true, the program continues with the line after the WEND. If it is true, the statements between WHILE and WEND are executed and the program returns to the WHILE statement and repeats the process until the expression is no longer true.

The MOUSE function

The MOUSE function is actually a group of functions—MOUSE(0) through MOUSE(6)—that make available information about the status of the mouse button and the location of the pointer. This information is trapped by a section of the Macintosh operating system called the *Event Manager.* (The Event Manager also traps information about other activities in your program, such as the most recently selected menu and menu item, and the fact that an inactive window has been clicked.)

Function	Information returned
MOUSE(0)	Returns integer between 3 and −3, depending upon status of mouse button
MOUSE(1) and MOUSE(2)	Return x and y coordinates, respectively, of pointer at moment MOUSE(0) function was last used
MOUSE(3) and MOUSE(4)	Return x and y coordinates of pointer at button-press prior to last use of MOUSE(0) function (starting point of drag)
MOUSE(5) and MOUSE(6)	Return x and y coordinates, respectively, of pointer at moment of last MOUSE(0) *if button was down at that moment;* if button was not down, they return coordinates at which it was last released

The number returned by the MOUSE(0) function ranges from −3 to 3, and is determined by whether the mouse button has been pressed since the last MOUSE(0) and, if so, whether there has been a single, double, or triple click and whether the button is still down. The possible values returned by MOUSE(0) are explained in the following table.

Mouse value	Meaning
0	Button is not down and has not been pressed since last MOUSE(0)
1	Button is not down, but single click has occurred since last MOUSE(0)
2	Button is not down, but double click has occurred
3	Button is not down, but triple click has occurred
−1 to −3	Same as 1 through 3, except that button is still down (in middle of drag)

So, what the WHILE statement in our program essentially does is allow the main body of the program to run as long as the mouse is not triple clicked. When a triple click is detected, program flow continues with the statement after the WEND that is paired with this WHILE: in this case an END statement, which stops the program and returns you to BASIC.

NOTE: You have to click very rapidly in order to register a triple click. If you click too slowly, BASIC interprets it as a single click and acts accordingly. (We will develop a routine, about two programs down the line, that gets around this problem.)

Nested WHILE...WEND statements

Within the outside WHILE...WEND loop in our program is another WHILE... WEND loop. The program cycles through the inner loop as long as MOUSE(0) is equal to −1; that is, as long as the mouse button is held down after a single click.

```
WHILE MOUSE(0) = -1
  CALL MOVETO (10, 20)
  PRINT "x="; MOUSE(1), "y="; MOUSE(2)
WEND
```

The two lines in this loop—a MOVETO statement and a PRINT statement—move the pen to the upper left part of window #2 and print the x and y values of the pointer position at the last MOUSE(0). Recall that MOUSE(1) and MOUSE(2) return the x and y coordinates of the pointer the last time the MOUSE(0) function was used. You could

substitute MOUSE(5) and MOUSE(6) for MOUSE(1) and MOUSE(2) in this application, since they return the current (x,y) position if the button is down, and this loop is executed only if the button is down. The WEND statement then returns program flow to the WHILE at the beginning of the loop. If the mouse button is still down, the program cycles through the MOVETO and PRINT statements again. If the mouse button has been released, the program continues with the statement after WEND: in this case the outer WEND that sends the program back to wait for the mouse button to go down again.

Experimenting with the program

Now that the explanations are out of the way, let's get on to the more interesting stuff. When you ran the program, you saw that it created a background window and a small display window, and that if you pressed the mouse button while the pointer was inside the small window, the program displayed the (x,y) coordinates of the pixel at the head of the pointer. You saw the coordinate display change as you moved the pointer. With a little experimentation, you can discover the origin (0,0) of the coordinate system at the upper left corner of the output window. If you move the pointer to the left of the origin, the x value increases in a negative direction; move it above the origin and the y value does the same. Now move the pointer to the lower left corner of the first line of instructions. You should be able to identify the point (2,50), which is where the MOVETO statement placed the pen before printing.

If you look at the display window, you will see that it has a title bar and size box just like the windows created by Macintosh application programs. And just as in those programs, you can use these features to control the location and size of the window. You move the window by placing the pointer in the title bar and dragging the mouse, and you can change the window's size by either dragging the size box or double clicking in the title bar. (Double clicking enlarges the window to fill the screen; double clicking in the title bar of the enlarged window returns it to its previous size.) As you change the size and location of the window and display new coordinates, notice that they are always relative to the upper left corner of the window.

These housekeeping tasks are taken care of through the special relationship between BASIC and the machine-language ROM calls provided by Apple. All that was required of you was a single command line to create the window. If you have experience

writing programs in BASIC for other computers, you can imagine the commands that would be required to duplicate the features of this window. You may also have noticed a housekeeping task that was *not* totally taken care of by BASIC. The text displayed in the window moves with the window as you drag it, but if you do anything that causes the text to be erased, such as double clicking the title bar to enlarge the window or dragging the size box to make the window smaller than required to show all the text, the text is not redisplayed. BASIC does trap this fact, and later we will learn how to tell when a window needs to be refreshed—that is, its text needs to be redrawn.

Stop the program by triple clicking in the the active window. Choose Show List from the Windows menu and, when the program appears, change the beginning size and location of window #2 (by changing the *rectangle* parameter), and perhaps its *type* (by changing the number at the right end of the statement to 2, 3, or 4). See if you can anticipate where the window will appear when you run the program again, and what it will look like. The keyboard commands that can be used in lieu of choices from the menu are convenient when making quick changes to a program and then checking to see the effect. Pressing Command-period will stop the program, Command-L shows the List window, and Command-R runs the program.

As you experiment with different window types, compare the size of each with the dimensions you specified in the WINDOW statement. Our original window, for example, was 200 pixels wide by 150 high (the ending minus the beginning x and y values). If you check the actual dimensions of the window, you will find that the title bar is *added* on to the top dimension, but the vertical column at the side is included in the window width specified. In addition, the Macintosh menu bar is always displayed in the top 20 pixels of the screen. This explains why the background window (window #1 in our program) has a beginning y coordinate of 38, indicating that its top is 38 pixels down from the top of the screen: The menu bar is 20 pixels high, and the title bar for the window is 18. If you had specified the coordinates *(0,0)–(512,342)* for this window in an attempt to fill the entire screen, the title bar would be totally off the screen and the top two pixels of the window would be behind the menu bar.

As a final experiment, change the *WHILE MOUSE(0) = −1* statement to *WHILE MOUSE(0) = 0*. This causes the program to display the position of the pointer as long as the mouse button is *not* pressed. This may not seem too significant right now, but it will be an important element of the program in the last chapter in this section.

Drawing a Grid

Now that you have a general understanding of how the coordinate system is used to position both text and graphic output on the Macintosh screen, let's whip out a slightly longer program (with a little less explanation) to create the grid shown in Figure 5-1. You can print copies of this grid and use them to help plan your screen displays.

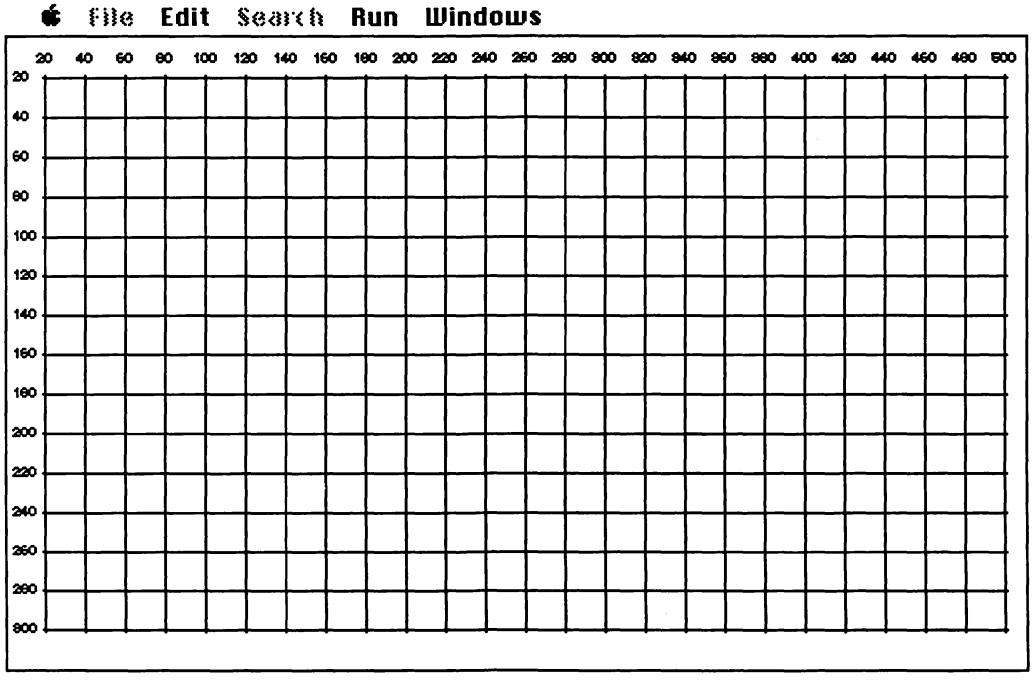


Figure 5-1. A grid for plotting displays

```
'* Drawing a grid
'*

   '*
   '* Define variables.
   '*
   top = 20                                          'top of output window
   left = 20                                         'left side
   bottom = 300                                      'bottom
   right = 500                                       'right side

   '*
   '* Open window and set display font.
   '*
   WINDOW 1, , (0, 20) - (512, 342), 3
   TEXTFONT 4                                        'Monaco font monospaced
   TEXTSIZE 6                                        'small, to get numbers in

   '*
   '* Print numbers down left side and draw horizontal lines.
   '*
   FOR hLine = top TO bottom STEP 20
      MOVETO 0, hLine + 2
      PRINT hLine
      MOVETO left - 1, hLine
      LINETO right + 1, hLine
   NEXT hLine

   '*
   '* Print numbers across top and draw vertical lines.
   '*
   FOR vLine = left TO right STEP 20

      '*
      '* Width function returns number of pixels in string.
      '*
      MOVETO vLine - WIDTH(STR$(vLine)) / 2 - 2, top - 7
      PRINT vLine
      MOVETO vLine, top - 1
      LINETO vLine, bottom + 1
   NEXT vLine
```

Figure 5-2. The grid-drawing program

more...

```
'*
'* Wait for mouse click.
'*
WHILE MOUSE(0) = 0
WEND
END
```

Figure 5-2. The grid-drawing program *(continued)*

The grid program listed in Figure 5-2 first creates a window that fills all the space beneath the BASIC menu bar, then changes the font size to 6-point in order to squeeze the line labels into a small space. The monospaced Monaco font is used to make the labels as easy to read as possible in the small size. It then draws and labels the set of horizontal and vertical lines that represent the imaginary coordinate system we discussed in Chapter 4, and pauses, waiting for you to click the mouse button to signal that it should end. (The final click was thrown in to give you a chance to print an image of the screen by pressing Command-Shift-3—a task we could also do from within the program.) You should recognize many of the statements in this program listing, so let's just review it quickly and then take a closer look at the new commands.

The first section of the program assigns the values of the coordinates that form the boundaries of the grid to *top, left, bottom,* and *right.* The variables *top* and *bottom* are, of course, the minimum and maximum y values; *left* and *right* are the minimum and maximum x values. These variables are used by the ROM calls that draw the boundary and grid lines. We could simply use the coordinate numbers, but attaching names to them makes it a little easier to follow what is going on in the program. More important, if you decide to change a value that is used throughout the program, you need change it only once, in the initial assignment.

It is a good practice to group the variables you will be using throughout the program near the beginning. If you use adequate comments in this section, it provides a handy and easy-to-locate reference to the use of each variable. With versions of BASIC prior to 2.1, there is a more important reason for doing this: These earlier versions store variables in a sequential list, in the order in which they are first encountered in the program. Each time a variable is encountered while running the program, the list

is searched, again sequentially, until the variable is found and its value retrieved, so the speed at which a complex program runs can definitely be affected by where in the list the most commonly used variables are stored. However, starting with version 2.1 of BASIC, variables are accessed by a different method, so this becomes a less important consideration.

The first statement in the grid program is the WINDOW statement, which creates the background for the grid:

```
WINDOW 1, , (0, 20) - (512, 342), 3
```

Since I have defined this as a type 3 window, which does not display a title, there is no point in including a *title* parameter in the statement. You must, however, include the commas to hold its place.

The TEXTFONT statement

TEXTFONT calls a Macintosh ROM routine in the same manner as the MOVETO call used in the last program. Remember that there are two syntaxes for the call statement: CALL *name* [(*argument list*)] and simply *name* [*argument list*]. We used the first syntax in the mouse-tracking program, and will use the second from now on. (The word CALL and the parentheses are optional, but not individually so: You must use both or neither.)

TEXTFONT provides access to the standard Macintosh fonts that you have used in other applications, and to any special or custom fonts you may have loaded onto your disk. The number after TEXTFONT specifies your choice. The font designated with this call will be used for all printing in the current output window. The table on the next page lists the standard fonts available at the time BASIC 2.0 was released.

The system font is the one used automatically when you are working at the operating-system level, as when typing a name under a file icon in the Finder. The application font is the default font for BASIC PRINT statements. In order to use a font, it must, of course, be on your disk. Since storing fonts requires a lot of disk space, you probably won't want to keep all of them on every disk. You can use the Font Mover program on

Font no.	Font
0	System font (Chicago)
1	Application font (Geneva)
2	New York
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
8	San Francisco
9	Toronto
10	Seattle
11	Cairo

your Macintosh System Disk to rearrange them as needed. (If the font you specify with TEXTFONT is not on your disk, the Macintosh substitutes what it considers the most similar one.) The *TEXTFONT 4* call in our program specifies that printing will be done in the Monaco font.

The TEXTSIZE statement

The TEXTSIZE call sets the point size of the selected font. As you have probably discovered in using other applications, each font has specific sizes in which it looks best. The monospaced Monaco font is readable at 6-point, and this small size will allow us to squeeze our grid labels into the limited space available.

The FOR...NEXT statements

The FOR...NEXT pair of statements is usually called a FOR...NEXT loop. The syntax of these statements is:

FOR *variable* = *x* TO *y* [STEP *z*]

NEXT [*variable*][, *variable*...]

The *variable* in the FOR statement represents a counter, which has an initial value of *x* and a final value of *y*. When BASIC encounters a FOR... statement, it tests the counter to see if it is greater than the final value (unless STEP is negative, in which case it

checks to see if the counter is less than the final value). If the counter is in the proper range, the program executes the statements between FOR and NEXT. When it gets to NEXT, the counter is incremented by the value of *z* and is again tested against the final value (if the optional *z* is omitted, the counter is incremented by 1). This loop continues until the counter exceeds the final value, at which time the program continues with the statement after NEXT.

The FOR...NEXT loop in our program initializes the variable *hLine* to the value of *top*—the y coordinate of the top horizontal line—and increments it by 20 after each loop until it exceeds *bottom*—the y coordinate of the bottom line.

```
FOR hLine = top TO bottom STEP 20
```

Therefore, *hLine* equals 20 on the first pass through the loop, it equals 40 on the second pass, and so on until the 15th pass, when it equals 300. After the 15th pass, *hLine* is incremented to 320, tested against the final value, and found to be greater, so program flow moves on to the line after the NEXT statement. As long as *hLine* is not greater than *bottom,* the statements between FOR and NEXT are executed. We'll look at these statements one at a time.

You are already familiar with the MOVETO ROM call, which positions the pen prior to printing or drawing. In this program one of the arguments we are passing to the routine is the variable *hLine,* which will take on a different value for each pass through the loop.

```
MOVETO 0, hLine + 2
```

This statement positions the pen two pixels *below* the horizontal line that will be drawn in a moment, in preparation for the BASIC statement that prints the line label beside each horizontal line.

The next command, *PRINT hLine,* is pretty straightforward. It simply prints the current value of *hLine,* which is some multiple of 20. Then the *MOVETO left −1, hLine*

call repositions the pen in preparation for drawing the horizontal line. The actual drawing is done by the next statement:

```
LINETO right + 1, hLine
```

LINETO is another ROM call. It draws a line from the current pen position to the coordinates specified in the argument list. On the first pass through this loop, it will draw a line from (19,20) to (501,20).

The statement *NEXT hLine* marks the end of the FOR...NEXT loop. The NEXT statement increments the counter and returns the program to the FOR statement. The computer remembers which NEXT goes with which FOR, so it isn't really necessary to include the variable name *hLine* in the NEXT statement—this is a matter of personal preference; but it is a good idea to include the variable if not doing so could cause confusion, as with embedded loops or long loops where the NEXT statement may be a page or more away from its corresponding FOR.

The WIDTH and STR$ functions

The next section of the program draws the vertical lines of the grid using, with two exceptions, the same commands used to draw horizontal lines. The exceptions are the WIDTH and STR$ functions, which keep the labels centered above their lines.

```
MOVETO vLine - WIDTH(STR$(vLine)) / 2 - 2, top - 7
```

This MOVETO call, like the first one in the previous section, positions the pen prior to printing a line label. However, things become a little more complex here. The labels printed across the top of the grid vary in width (some are two digits wide and some are three), and our grid will look neater and more orderly if each label is centered on its line. To accomplish this you still pass just two arguments—the horizontal and the vertical distances from the origin—to the MOVETO call, but this time you use compound expressions that evaluate to two simple numbers.

NOTE: Compound expressions may look a bit confusing at first. If you are reading, and trying to understand, a program that contains them, their meaning will usually become clear if you work your way through several steps of the program, replacing each variable with its current value. If you are writing a program and want to describe a certain location or value that can only be expressed relative to other values, try first describing the value in English. Then look through your list of BASIC commands for one that will shape the English expression into an expression that the computer can understand.

To center a two- or three-digit number on a vertical line, you want to start printing the number about half its width to the left of the line. In this case, both the number and the position of the line are represented by the variable *vLine,* so the location to start printing will be *vLine* minus the quantity that is half its width (*vLine* minus *vLine* divided by 2). Looking through the list of BASIC commands, you will discover the WIDTH function, which returns the width of a string in pixels. This is fine, except that *vLine* is a numeric variable, not a string. A little more rummaging around will produce the STR$ function, which returns a string representation of a numeric value. Combining the two gives you *WIDTH(STR$(vLine))*, which should be the width of the present value of *vLine* in pixels. Now divide this by 2 and subtract it from the location of *vLine* and you have what turns out to be *almost* the correct spot to start printing. Why almost? Well, if you use this value for the horizontal distance, you will find that the numbers are printed with their centers slightly to the right of each line. This is because BASIC always prints four pixels of white space in front of a number. Going back to your formula and subtracting another two pixels will take care of this. This is the type of problem that is usually solved through experimentation the first few times it occurs; then you start remembering the extra space.

The expression for the distance you want the number to be printed above the grid, *top −7,* is easier to evaluate. Bearing in mind that grid coordinates increase as you move down and to the right, subtracting 7 from the top of the grid moves the print location seven pixel lines *up* from the location passed to the ROM call that drew the horizontal line at the top of the grid. Since the line is drawn with the pixel below the location passed to it, and the number is printed starting at the pixel above the print location, there will be seven pixels of open space between the two.

Ending the program

When the program ends, the Command window, and possibly the List window, appear on the screen, covering parts of the grid. So that this doesn't happen before you are ready for it, the last WHILE...WEND loop pauses the program until you press the mouse button, to give you time to study the grid or to use Command-Shift-3 or Command-Shift-4 to send a copy of the screen display to the printer or to a disk file.

```
WHILE MOUSE(0) = 0
WEND
```

NOTE: There are several ways besides Command-Shift-3 to reproduce a screen display on a graphic printer such as the ImageWriter. The easiest, if your printer is hooked up and ready to print, is to insert an LCOPY statement in your program at the point when the screen will be displaying the image you want to print.

Armed with a printout of this grid and an understanding of the relative coordinates used by the different graphic commands in BASIC, you should now find it easier to plan the layout of your windows, dialog boxes, pushbuttons, and other designs.

More experiments

As you type these programs and experiment with them, bear in mind that routines developed in one program, and even entire programs, can easily be included in other programs. For example, you could replace the WHILE...WEND loop at the end of this program with the *WHILE MOUSE(0) <> −3 ...WEND* loop from the mouse-tracking program. This would allow you to confirm the accuracy of your grid and interpret exact locations between the printed lines. If you do this, change the location at which the coordinates are printed from (10,20) to (30,315) and end the PRINT statement with a semicolon, to prevent a carriage return after the coordinates are printed (a carriage return here would scroll the screen, since you are printing on the bottom line). With the new WHILE...WEND loop the program should look like Figure 5-3.

```
'* Drawing a grid, Version 2
'*

   '*
   '* Define variables.
   '*
   top = 20                                          'top of output window
   left = 20                                         'left side
   bottom = 300                                      'bottom
   right = 500                                       'right side

   '*
   '* Open window and set display font.
   '*
   WINDOW 1, , (0, 20) - (512, 342), 3
   TEXTFONT 4                                        'Monaco font monospaced
   TEXTSIZE 6                                        'small, to get numbers in

   '*
   '* Print numbers down left side and draw horizontal lines.
   '*
   FOR hLine = top TO bottom STEP 20
      MOVETO 0, hLine + 2
      PRINT hLine
      MOVETO left - 1, hLine
      LINETO right + 1, hLine
   NEXT hLine

   '*
   '* Print numbers across top and draw vertical lines.
   '*
   FOR vLine = left TO right STEP 20

      '*
      '* Width function returns number of pixels in string.
      '*
      MOVETO vLine - WIDTH(STR$(vLine)) / 2 - 2, top - 7
      PRINT vLine
      MOVETO vLine, top - 1
      LINETO vLine, bottom + 1
   NEXT vLine
   TEXTSIZE 10
```

Figure 5-3. The grid program with a new WHILE...WEND

more...

```
'*
'* Wait for the mouse button and print coordinates.
'*
WHILE MOUSE(0) <> -3                                    'while no triple-click
   WHILE MOUSE(0) < 0                                   'while button is down
      CALL MOVETO (30, 315)
      PRINT "x="; MOUSE(1), "y="; MOUSE(2)              'x and y coordinates
   WEND

   '*
   '* The button has been released. Go back and wait for it to be
   '* pressed again.
   '*
WEND
END
```

Figure 5-3. The grid program with a new WHILE...WEND *(continued)*

Now that you understand how to specify the location of text and graphics on the Mac screen, let's move on to some programs that let you use your new knowledge.

Transferring a Picture

Chapter 6

An advantage of storing information in common formats, as most applications for the Macintosh do, is that you can easily transfer information between applications. Most versions of BASIC are capable of reading text files created by word-processing, spreadsheet, and database programs, but Microsoft BASIC for the Macintosh goes one step beyond these, by allowing you to control *graphic* information produced in other applications. You can transfer into BASIC any graphic that you can cut or copy to the Clipboard. You can then move, scale, and modify these images just as you would images you created entirely in BASIC. And you can send any graphic created or modified in BASIC to the Clipboard and then paste it from there into other applications (assuming they accept graphics).

Using the Clipboard in this way has its drawbacks. Information stored in the Clipboard is rather transitory, disappearing the next time you cut or copy something else, and using the Clipboard as the direct intermediary between some other application and BASIC can be a bit of a hassle, since you have to load the application, create the picture, copy it to the Clipboard, quit the application, load BASIC, load or type the program to read the Clipboard, and then run the program. If you are interrupted, reset your Macintosh, cut something else to the Clipboard, or mess up in some other manner, you will probably have to go all the way back to the beginning and start over. And you can transfer only one picture each time you go through this process.

A less frustrating method is to use the Clipboard to transfer as many images as you like, one at a time, from the application to the Scrapbook. Since the Scrapbook is stored as a disk file, you can later copy it to the BASIC disk (I'm assuming your BASIC disk is the startup disk), load BASIC and run the program that reads the Clipboard, open the Scrapbook, select an image, and choose Copy from the Edit menu. The selected image is transferred to the Clipboard and brought into BASIC. If you want to

bring in a second image, you simply select and copy again. (By the time this book is released, Apple's Switcher should be a standard feature on everyone's menu, and the hassles just described will live on only in the memories of the "old timers.")

NOTE: You can have only one file named *Scrapbook* on a disk, so if you don't want to lose BASIC's existing Scrapbook file when you copy the Scrapbook from the application to the BASIC disk, rename the existing one *x-Scrapbook*, or *Scrapbook-2*, or something else that you will recognize later. When you have finished transferring the images, you can discard the Scrapbook they are stored in and give *x-Scrapbook* its old name back.

Transferring the picture

To avoid having to rename the currrent Scrapbook file every time you want to transfer a new set of stored images onto your BASIC disk, you can store each image in a file of its own. You bring a picture into BASIC from a file where BASIC has previously stored it the same way you bring a picture in from the Clipboard. Both storage areas are treated as sequential files, so you use the same commands to get the information; the only difference is whether you open the Clipboard or the file for input. In this chapter we will write a short program that brings an image in from the Clipboard and writes it back out to a disk file. Then in Chapter 7 we will develop a more substantial program that brings an image in from a disk file and allows you the flexibility of copying, moving, and scaling it.

Figure 6-1 lists the first program, which asks you for a file name, then brings a picture in from the Clipboard and stores it in that file. After typing this program, run it and follow the instructions. You are told to copy an image to the Clipboard from the Scrapbook, and then asked to provide a name for the file in which it will be stored. I used the picture of a robot that seems to be a standard fixture in the Scrapbook of most applications (though my robot has developed romantic interests, the clandestine handiwork of my teenage daughter, who recently developed a fascination for MacPaint). The picture you copied to the Clipboard appears in an output window (Figure 6-2) and when you quit BASIC, you will discover that you now have a new icon bearing the file name you assigned.

Although this program is short, it shows off several commands that are unique to the Macintosh. Let's take a look at the ones we haven't encountered.

```
'* Transferring a picture
'*

   '*
   '* Clear screen.
   '*
   CLS

'*
'* Tell user what to do.
'*
Start:
   WINDOW 2, , (100, 50) - (350, 170), 2
   PRINT "Copy a picture from the Scrapbook"
   PRINT "and then click OK."
   PRINT "Click Quit to return to BASIC."
   BUTTON 1, 1, "Quit", (20, 85) - (80, 105)
   BUTTON 2, 1, "OK", (175, 85) - (235, 105)

   '*
   '* Wait until button clicked.
   '*
   WHILE DIALOG(0) <> 1
   WEND
   butSel = DIALOG(1)
   IF butSel = 1 THEN END
   WINDOW CLOSE 2

   '*
   '* Transfer picture.
   '* Open Clipboard in preparation for bringing in picture
   '* previously placed there.
   '*
   OPEN "clip:picture" FOR INPUT AS #1
   image$ = INPUT$ (LOF(1), 1)                    'bringing in image
   CLOSE 1
   IF image$ = "" THEN GOTO Start
```

Figure 6-1. The picture-transferring program

more...

```
'*
'* Open output file to store image in.
'*
filename$ = FILES$(0, "Store image in file:")
IF filename$ = "" THEN END
OPEN filename$ FOR OUTPUT AS #2
PRINT #2, image$                                    'storing image in file
CLOSE 2                                             'closing file
PICTURE (50, 50) - (200, 200), image$               'displaying image
BUTTON 1, 1, "Continue", (400, 250) - (460, 280)
WHILE DIALOG(0) <> 1
WEND
GOTO Start
```

Figure 6-1. The picture-transferring program *(continued)*

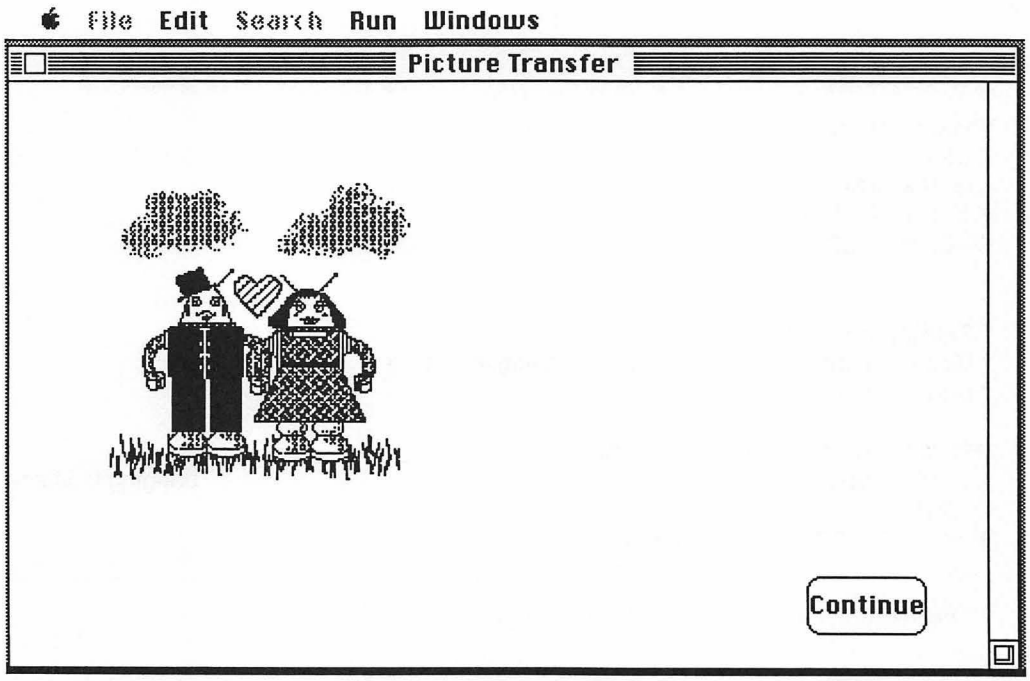


Figure 6-2. The robot from the Scrapbook, with a friend

The CLS statement

The program starts by clearing the screen and using the now-familiar WINDOW statement to open a small window, where the instruction to copy a picture to the Clipboard is displayed. This is the first time we have used the CLS clear-screen statement, which clears the contents of the current output window and positions the pen in the upper left corner. On the Macintosh, this command affects only the current output window, and does not erase edit fields or buttons in the cleared window.

The BUTTON statement

At the bottom of the window are two buttons: a Quit button and an OK button. Here are the statements that create them:

```
BUTTON 1, 1, "Quit", (20, 85) - (80, 105)
BUTTON 2, 1, "OK", (175, 85) - (235, 105)
```

The Quit button is used to signal to the program that you are through transferring pictures and would like to return to BASIC. Clicking the OK button indicates that you are satisfied with the file name you have entered, and the program can transfer the picture and store it in that file.

Toward the end of the program, after the picture has been brought into the program and displayed on the screen, another button, called Continue, is created and displayed in the lower right corner of the screen.

```
BUTTON 1, 1, "Continue", (400, 250) - (460, 280)
```

Clicking this button, after you have taken a moment to scrutinize the picture just brought in, returns you to the beginning of the program.

Buttons are pretty routine items in Macintosh applications, but this is the first time we have created our own in a BASIC program. The syntax of the BUTTON statement is similar to that of the WINDOW statement:

BUTTON *ID, state* [*, title, rectangle*[*, type*]]

Except for *state,* each of these parameters is used like its counterpart in the WINDOW statement. For instance, *ID* is a number greater than zero that you assign to identify the buttons in a window. Buttons are usually numbered consecutively, starting with 1, but this isn't required. Since there is no practical limit to the number of buttons you can have in a window and the numbering system for each window is separate from that of other windows, the same numbers can be used in more than one window.

The *state* is a number from 0 through 2 that indicates the current status of the button, as follows:

State	Button condition
0	Inactive and dimmed on screen
1	Active, but not currently selected
2	Active and currently selected

The *title* is the text associated with the button (in this case Quit, OK, and Continue), and *rectangle* refers to the screen coordinates within the current window.

There are three button *types* available: Figure 6-3 shows them in each of their possible states. Although you can use each of the button types for any task you would like, there are some accepted standard usages. Button type 1 is normally used when you want the user to select an action such as Quit, Run, Cancel, OK, and so on. Button type 2 is used to select one or more options from a list. Button type 3 is used to select one item from a list of mutually exclusive items; that is, each time a new item is selected, the previously selected item should be deselected (we will soon see how a button is deselected).

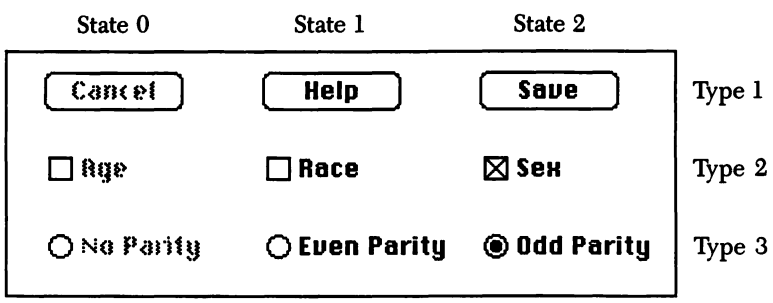


Figure 6-3. Buttons in all possible type/state combinations

The DIALOG function

Once the buttons are displayed, the program has to have a way of knowing when one is clicked. This is one of several events trapped by the DIALOG function, in a manner similar to that by which the MOUSE function traps mouse events. DIALOG(0) returns a number from 0 through 7 indicating the kind of event trapped, and the DIALOG(1) through DIALOG(5) functions return more information about specific events. The number returned by a DIALOG function is reset to 0 each time it is read, so it must be stored in a variable in order to be used later. We will look into the DIALOG function in more detail in Section III; in this program we use only DIALOG(0) and DIALOG(1).

```
WHILE DIALOG(0) <> 1
WEND
butSel = DIALOG(1)
IF butSel = 1 THEN END
```

DIALOG(0) returns a 1 if a button is clicked in the active output window (otherwise it returns a 0), so the *WHILE DIALOG(0) <> 1...WEND* loop causes the program to pause until a button is clicked.

DIALOG(1) returns the ID of the most recently clicked button, so setting the variable *butSel* equal to DIALOG(1) after a button is clicked stores the number of that button in *butSel. butSel* is then tested and the program either ends or continues, depending upon its value. If the program continues, it closes window #2 and opens the Clipboard for input.

The OPEN statement

The OPEN statement is used to associate a file number with the device (screen, keyboard, printer, Clipboard, or communication port) or file name that information is going to or from. All subsequent statements dealing with that device or file refer to it by the assigned file number, which can be any integer or integer expression with a value from 1 through 255.

There are two ways to express the OPEN statement. Both provide the same information but in a different order:

OPEN *mode*, [#]*filenumber*, *filespec*[,*file-buffer-size*]

OPEN *filespec* [FOR *mode*] AS [#]*filenumber* [LEN = *file-buffer-size*]

The two formats are interchangeable, though the actual words used in each may differ, depending upon what you are opening and why. We will discuss the variations as we use them, but for now I will simply explain the OPEN statements used in this program.

```
OPEN "clip:picture" FOR INPUT AS #1
```

The first OPEN statement uses the second format. The *filespec* "clip:picture" indicates which way you wish to open the Clipboard. There are three possible ways:

filespec	Use
"clip:"	Transferring tabular data such as spreadsheets
"clip:text"	Transferring text to and from word processors
"clip:picture"	Transferring graphic images

The OPEN statement's *mode* parameter is INPUT. The words "input" and "output" are relative to the controlling program, not to the file or device—that is, opening for input means opening a file to input data *to* the program *from* the file. The file number assigned to this file is *#1*.

```
OPEN filename$ FOR OUTPUT AS #2
```

This statement uses the same format as the one that opened the Clipboard for input. When a file is opened for output, data is sent *from* the program *to* the disk file. If you open a nonexistent file for output, a new file with the specified name is automatically created.

Disk files use two storage formats: sequential and random access. The files we create with this program are sequential files, which means that the data is stored sequentially, just as it is read in, and can be accessed only in the same manner.

The INPUT$ and LOF functions

The next line in the program uses two functions to bring the data from the Clipboard and assign it to the string variable *image$*.

```
image$ = INPUT$ (LOF(1), 1)
```

The general format of these two functions is:

INPUT$ (X[, [#]*filenumber*])

LOF(*filenumber*)

The *X* argument in INPUT$ stands for the number of characters to be read from the file referenced by *filenumber*. The function *LOF(1)*, which replaces *X* in this program, returns the number of characters in file #1, so the combination says "read however many characters there are in file #1, from file #1." The characters that are read in are stored as a single string variable (string variables can be up to 32,767 characters long) called *image$*.

The CLOSE statement

The CLOSE statement dissociates the file number from the file or device it was associated with by the OPEN statement.

```
CLOSE 1
```

Having dissociated file #1 from the Clipboard, you will be able to reuse the same number with a different file, or you can reopen the original file with the same or a different number and mode.

The FILES$ function

After the file is opened and its contents are stored in *image$*, *image$* is checked to make sure it contains something—there's no point in filling files with blank pictures. Since the instructions offered you the option of quitting, if you continued without putting something in the Clipboard, the program assumes you didn't understand and returns to the beginning of the routine and displays the instructions again. If there is a picture in the Clipboard, the program solicits a file name under which to store it, by displaying a dialog box (shown in Figure 6-4) similar to the one displayed when you choose Save As... from the File menu. The command line in our program that asks the user to name the file in which the image brought in from the Clipboard will be stored is:

```
filename$ = FILES$(0, "Store image in file:")
```

There are two variations of the FILES$ function: One gets the name of a file to open and the other, as you have seen in this program, asks you for a name under which to store a file. The general format for the two is:

FILES$(*n*[, *prompt-string*])

The argument *n* can be either 0 or 1. If it is 0, the Save-type dialog box is displayed and the function returns either the file name entered by the user, or, if the Cancel button is clicked, a null string (a string of zero length). If *n* is 1, a dialog box

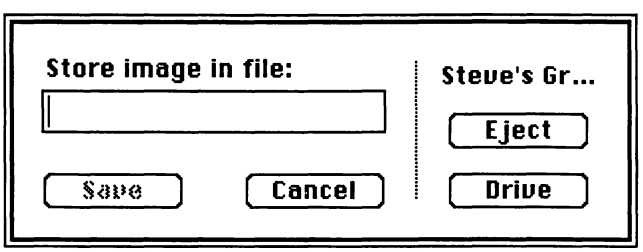


Figure 6-4. Dialog box for soliciting file name

similar to the standard Macintosh Open dialog box is displayed. We will take a closer look at FILES$(1) in the next program. The *prompt-string* parameter has a different purpose in each variation. For FILES$(0), the text you enter as *prompt-string* appears in the dialog box to prompt the user; for FILES$(1), this string can be a list of the types of files you would like listed in the mini-finder, from which the user can select.

The IF...THEN...ELSE statement

After asking the user for a file name and assigning the response to the variable *filename$*, the program must decide what to do next.

```
IF filename$ = "" THEN END
```

This line uses the IF...THEN...ELSE statement to test whether a file name was entered in response to the previous line, and to quit if none was. The syntax for this statement is:

IF *expression* THEN *then-clause* [ELSE *else-clause*]

The statement first evaluates the expression after IF. If it is true, the program executes the *then-clause.* If the expression is not true, the program executes the optional *else-clause,* if present; otherwise it continues with the next command line, the *OPEN filename$ FOR OUTPUT AS #2* statement already discussed.

A new kind of PRINT statement

I mentioned that the PRINT statement has many variations. When followed by a file number, as it is here:

```
PRINT #2, image$
```

it prints whatever follows the comma to the specified file. The picture you stored in *image$* was stored as a series of numbers that tells the Macintosh which pixels to darken on the screen in order to reproduce the picture. This PRINT statement stores

image$ in the file you opened as file #2, so that at some other time the image can be read back into the computer and displayed on the screen. After the string is printed to file #2, the file is closed.

The PICTURE statement

The program has actually done all the work of bringing the image in from the Clipboard and storing it in the disk file; I threw in the next statement just to prove that something actually happened while the disk drive was whirring.

```
PICTURE (50, 50) - (200, 200), image$
```

The PICTURE statement draws the picture stored in *P$* (in this case *image$*) within the rectangular space defined by the coordinates (x1,y1) and (x2,y2), scaling it as necessary to fit. The generic syntax for this statement is:

PICTURE [(*x1,y1*) [- (*x2,y2*)]][, P$]

You may want to take a moment to look at the picture just brought in, then click the Continue button displayed in the lower right corner of the screen to return to the beginning of the program.

Bringing your picture back from the disk file you just stored it in is no more difficult than bringing it in from the Clipboard. The program in the next chapter not only brings your picture back to life; it also allows you to move it around the screen and change its size.

Manipulating a Picture

Chapter 7

The program in this chapter retrieves the picture you stored in a disk file in Chapter 6 and again displays it in the output window. You then have three options: You can click once and drag across an area of the window, to select an image to be copied or moved. You can click twice and drag, to reproduce a previously selected image in the scale of the new area you just dragged over. Or you can click three times and drag, to cause the selected image to follow the pointer around the window.

Because this program is longer and more complex than our previous ones, I will discuss it in five sections: bringing in the picture, branching on a click, and the three options. The full program is listed in Figure 7-9, at the end of this chapter.

Bringing in the picture

The first section of the program, shown in Figure 7-1, is very similar to the program in Chapter 6. It defines all variables as integers, dimensions an array, creates two windows, prints instructions, brings the picture in from the disk file, displays it, and then sets the pen mode for future graphic calls. This section contains two new statements (*DEFINT a-z* and *DIM pict(3000)*), one new ROM call (*PENMODE 10*), and one new function (*FILES$(1, "TEXT")*).

The DEFINT statement

DEFINT is one of a group of four statements that are used to declare variables as integers, single- or double-precision numbers, or strings. The other three statements are DEFSNG, DEFDBL, and DEFSTR. The syntax for the entire group is:

STATEMENT-NAME *letter-range*

```
'*
'* Bring in picture.
'*
CLS                                                    'clear screen
DEFINT a - z                                           'integers are faster
DIM pict(3000)                                         'space to store picture
WINDOW 1, , (0, 20) - (512, 342), 3                    'open display window
WINDOW 2, , (10, 220) - (500, 340), 3
PRINT "                              INSTRUCTIONS"
PRINT "Select a picture file saved by picture-transferring program."
PRINT "Use the following mouse actions to manipulate the picture:"
PRINT
PRINT "Single click and drag selects an area to work with."
PRINT "Double click and drag copies and scales selected area to new rectangle."
PRINT "Triple click and drag moves selected area.";
filename$ = FILES$(1, "TEXT")                          'which file to open
IF filename$ = "" THEN END                             'quit if no file
OPEN filename$ FOR INPUT AS #1                         'open file we stored picture in
image$ = INPUT$ (LOF(1), 1)                            'bring in picture
CLOSE 1                                                'close file

'*
'* Define picture's boundaries.
'*
top = 50
left = 50
bottom = 200
right = 200
WINDOW 1
PICTURE (top, left) - (bottom, right), image$
PENMODE 10                                             'write to screen in XOR mode
```

Figure 7-1. Bringing in a picture from a disk file

So, in this program the statement *DEFINT a-z* tells BASIC to treat all variables beginning with all letters from a through z as integers.

Numeric variables should be declared as integers unless they absolutely have to have the higher precision offered by the other types. This is because integers require less memory to store and can be handled faster. If you don't specifically declare the variables, they default to double precision in the decimal version of BASIC and to single precision in the binary version.

A type-declaration character (%, !, #, and $ for integer, single-precision, double-precision, and string variables, respectively) can be used within a BASIC statement to override this blanket declaration. This enables you to make a general declaration about variables at the beginning of a program and then make specific exceptions if circumstances warrant it.

The DIM statement

An array variable, also called a subscripted variable, is a group of related variables that have been gathered together under a common name. Each element in the array is identified by a subscript (in parentheses) added to the variable name. For example, if I wanted to assign a list of years to variable names, I could use the subscripted variable *year(n)*, and assign 1980 to *year(0)*, 1981 to *year(1)*, and so on.

The dimension (DIM) statement lists the arrays that will be used in the program and specifies the maximum value of the subscript for each, using the general format:

DIM *subscripted-variable-list*

If there is only one subscript listed after the array variable, the array is said to be "one-dimensional." An array can have up to 255 dimensions, though actually using more than four dimensions is unusual. The array in this program is one-dimensional; we will use a two-dimensional array in the next program.

It is not absolutely necessary to dimension an array in order to use a subscripted variable; if you don't, the maximum value of the subscript simply defaults to 10. However, if you do dimension an array, you must do so before the first time you reference one of its elements. To ensure that this is the case, the DIM statement is usually placed at the beginning of the program. A second reason for placing it there is that you don't want the program to flow past it a second time: Attempting to dimension an array a second time, or attempting to dimension a variable that has already been referenced (and thereby defaulted to a maximum subscript of 10) will cause a "Duplicate definition" error and stop the program.

This program lists only one array, using the statement *DIM pict(3000)*. The number in parentheses after the variable name is the largest subscript that may be used with that variable. However, unless you specify otherwise with the OPTION BASE statement, the lowest subscript is 0, so you can actually have one more variable in an

array than the value given in the DIM statement—a fact that occasionally confuses people. (Programmers often avoid this confusion by simply not assigning a value to the zero-subscripted variable, though this is a slight waste of memory space.)

The DEFINT statement has defined all variables as integers, which each require two bytes of memory for storage, so the integer array *pict(3000)* sets aside 6002 bytes of memory to hold the section of the output window we will later select by single clicking and dragging over it. I will explain how I came up with the number 3000 in a moment; for now let's skip to the next new command.

A new version of FILES$

A few lines down we encounter a new version of the FILES$ function. FILES$(1), used to determine which file you want to open in this statement:

```
filename$ = FILES$(1, "TEXT")
```

is the alternate form of the function used in the last chapter's picture-transferring program to get the name of a file in which to store the picture. The four lines following this one, familiar from the previous program, open the file, bring in the picture, and close the file again.

The PENMODE call

Having assigned values to a few variables and displayed the image with a PICTURE statement, the program uses a PENMODE ROM call (without the optional CALL statement and parentheses) to determine the effect of subsequent graphic calls on existing images and background patterns in the output window. The PENMODE call has the following format:

PENMODE *mode*

and there are eight modes, numbered 8 through 15. For now, we need to worry about only two of them. Mode 8, the default mode, copies the new pattern on top of any existing pattern, each pixel of the new pattern replacing the corresponding pixel of the

old pattern at that location. Mode 10, specified in this program with the simple statement *PENMODE 10,* XORs the pixels of the new pattern with those of the old, inverting each pixel of the old pattern that is covered by the new pattern. If terms like *XOR* are new to you, don't worry: We will play with modes after we get the program running.

Branching on a click

After the image is displayed on the screen, the program goes into the loop shown in Figure 7-2, where it stays until there is a single, double, or triple click of the mouse button. The program is then immediately diverted to the subroutine appropriate to the number of clicks.

The WHILE...WEND loop simply holds the program at that point until the mouse button is pressed (remember that MOUSE(0) returns a negative number when the button is held down). Once the button is pressed, the FOR...NEXT loop is used as a delay. The program should branch to one of three subroutines, based on the number of clicks, but BASIC is so fast that it is often impossible to get the second click in before the program branches to the subroutine for a single click. This delay loop gives the

```
'*
'* Branch on click.
'*
Loop:

   '*
   '*Wait for mouse click and drag.
   '*
   WHILE MOUSE(0) > -1
   WEND
   FOR pause = 1 TO 2000
   NEXT
   IF MOUSE(0) = -1 THEN GOSUB GetPicture          'single click and drag
   IF MOUSE(0) = -2 THEN GOSUB PutPicture          'double click and drag
   IF MOUSE(0) = -3 THEN GOSUB MovePicture         'triple click and drag
   GOTO Loop
```

Figure 7-2. Branching on a single, double, or triple click

user time (about one second) to get all the clicks in before the decision is made where to go. (After you have the program running, remove the delay to see the difference.) This is all familiar territory, but the next set of statements introduces something new.

The GOSUB...RETURN statements

The three IF...THEN loops use GOSUB statements to branch to the subroutine appropriate to the number of clicks. Each subroutine must be identified with either a line number or a label. Once called, the subroutine is in control until a RETURN statement is encountered, at which time control returns to the statement following the most recent GOSUB, or to the line or label optionally specified after the RETURN. This is the generic syntax of the GOSUB statement:

GOSUB *line* ... RETURN [*line*]

The IF...THEN statements are mutually exclusive, so after executing the appropriate subroutine, the program returns to the *GOTO Loop* statement, which sends it back to the label at the top of the loop. However, if you learned BASIC from someone who rapped your knuckles with a ruler every time you wrote a GOTO, and you just can't get over your aversion to them, this GOTO could be replaced by enclosing the entire branching loop in another WHILE...WEND loop—perhaps something like *WHILE 1=1...WEND.* I'll leave the GOTO haters to find their own solution and move on to discuss the three subroutines.

The *GetPicture* subroutine

The section of the program shown in Figure 7-3 does three things. First, as you drag the mouse, the program draws a rectangle from the location of the pointer when you clicked to its current location. Second, when you release the button, the program stores the image enclosed by the rectangle in the integer array *pict(3000).* Third, before returning, it sets the flag variable *lastAction* to −1 (a logical *true*) to indicate to subsequent routines that a single click was the previous action.

The segment of the program within the WHILE...WEND loop draws, erases, and redraws the selection rectangle as long as MOUSE(0) returns a value of −1 (meaning that the button is still down after a single click). To do this, it calls two other subroutines and uses a couple of new statements and ROM calls.

```
'*
'* Single click and drag will drag out rectangle and, when mouse
'* button is released, store enclosed picture.
'*
GetPicture:
   WHILE MOUSE(0) = -1
      GOSUB GetRectangle

      '*
      '* Draw and erase frame from starting to ending points.
      '*
      FRAMERECT VARPTR(boundary(0))
      FRAMERECT VARPTR(boundary(0))
   WEND

   '*
   '* Use final set of coordinates to define rectangle enclosing
   '* picture we will store.
   '*
   GOSUB Reassign

   '*
   '* Store picture in pict array, and then return to await
   '* next click.
   '*
   GET (left, top) - (right, bottom), pict
   lastAction = -1
   RETURN
```

Figure 7-3. Selecting an image with *GetPicture*

The *GetRectangle* subroutine

Each time through the loop, the *GetRectangle* subroutine is used to assign the starting and ending coordinates of the drag to the appropriate elements of an integer array called *boundary*. The listing in Figure 7-4 on the following page shows this subroutine.

GetRectangle first assigns the current values of MOUSE(4), (3), (6), and (5)—in that order—to the integer array variables *boundary(0)* through *boundary(3)*. Two IF...THEN statements then compare the starting and ending points to see whether

```
'*
'* Retrieve starting and ending x and y coordinates
'* of mouse-drag. These will be used to draw rectangle on screen.
'*
GetRectangle:
   boundary(0) = MOUSE(4)                          'starting y coordinate
   boundary(1) = MOUSE(3)                          'starting x coordinate
   boundary(2) = MOUSE(6)                          'ending y coordinate
   boundary(3) = MOUSE(5)                          'ending x coordinate

   '*
   '* If dragging down or left, swap appropriate coordinates so
   '* ending coordinate is always larger than starting coordinate.
   '*
   IF boundary(0) > boundary(2) THEN SWAP boundary(0), boundary(2)
   IF boundary(1) > boundary(3) THEN SWAP boundary(1), boundary(3)
   RETURN
```

Figure 7-4. Getting the mouse-drag coordinates with *GetRectangle*

you are dragging either left or up, in which case the starting and ending coordinates are swapped. (The SWAP command exchanges the values of the two variables listed after it.) This is necessary (if the drag is not down and to the right) because the ROM call that draws the rectangle expects *boundary(0)* to specify the top of the rectangle, *boundary(1)* the left edge, *boundary(2)* the bottom, and *boundary(3)* the right edge.

The FRAMERECT call

After control returns from *GetRectangle,* the FRAMERECT ROM call draws a rectangle in the current output window, using the current height, width, pattern, and mode of the pen. Before invoking this ROM routine, you must store the value of the top, left, bottom, and right edges of the rectangle in an array, as we just did in the *GetRectangle* subroutine.

The function VARPTR(*array(n)*) returns the address in memory of the specified element of the array, so in the first FRAMERECT call:

```
FRAMERECT VARPTR(boundary(0))
```

VARPTR returns the address of *boundary(0)* so FRAMERECT can read that element and the next three and use them to draw the rectangle.

The FRAMERECT ROM call is one of a group of similar calls that draw, erase, fill, paint, and invert rectangles, ovals, and other shapes. All require that you use this same method of storing the edges of the shape in an array; they then reference the array with the VARPTR function.

The second time FRAMERECT is called in each pass through *GetRectangle*'s WHILE...WEND loop, a second rectangle is drawn on top of the first. Since we have specified PENMODE 10 (the inversion mode) for the pen, the second drawing inverts the colors of the pixels of the first, making it disappear. The effect of this is to drag out a shimmering rectangle as you move the mouse. When you release the mouse button, MOUSE(0) is no longer −1, so the WHILE condition is not satisfied and the program continues with the line after the WEND.

The *Reassign* subroutine

The statement immediately after the loop is another GOSUB directing the program to the *Reassign* subroutine shown in Figure 7-5.

This subroutine assigns the current values of *boundary(0)* through *boundary(3)* to the variables we are using to identify the top, left, bottom, and right edges of our

```
'*
'* Reassign top, left, bottom, and right boundaries
'* of picture. These boundaries are used to either GET or PUT picture.
'*
Reassign:
   top = boundary(0)
   left = boundary(1)
   bottom = boundary(2)
   right = boundary(3)
   RETURN
```

Figure 7-5. Changing variables with *Reassign*

picture. These four assignment statements are an example, like CALL, of a statement with an optional name. The first statement is actually *LET top = boundary(0),* but the word LET is optional.

The GET statement

The program returns from *Reassign* to the GET statement in the *GetPicture* subroutine. GET has two totally different forms: a random-file GET and a screen GET. The screen GET used here (we'll discuss the random-file GET in another chapter) records the condition of each pixel (on or off) within the area defined by the (x,y) coordinates of the upper left and lower right corners of a rectangle. This information is stored in the array specified after the coordinates in the GET statement:

GET (x1,y1) - (x2,y2), *array* [(*index*[, *index*..., *index*])]

In this case the coordinates of the rectangle are specified by the variables *left, top, right,* and *bottom,* which are the starting and ending points of your drag.

GET (left, top) - (right, bottom), pict

This image is stored in the *pict* array, which we dimensioned to a maximum subscript value of 3000 at the beginning of the program. I chose the number 3000 pretty much through experimentation: This is a large enough array to hold the size rectangle I usually drag out. If you drag out a rectangle too large for the array, the program will crash with an "Illegal function call" error message, in which case you will want to dimension *pict* to a larger value. If you know the size of the largest rectangle you will use, the following formula will tell you exactly how big the array must be:

$$4+(((y2-y1)+1)*2*\text{INT}(((x2-x1)+16)/16))$$

The values $y2-y1$ and $x2-x1$ represent the right minus the left edge and the bottom minus the top of your proposed rectangle. The formula returns the number of bytes of storage the rectangle will require. Since an integer array allocates two bytes per element, you then dimension the array to hold *half* the value returned by the formula.

Following the GET statement, the variable *lastAction* is set to −1 so that any routine that might follow will know that a single-click action was just performed. Then the RETURN statement sends the program via the *GOTO Loop* statement back to the *Loop* label, to wait for the next single, double, or triple click of the mouse.

The *PutPicture* subroutine

Figure 7-6 lists the routine that the program branches to on a double click. The only differences between this and the previous section are that the WHILE...WEND loop is active as long as the button is held down after a double click (*MOUSE(0)* = −2) and that the PUT statement is used to redisplay the image, rather than the GET statement that originally got it.

The PUT statement

PUT, like GET, has two forms: a random-file PUT, which we will deal with later, and a screen PUT, which redisplays the image stored by the screen GET. The format of the screen PUT is:

PUT (x1,y1) [- (x2,y2)], *array* [(*index*[, *index*..., *index*])][, *action-verb*]

```
'*
'* Double click and drag defines new rectangle, and places stored picture in it.
'* Picture is automatically scaled to fit.
'*
PutPicture:
   WHILE MOUSE(0) = -2
      GOSUB GetRectangle
      FRAMERECT VARPTR(boundary(0))
      FRAMERECT VARPTR(boundary(0))
   WEND
   GOSUB Reassign
   PUT (left, top) - (right, bottom), pict
   lastAction = -2
   RETURN
```

Figure 7-6. Changing the size of the picture with *PutPicture*

Notice the square brackets indicating the optional portions of this statement. You need specify only the (x1,y1) coordinates and the array name if you want to reproduce the stored image in its original size, with the upper left corner of the display area located at the specified coordinates. If you also specify the optional (x2,y2) coordinates of the lower right corner, the image is scaled to fit in the area defined by the two sets of coordinates. As you will see when you run the program, this allows you to change the size and proportions of an image.

The *MovePicture* subroutine

The last section of this program, listed in Figure 7-7, is the one branched to on a triple click. The *MovePicture* subroutine gets the image contained in the last rectangle dragged out after a single click and moves the image around the screen as you move the mouse.

The variables *top, left, bottom,* and *right* are redefined after each drag, so they always contain the boundaries of the most recent rectangle. *MovePict* first checks to see if this rectangle was defined by a single or a double click. If it was defined by a double click, which means that a scaled version of the original image was the last thing drawn, then *MoveRect* redefines *top* and *left* as the current position of the mouse pointer, and displays the original image there. If the current rectangle was defined by a single click, meaning an image was selected, the selected image is displayed on top

```
'*
'* Triple click and drag moves stored picture around window.
'*
MovePicture:
  WHILE MOUSE(0) = -3
    IF lastAction <> -2 THEN PUT (left, top), pict
    left = MOUSE(1)
    top = MOUSE(2)
    PUT (left, top), pict
    lastAction = -3
  WEND
  RETURN
```

Figure 7-7. Moving the picture with *MovePict*

of itself, making it disappear. Then, as long as MOUSE(0) is equal to −3, the picture is drawn with its upper left corner at (*left,top*). *left* and *top* are then redefined as the current mouse coordinates, and the picture is redrawn. After the first image is drawn, the drawing created by the first PUT statement in the loop is always superimposed on top of a previous drawing, making it disappear. The drawing created by the second PUT in the loop redisplays the picture at a new location.

Sending a picture out of BASIC

Having manipulated a picture in BASIC, you may want to transfer it to your word processor or some other application. The program for sending a picture out of BASIC differs very little from the one we developed in Chapter 6 to bring one in. If you use the program in Figure 6-1 to create an image you'd like to transfer to another program, a few lines of code before and after the part that draws the picture will capture it and send it to the Clipboard; from there, you can recover it once you have loaded the other application. The modified program should look like the one in Figure 7-8.

```
'* Transferring a picture
'*

   '*
   '* Clear screen.
   '*
   CLS

'*
'* Tell user what to do.
'*
Start:
   WINDOW 2, , (100, 50) - (350, 170), 2
   PRINT "Copy a picture from the Scrapbook"
   PRINT "and then click OK."
   PRINT "Click Quit to return to BASIC."
   BUTTON 1, 1, "Quit", (20, 85) - (80, 105)
   BUTTON 2, 1, "OK", (175, 85) - (235, 105)
```

Figure 7-8. Transferring an image out of BASIC

more...

```
'*
'* Wait until button clicked.
'*
WHILE DIALOG(0) <> 1
WEND
butSel = DIALOG(1)
IF butSel = 1 THEN END
WINDOW CLOSE 2

'*
'* Transfer picture.
'* Open Clipboard in preparation for bringing in picture
'* previously placed there.
'*
OPEN "clip:picture" FOR INPUT AS #1
image$ = INPUT$ (LOF(1), 1)                          'bringing in image
CLOSE 1
IF image$ = "" THEN GOTO Start

'*
'* Open output file to store image in.
'*
filename$ = FILES$(0, "Store image in file:")
IF filename$ = "" THEN END
OPEN filename$ FOR OUTPUT AS #2
PRINT #2, image$                                     'storing image in file
CLOSE 2                                                    'closing file
PICTURE ON
PICTURE (50, 50) - (200, 200), image$                'displaying image
PICTURE OFF
OPEN "clip:picture" FOR OUTPUT AS #1
PRINT #1, PICTURE$
CLOSE #1
BUTTON 1, 1, "Continue", (400, 250) - (460, 280)
WHILE DIALOG(0) <> 1
WEND
GOTO Start
```

Figure 7-8. Transferring an image out of BASIC *(continued)*

In this program, on the line before the program starts drawing the picture, the PICTURE ON statement has been inserted to start recording screen graphic statements. After the picture is drawn, the PICTURE OFF statement stops the recording.

The PICTURE$ function returns the string containing the image recorded by the last PICTURE ON statement. So the next two program lines open *"clip:picture"* for output and print PICTURE$ to the Clipboard.

NOTE: If you want your picture to appear on the screen during this recording, also insert *CALL SHOWPEN* just before or after the PICTURE ON statement.

That's all there is to it. After running the new program, you can open the Scrapbook and paste the contents of the Clipboard into it. Since you changed the picture's size when you transferred it into BASIC, it will look distinctly different from the original when you transfer it back to the Scrapbook, so you will know you haven't simply pasted back out what you originally copied in.

```
'*Manipulating a Picture
'*

   '*
   '* Bring in picture.
   '*
   CLS                                                    'clear screen
   DEFINT a - z                                           'integers are faster
   DIM pict(3000)                                         'space to store picture
   WINDOW 1, , (0, 20) - (512, 342), 3                    'open display window
   WINDOW 2, , (10, 220) - (500, 340), 3
   PRINT "                              INSTRUCTIONS"
   PRINT "Select a picture file saved by picture-transferring program."
   PRINT "Use the following mouse actions to manipulate the picture:"
   PRINT
   PRINT "Single click and drag selects an area to work with."
   PRINT "Double click and drag copies and scales selected area to new rectangle."
   PRINT "Triple click and drag moves selected area.";
   filename$ = FILES$(1, "TEXT")                          'which file to open
   IF filename$ = "" THEN END                             'quit if no file
   OPEN filename$ FOR INPUT AS #1                         'open file we stored picture in
   image$ = INPUT$ (LOF(1), 1)                            'bring in picture
   CLOSE 1                                                'close file
```

Figure 7-9. The complete picture-manipulating program

more...

```
   '*
   '* Define picture's boundaries.
   '*
   top = 50
   left = 50
   bottom = 200
   right = 200
   WINDOW 1
   PICTURE (top, left) - (bottom, right), image$
   PENMODE 10                                    'write to screen in XOR mode

'*
'* Branch on click.
'*
Loop:

   '*
   '*Wait for mouse click and drag.
   '*
   WHILE MOUSE(0) > -1
   WEND
   FOR pause = 1 TO 2000
   NEXT
   IF MOUSE(0) = -1 THEN GOSUB GetPicture          'single click and drag
   IF MOUSE(0) = -2 THEN GOSUB PutPicture          'double click and drag
   IF MOUSE(0) = -3 THEN GOSUB MovePicture         'triple click and drag
   GOTO Loop

'*
'* Single click and drag will drag out rectangle and, when mouse
'* button is released, store enclosed picture.
'*
GetPicture:
   WHILE MOUSE(0) = -1
      GOSUB GetRectangle

      '*
      '* Draw and erase frame from starting to ending points.
      '*
      FRAMERECT VARPTR(boundary(0))
      FRAMERECT VARPTR(boundary(0))
   WEND
```

Figure 7-9. The complete picture-manipulating program *(continued)*

more...

```
    '*
    '* Use final set of coordinates to define rectangle enclosing
    '* picture we will store.
    '*
    GOSUB Reassign

    '*
    '* Store picture in pict array, and then return to await
    '* next click.
    '*
    GET (left, top) - (right, bottom), pict
    lastAction = -1
    RETURN

'*
'* Double click and drag defines new rectangle, and places stored picture in it.
'* Picture is automatically scaled to fit.
'*
PutPicture:
    WHILE MOUSE(0) = -2
        GOSUB GetRectangle
        FRAMERECT VARPTR(boundary(0))
        FRAMERECT VARPTR(boundary(0))
    WEND
    GOSUB Reassign
    PUT (left, top) - (right, bottom), pict
    lastAction = -2
    RETURN

'*
'* Triple click and drag moves stored picture around window.
'*
MovePicture:
    WHILE MOUSE(0) = -3
        IF lastAction <> -2 THEN PUT (left, top), pict
        left = MOUSE(1)
        top = MOUSE(2)
        PUT (left, top), pict
        lastAction = -3
    WEND
    RETURN
```

Figure 7-9. The complete picture-manipulating program *(continued)*

more...

```
'*
'* Retrieve starting and ending x and y coordinates
'* of mouse-drag. These will be used to draw rectangle on screen.
'*
GetRectangle:
   boundary(0) = MOUSE(4)                          'starting y coordinate
   boundary(1) = MOUSE(3)                          'starting x coordinate
   boundary(2) = MOUSE(6)                          'ending y coordinate
   boundary(3) = MOUSE(5)                          'ending x coordinate

   '*
   '* If dragging down or left, swap appropriate coordinates so
   '* ending coordinate is always larger than starting coordinate.
   '*
   IF boundary(0) > boundary(2) THEN SWAP boundary(0), boundary(2)
   IF boundary(1) > boundary(3) THEN SWAP boundary(1), boundary(3)
   RETURN

'*
'* Reassign top, left, bottom, and right boundaries
'* of picture. These boundaries are used to either GET or PUT picture.
'*
Reassign:
   top = boundary(0)
   left = boundary(1)
   bottom = boundary(2)
   right = boundary(3)
   RETURN
```

Figure 7-9. The complete picture-manipulating program *(continued)*

Generating a Pattern

Chapter 8

In this chapter we will work our way through a program that quickly computes the values required to define a custom-made pattern. Quite a few ROM calls available through BASIC make use of patterns: setting them, drawing lines in them, and painting or filling shapes with them. The process of defining the patterns is easy, once you have done it a few times. However, attempting to figure it out for the first time may be a bit frustrating, so before we get into the program, let's have a look at the general format used in pattern ROM calls, and the type of information you pass to them.

As you know, the Macintosh display is composed of rows and columns of pixels that can be turned on or off: To draw a black line on the screen, the Macintosh simply turns on consecutive pixels in the path of the line. Before a specific pattern can be used, someone has to define which pixels the Macintosh must turn on to create that pattern. The definition encompasses an eight-row by eight-column block of pixels, which is then repeated in a horizontal or vertical direction, as needed, to draw or fill an area. When you draw, fill, or spraypaint (as in MacPaint) with a pattern, the only pixels turned on in the line or fill area are those that correspond to the turned-on pixels in the pattern.

To define a pattern block by hand, you first sketch an eight-by-eight grid and then blacken the squares representing the pixels you want turned on. The sample grid in Figure 8-1 on the next page would produce a gray pattern, since every other pixel is turned on.

Now consider each set of two rows in the grid, starting with the top two, to be a 16-bit binary number, with each black square representing a 1 and each white square a 0. The first row is the left half of this 16-bit number, and the second row is the right half. Figure 8-2 on the following page shows this concept applied to the top two rows of the gray pattern.

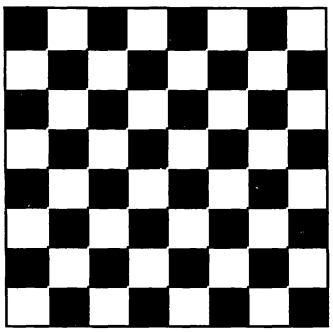


Figure 8-1. A gray pixel pattern

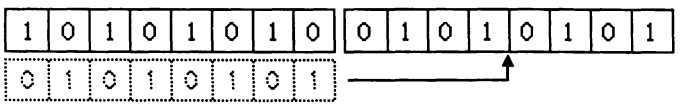


Figure 8-2. Two rows of a pattern grid that make a 16-bit number

If you do the same thing with the other three sets of rows, you will have a group of four 16-bit binary numbers that uniquely defines the gray pattern. You can pass this definition to a Macintosh ROM call so that it can use the pattern while drawing a line or filling an area on its screen. To pass the pattern definition, you store these four numbers in an integer array and include the first element of this array as a parameter of the pattern ROM call, just as you did with the FRAMERECT ROM call in the picture-manipulation program.

Hex numbers revisited

Computers work very comfortably with binary numbers, but we mere mortals find them a little cumbersome: We have to convert these numbers to either decimal or hexadecimal before entering them into an array. There are several methods by which we can make this conversion. Since we are working this out by hand right now, let's use an easy way. Later, in the pattern-generator program, we will let the computer do it another way.

The following table shows the binary and decimal equivalents of hexadecimal numbers 0 through F, for those who are a little rusty on number systems.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

You can convert the 16-bit binary number shown earlier into hexadecimal form by partitioning the binary number into four 4-bit segments and converting each segment to hex, using the table above if necessary. Figure 8-3 shows how the 16-bit number that stores the first two rows of the gray pattern ends up as the hex number AA55.

If this method seems too easy to be true, you can always add up the powers of 2 to convert the binary number 1010101001010101 to the decimal number 43605, and then convert 43605 to hex—but you'll end up with the same number.

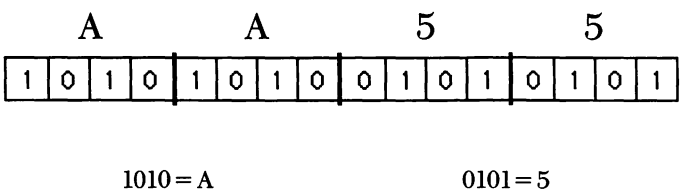


Figure 8-3. Converting a 16-bit binary number to hex

Storing the pattern

The four sets of rows in the gray pattern are identical, so the hex number AA55 can be used to describe each. When you store this number in the computer as an array element, you have to tell the computer that this combination of letters and numbers is a hex number, not a string, by preceding it with the symbol &H. To the computer, &HAA55 is a hex number that is the same as the decimal number 43605.

To store the gray pattern in memory, assign the four &H row-set numbers to four consecutive elements of an integer array (defined as integer with either a DEFINT statement or the % symbol). In this case, the elements of the array could be:

```
pat%(0) = &HAA55
pat%(1) = &HAA55
pat%(2) = &HAA55
pat%(3) = &HAA55
```

Then to use the pattern, point to its first element with the VARPTR function included in all pattern ROM calls. For example, to set the pen pattern to gray, you would use the ROM call:

```
CALL PENPAT (VARPTR (pat%(0)))
```

The ROM calls that create shapes get the top, left, bottom, and right boundaries of the shape from another array, so a single ROM call that both creates and fills a shape will include two VARPTR functions, to pass the address of the first element of each array. For example, to create the filled rectangle shown in Figure 8-4, you would store the boundaries in an array, like this:

```
bound%(0) = 50
bound%(1) = 20
bound%(2) = 200
bound%(3) = 100
```

and then use the ROM call:

```
CALL FILLRECT (VARPTR (bound%(0)), VARPTR (pat%(0)))
```

That is how you do the whole process of setting a pattern by hand. Now let's have a look at a program that shows you what the pattern you are setting will look like, and automatically provides the numbers you need for the array. This program creates the work area shown in Figure 8-5 on the next page. When you click a magnified "pixel" in the grid on the left, the value of that row, in both decimal and hexadecimal, is displayed in the center window and the pattern, in a normal scale, is displayed in the right window. The program goes through essentially the same process we just did by hand, only faster. I will describe the activities in this program in eight sections. The complete program is shown at the end of the chapter in Figure 8-18.

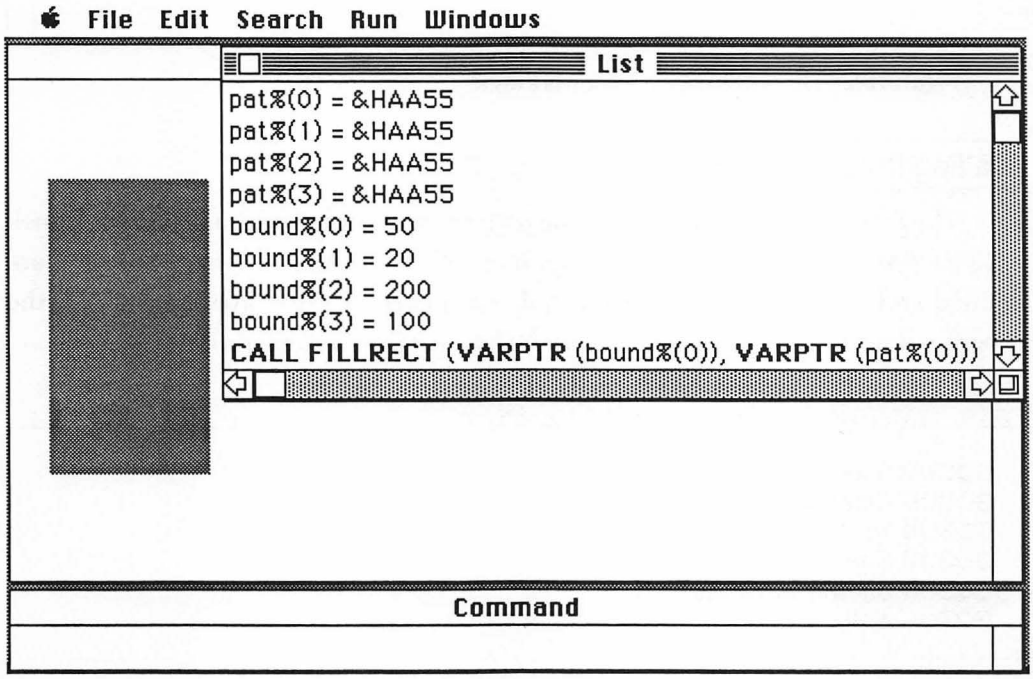


Figure 8-4. A filled gray rectangle

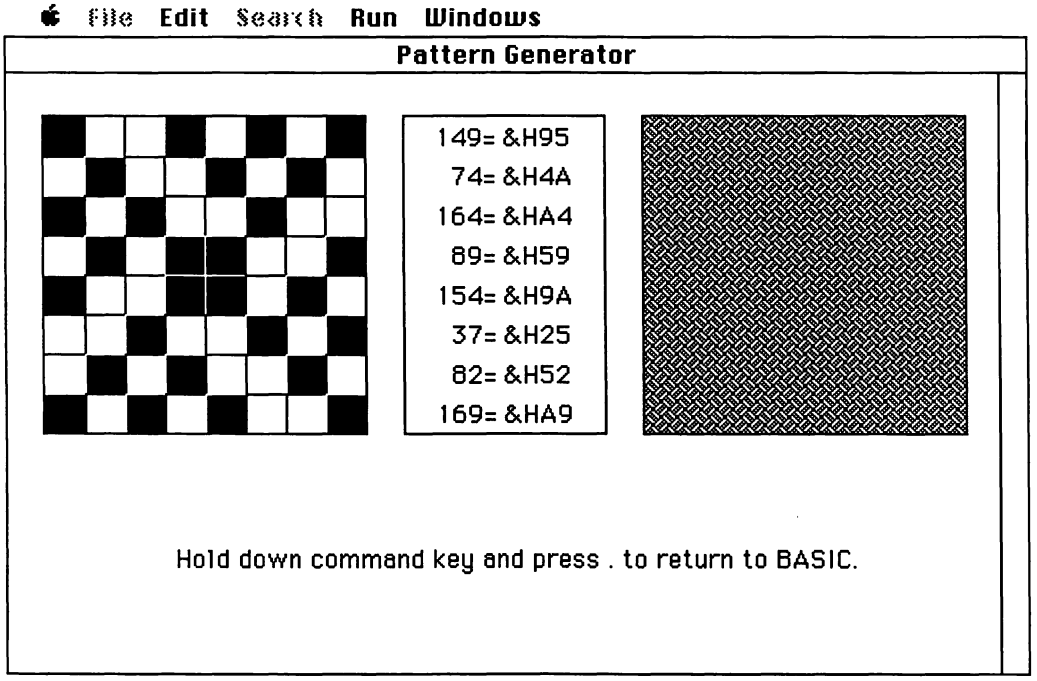


Figure 8-5. The pattern-generator work area

Calling the subroutines

The first section (Figure 8-6) of the pattern-generator program is simply a bunch of GOSUB statements that send the program to other sections, where initial values are defined and the screen display is created. After executing the five subroutines, the program flows into the main loop—*CatchClick*—where it stays until you quit.

```
GOSUB DefineVariables
GOSUB CreateWindows
GOSUB InitializePowers
GOSUB CreateGrid
GOSUB ShowInitialValues
GOTO CatchClick
```

Figure 8-6. Calling the subroutines

Dimensioning arrays and defining variables

The *DefineVariables* subroutine listed in Figure 8-7 dimensions the arrays used by the program and defines some of the more common variables. The pattern-generation program uses five arrays: *bound(4), pattern(3), decimalValue(8), displayWin(3),* and *power(15).* I will explain the contents of each of these arrays when we put something in them.

The variables *wide* and *high* are simply the width and height of the two square windows. The variable *increment* is the width, in pixels, of a row or column. Rather than use constants for these values, which are later used in formulas, I have assigned them to variables to make it easier to understand the computations (this also makes it easier to change them throughout the program should you later decide to modify the display). Moreover, BASIC can mathematically manipulate a value stored as a variable

```
'*
'* Dimension arrays and define many variables used in program.
'*
DefineVariables:
  DEFINT a - z                                    'use integers for speed

  '*
  '* Dimension arrays used by program.
  '*
  DIM bound(4)                                    'edges of grid location
  DIM pattern(3)                                  'pattern array
  DIM decimalValue(8)                             'decimal value of row
  DIM displayWin(3)                               'edges of display window
  DIM power(15)                                   'powers of 2
  wide = 160                                      'width of pattern display window
  high = 160                                      'height of pattern display window
  increment = 20                                  'height of row, width of column
  displayWin(0) = 0                               'top of pattern display window
  displayWin(1) = 0                               'left
  displayWin(2) = 160                             'bottom
  displayWin(3) = 160                             'right
  RETURN
```

Figure 8-7. Dimensioning arrays and defining the variables

faster than it can manipulate the same value given directly as a constant. This, plus the fact that BASIC *2.0* stores, and therefore accesses, variables in the order they appear in the program, should encourage you to use descriptive variables for values to be manipulated mathematically, define all variables at the beginning of the program, and place the ones that require the most rapid access first. (As mentioned earlier, BASIC *2.1* stores variables in a different manner, so with that version the reasons for doing all this aren't so strong.)

The *DefineVariables* subroutine finishes by assigning the coordinates of the top, left, bottom, and right sides of the pattern display area in window #2 to the elements of the *displayWin* array.

Creating windows

The *CreateWindows* subroutine (Figure 8-8) creates four windows and prints an instruction in the first one—nothing new to you. This is a good time to remind you, though, that PRINT statements display their text in the active output window. Placing the PRINT statement after any WINDOW statement other than the first one would cause the instructions to be printed in that window.

```
'*
'* Create four windows used by program. Window #4 is background window,
'* window #3 displays decimal and hexadecimal values of each row,
'* window #2 displays pattern, and window #1 is used to create pattern.
'* Note that window #1 is modal (specified by negative type number),
'* meaning that it is only window in which user can make selection.
'*
CreateWindows:
  WINDOW 4, "Pattern Generator", (0, 38) - (512, 342), 1          'background
  MOVETO 85, 250
  PRINT "Hold down command key and press . to return to BASIC."
  WINDOW 3, , (200, 60) - (300, 220), 3                          'display values
  WINDOW 2, , (320, 60) - (480, 220), 3                          'display patterns
  WINDOW 1, , (20, 60) - (180, 220), -3                          'display pixels
  RETURN
```

Figure 8-8. Creating four windows

The only other thing worth pointing out in this section is the use of −3 as a *type* for window #1. You will recall from our earlier explanation of windows that this creates a modal window, which means that as long as that window is displayed, user input is limited to it—even the menus are inaccessible.

Initializing the *power* array

The *InitializePowers* subroutine, shown in Figure 8-9, fills an array with powers of 2—that is, with the number 2 raised to each of the powers from 0 through 14, plus the negative value −32768 to represent the 15th power of 2, which would exceed the upper limit for an integer variable (32767) if calculated directly. The *power* array will be used to convert the decimal values of the double rows of pixels in the pattern grid to binary numbers, which will then be used to create the pattern.

Creating a grid

The two FOR...NEXT loops in the *CreateGrid* subroutine (Figure 8-10 on the next page) are used to draw a seven-by-seven grid in window #1. This actually forms a grid composed of eight rows by eight columns, since the window frame forms the outer boundary. This grid is used to simulate the pattern being defined.

```
'*
'* Initialize array containing powers of 2.
'* Used to convert from pixels to binary numbers:
'*    power(bit) = 2 ^ bit
'*
InitializePowers:
  FOR bit = 0 TO 14
    power(bit) = 2 ^ bit
  NEXT bit
  power(15) = -32768!
  RETURN
```

Figure 8-9. Initializing the *power* array

```
'*
'* Draw grid used to simulate 8 by 8 array of pixels.
'*
CreateGrid:
  FOR hLine = 1 TO 7                                    'draw horizontal lines
    MOVETO 0, hLine * increment
    LINETO wide, hLine * increment
  NEXT hLine

  FOR vLine = 1 TO 7                                    'draw vertical lines
    MOVETO vLine * increment, 0
    LINETO vLine * increment, high
  NEXT vLine
  RETURN
```

Figure 8-10. Creating the pattern grid

Showing initial values

The short *ShowInitialValues* subroutine in Figure 8-11 simply sets the display conditions at the start of the program, before any pixels have been clicked.

The *decimalValue* array is where the current value of each row is stored. The FOR...NEXT loop selects each row in turn and sets its decimal value equal to zero. The GOSUB statement then diverts program flow to the *ShowNum* subroutine, which makes window #3 active, prints the decimal and hexadecimal values, and then reactivates window #1. We'll leave a more detailed discussion of *ShowNum* for later.

```
'*
'* Display initial decimal and hexadecimal values of each row of grid.
'*
ShowInitialValues:
  FOR row = 1 TO 8
    decimalValue(row) = 0
    GOSUB ShowNum
  NEXT row
  RETURN
```

Figure 8-11. Showing the values at the beginning of the program

Recording the pattern-pixel selected

The *CatchClick* routine, listed in Figure 8-12, is active most of the time the program is running, waiting for you to click a spot in the grid and then using the mouse

```
'*
'* Wait for user to click in grid, then determine row/column location
'* of click, update values of grid and call subroutines to show
'* decimal and hexadecimal values and display pattern created.
'*
CatchClick:

   '*
   '* Wait for button press.
   '*
   WHILE MOUSE(0) = 0
   WEND
   xCord = MOUSE(1)                          'current x coordinate of mouse
   yCord = MOUSE(2)                          'current y coordinate of mouse
   row = ((8 * yCord \ high) + 1)                    'compute row number
   doubleRow = INT((row - 1) \ 2)                    'which double-row set
   IF row MOD 2 = 0 THEN offset = 8 ELSE offset = 16
   column = ((8 * xCord \ wide) + 1)                 'compute column number
   bitLocation = offset - column                        'which bit out of 16

   '*
   '* Set up array to describe selected grid location.
   '*
   top = (row -1) * increment                                   'top border
   left = (column -1) * increment                               'left edge
   SetRelRect top, left
   INVERTRECT VARPTR(bound(0))                               'invert selection

   '*
   '* Update decimal value of row.
   '*
   decimalValue(row) = decimalValue(row) XOR power(8 - column)
   GOSUB ShowNum                                              'show number
   GOSUB PaintWin                                             'show pattern
   GOTO CatchClick                                        'wait for next click
```

Figure 8-12. Recording the pixel selected in the grid with *CatchClick*

coordinates to compute the row and column containing that spot. Once it knows the grid location you have clicked, *CatchClick* uses two subroutines—*ShowNum* and *PaintWin*—to compute and show the value of the row in the middle window, and to create and display the pattern in the right window.

The first loop in this section of code simply stalls as long as the value returned by MOUSE(0) is equal to zero (no mouse action). The program breaks out of this WHILE...WEND loop only when the mouse button is clicked. The MOUSE(1) and MOUSE(2) functions return the x and y coordinates of the pointer at the time of the last MOUSE(0), which would be when the button was clicked, and these values are assigned to the program variables *xCord* and *yCord*.

Integer division

The next line introduces one new concept: that of integer division. Integer division, denoted by a backslash (\) rather than a regular slash, rounds the dividend and divisor to integer values before dividing, and then truncates the quotient—about the same thing you would do if you had to divide 8.9 by 4.2 in your head (you would round it to 9 divided by 4 and say the truncated answer is about 2). The computer performs integer division faster than it does floating-point division (the regular slash), and in situations like this, where the operands are all integers and the dividend is an even multiple of the divisor, the answer is the same either way.

```
row = ((8 * yCord \ high) + 1)
```

The method by which this program line computes the row number is almost exactly the reverse of the process used to compute the location to draw a grid line. The expression (*8 * yCord \ high*) evaluates to an integer in the range 0 through 7; adding 1 makes it a row number. You can check the math on this quite easily. Pick a possible value of *yCord* (remember, it can vary from 0 through 160, the coordinates assigned to the top and bottom of the pattern display window), then multiply by 8 and divide by 160 (the value of *high*). For example, let's say you picked 70 for *yCord:* 70 times 8 is 560; 560 divided by 160 would be 3.5 if this were floating-point division, but since this is integer division, the answer is truncated to 3; add 1 and you have 4. If you do a

quick sketch of our grid and figure out where 70 pixels down from the top would be, you will find it is in the fourth row.

We will use the row and column numbers directly to highlight the location clicked on the grid, but to create the pattern displayed in window #2, we must compute the value of a double row, just as when we performed this operation by hand. There are four double rows, numbered 0 through 3, and the next line decides which double row was clicked:

```
doubleRow = INT((row - 1) \ 2)
```

The INT function returns the largest integer that is less than or equal to the expression within the parentheses. Again, try the math in your head with a few row numbers.

Modulo arithmetic

The next line introduces another new concept: modulo arithmetic. Modulo arithmetic, denoted by the operator MOD, provides the integer *remainder* of integer division. Modulo arithmetic is very useful, once you get in the habit of using it. I use it most frequently to count by some number other than one. MOD's syntax is:

dividend MOD *divisor* = *remainder*

Both the dividend and the divisor are rounded to integers before the division takes place, and the remainder is naturally an integer.

This program line uses MOD to check if the row clicked is the first or second row of a double-row set.

```
IF row MOD 2 = 0 THEN offset = 8 ELSE offset = 16
```

As you can see, if the row is equal to 2, 4, 6, or 8, *row MOD 2* will be equal to 0 and the IF...THEN statement will set *offset* to 8. If the row is an odd number, then the ELSE clause will take effect and *offset* will be set to 16.

The column number is computed in the same manner as the row number, using the line *column =((8 * xCord \ wide) +1).* Then the *offset* and *column* values are used to assign a value to *bitLocation,* which tells us which of the 16 bits in the double row was clicked. (We'll do something kind of tricky with this in a few minutes.)

```
bitLocation = offset - column
```

In this program I have used two different methods to number grid positions, and the calculation of *bitLocation* is a conversion point between the two that could cause some confusion. I numbered columns in the grid from 1 through 8, going from left to right. This seemed like an intuitively correct way to do it. But I numbered the bit locations in the double row from 0 to 15, going from right to left, since we will use the combined bits of the double row to represent a binary number and that is the conventional notation for binary numbers. Figure 8-13 shows the relationship between the *bitLocation* values and the columns.

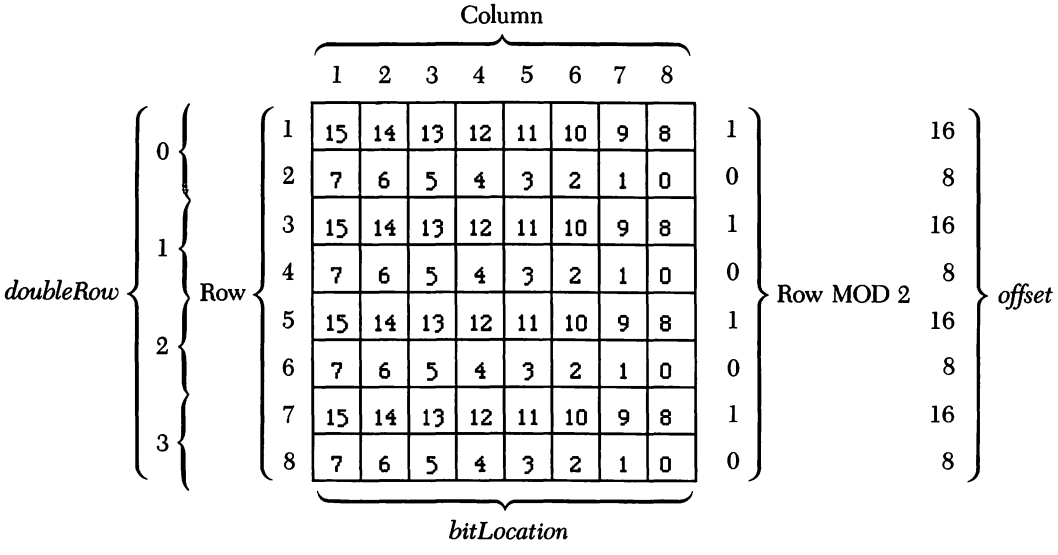


Figure 8-13. The relationship of *bitLocation* values to grid columns

Calling a subprogram

The next few lines store the boundaries of the selected grid location in an array and then use the array to invert the selection, which simulates a black pixel in the giant grid. This section introduces the concept of calling subprograms, which you will find is a handy tool for rapidly writing clear and uncluttered programs.

A subprogram is similar to a subroutine, except that the subprogram has its own set of variables. Unless you specifically instruct the subprogram to share some of the variables used in the main program, its variables remain unique to it. This feature allows you to use the same subprogram in different calling programs without worrying about whether you are overwriting a significant variable in the main program. Another way in which a subprogram differs from a subroutine is that when you call it, you can pass it information to use while it performs its function.

NOTE: Although the variables in a subprogram are unique, the line labels are not, so you might want to develop line labels that include a reference to the subprogram containing them, to make them unique.

In the *CatchClick* routine, the statement *SetRelRect top, left* calls the subprogram *SetRelRect* and passes it the values of *top* and *left* (the argument list). Just as with a call that summons a ROM routine or an assembly-language program, the CALL statement itself is optional. If CALL is used, however, parentheses are required around the argument list.

Let's take a closer look at the subprogram. In fact, let's look at two (Figure 8-14 on the next page), since the primary purpose of the first is to call the second. Between them, these examples demonstrate almost everything that can be included in a subprogram.

The first line of a subprogram identifies it as a subprogram and lists the parameters being passed, using this syntax:

SUB *subprogram-name* [*(formal-parameter-list)*] STATIC

The *subprogram-name* is what you use to call it from within the main program. The *formal-parameter-list* is one of two methods of passing values between the main program and the subprogram. When the program is run, each variable listed in the parameter list will have assigned to it the current value of the sequentially corresponding variable in the argument list in the calling statement. For instance, the *SetRelRect* subprogram assigns the values of *top* and *left* to the variables *x* and *y*.

```
'*
'* SetRelRect is passed upper left corner of rectangle,
'* computes other two sides, and stores values in array.
'*
SUB SetRelRect(x, y) STATIC
   SHARED bound(), increment
   setRectangle bound(), (x), (y), x + increment, y + increment
END SUB

'*
'* Take pair of points and set rectangle so it encloses these points.
'*
SUB setRectangle(array(), y1, x1, y2, x2) STATIC
   array(0) = y1
   array(1) = x1
   array(2) = y2
   array(3) = x2
END SUB
```

Figure 8-14. The *SetRelRect* and *SetRectangle* subprograms

The word STATIC at the end of the line is required through version 2.1 of BASIC, and means that all variables in the subprogram not specifically passed from the main program retain their values between the times the subprogram is called (as long as you are running the calling program). Since there are no alternatives to STATIC, it may seem pointless to require you to include it: The obvious implication is that future versions of BASIC will allow a different treatment of the variables between times the subprogram is called.

The SHARED statement on the second line of the subprogram demonstrates the other method of making variables in the main program accessible to the subprogram. The values of the variables listed in the SHARED statement can be altered both from within the subprogram and from the main program. Notice that to reference an array, such as the first variable in the SHARED statement, you include empty parentheses, without the number of dimensions.

The *SetRelRect* subprogram modifies the variables passed to it and passes them on to the *SetRectangle* subprogram with this statement:

```
setRectangle bound(), (x), (y), x + increment, y + increment
```

Each of the arguments passed is assigned to a new variable in the *SetRectangle* subprogram, which fills an array with these values and then ends, returning control to the first subprogram and then to the line after the calling statement in the main program. Notice that variables that are passed as parameters to one subprogram, and then passed by that subprogram to another one, must be enclosed in parentheses for the second transfer.

The END SUB statement obviously ends the subprogram. There can be only one END SUB in a subprogram: If you want to conditionally branch out of the subprogram from within its body, use the EXIT SUB statement.

So, we left the main program back in the *CatchClick* section, after assigning a value to the top border and left edge of the selected grid location. The subprogram *SetRelRect* causes an array to be filled with the information necessary to describe a rectangle around the selection. The line the subprogram returns to—*INVERTRECT VARPTR(bound(0))*—then inverts the selected rectangle. By inverting rather than painting with black or white, the program can just reverse the color of a selection each time you click; it doesn't have to keep track of what color the selection is.

The XOR operator

In this program, *decimalValue* is the array that stores the current decimal value of each row. This next line sets the new value by XORing the old value with the decimal value of 2 raised to the power of the number of the column clicked:

```
decimalValue(row) = decimalValue(row) XOR power(8 - column)
```

Make sense to you? Sentences like that have been known to make people lose interest in programming—or at least in reading about programming. But XORing numbers is a very handy technique that you will master pretty easily if you take the time to study it carefully, so let's pick the concept apart a bit.

XOR is a logical operator, in the same class as AND and OR. XOR performs its operation on two operands. The operands must be either numbers or expressions that evaluate to numbers. Before performing the operation, the computer converts the operands to 16-bit, signed, two's-complement integers between −32768 and +32767. An operand outside this range will stop the program and cause an error message to be displayed.

The "16-bit, two's-complement" business sounds a little complex, but it is really just a binary number composed of 1s and 0s. All the fancy stuff is just how the computer manages to deal with 2 raised to the 15th power (32768), which is one greater than the highest allowable integer.

So, the computer performs this operation we are so gently working up to by comparing the operands, bit by bit, and forming a new number based on the comparison. If the digits match, the result is 0; if they don't, the result is 1. The result of XORing any two binary digits is shown in this truth table:

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

Now, let's leave the discussion of XOR for a moment and have a closer look at the rows and columns in our eight-by-eight grid. As I said earlier, to help me keep the columns straight in my mind, I numbered the columns in each row from 1 through 8, going from left to right. The program, however, computes the value of the row by numbering the columns from 0 through 7, right to left, and considering each clicked column to have the value of 2 raised to the power of its column number. Figure 8-15 shows the value of each column in a row.

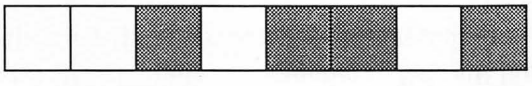
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 8-15. The value of each column in a row

This is obviously standard binary notation, so each row can be represented by a binary number, with a 1 for each cell that has been clicked and a 0 for each that hasn't. The decimal equivalent of the row is computed by adding up the powers of 2 for the clicked cells. This table shows the binary and decimal values of each power of 2 from 0 through 7:

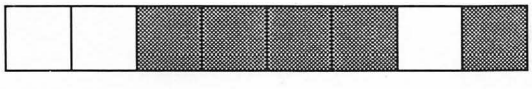
Binary	Decimal	Power of 2
00000001	1	0
00000010	2	1
00000100	4	2
00001000	8	3
00010000	16	4
00100000	32	5
01000000	64	6
10000000	128	7

So, if the pattern for a row is:



0 0 1 0 1 1 0 1 = 45 decimal

then it can be represented by the binary number 00101101, which is equal to decimal 45. If you click column 4:



0 0 1 1 1 1 0 1 = 61 decimal

the binary representation of that row of the pattern should change to 00111101, which is equal to decimal 61. The program figures this new value by using our formula that says it should XOR the old value with the value of the power of the column clicked. Let's check that out.

Old value:	00101101
$2^4 = 16$:	00010000
Old value XOR 2^4:	00111101

Use the truth table to check each bit comparison. Then work out what would happen if you clicked column 4 again. This same operation could be done with FOR...NEXT loops and a bunch of math, but the computer does it so much faster with XOR that it is worth making the effort to understand the technique.

Displaying the values

CatchClick goes on to call the *ShowNum* subroutine (Figure 8-16) which updates the display of the decimal and hexadecimal values for the row just clicked. There are two new commands introduced in this section: the PRINT USING statement and the HEX$ function.

This subroutine first makes the number-display window (window #3) active, so that subsequent PRINT statements will display there. (An alternative, since the program does not expect any input from this window, is to use the *WINDOW OUTPUT 3* statement, which leaves window #1 active for input but sends output to window #3.) It then uses a MOVETO statement to position the pen in window #3, aligned with the row just clicked in window #1, before the decimal and hexadecimal values of the row are printed.

```
'*
'* Print value of row in decimal and hexadecimal.
'*
ShowNum:
  WINDOW 3                                        'make value display window active
  MOVETO 15, (row * increment) - 5                    'get ready to print
  PRINT USING "###"; decimalValue(row);               'print value
  PRINT "= "; "&H"; HEX$ (decimalValue(row));
  WINDOW 1                                        'make window #1 active
  RETURN                                      'go back to end of CatchClick
```

Figure 8-16. Displaying the row values with *ShowNum*

The next line introduces the PRINT USING variation on the standard PRINT statement, which allows you to format the appearance of printed material. The syntax for this statement is:

PRINT USING *string-exp; expression-list*

The *string-exp* is a list of special formatting characters—in our case #s—and the *expression-list* is the list of string or numeric expressions to be printed. Each # sign in *string-exp* is a space holder representing one digit of a number to be printed. Other formatting characters are available to control how many characters are to be printed from a string, and to force other special formatting. If the number actually printed has fewer digits than the spaces available, it is right-aligned (preceded by the extra spaces). The purpose of the PRINT USING statement in this section is to keep the decimal values, which may have different numbers of digits, neatly aligned so the equal signs and hex numbers printed to their right will also be neatly aligned.

The HEX$ function in the next line returns the hexadecimal equivalent of the decimal number within the parentheses. Although the PRINT USING statement that prints the decimal number and the PRINT statement that prints the hexadecimal number are on separate lines in the program, the semicolon at the end of the first statement forces everything to be displayed on one line in window #3.

Before returning control to *CatchClick,* the subroutine uses the *WINDOW 1* statement to make that window active again.

Displaying the pattern

CatchClick now calls the *PaintWin* subroutine, listed in Figure 8-17 on the next page, to make window #2—the one in which the pattern is displayed in its normal size—the active output window. *PaintWin* in turn calls the *UpdatePattern* subroutine to change the value in the *pattern* array for the double row just clicked. *UpdatePattern* uses the same XOR method used to update the decimal value of the row. When control returns to *PaintWin,* it uses the ROM call *FILLRECT* to fill window #2 with the pattern. The two *VARPTR* statements used by FILLRECT point to the first element of the *displayWin* array (window-boundary) and the first element of the *pattern* array.

```
'*
'* Call UpdatePattern to update pattern array,
'* then fill window #2 with new pattern.
'*
PaintWin:
   WINDOW 2                                          'make window #2 active
   GOSUB UpdatePattern                               'compute pattern array

   '*
   '* Fill window with new pattern.
   '*
   FILLRECT VARPTR (displayWin(0)), VARPTR (pattern(0))
   WINDOW 1
   RETURN
```

Figure 8-17. Displaying the pattern with *PaintWin*

And that's all there is to the program. If you would like a challenging experiment, you might try to modify this program so that dragging across a block of pixels on the big grid inverts the entire block.

```
'* Generating a pattern
'*
   GOSUB DefineVariables
   GOSUB CreateWindows
   GOSUB InitializePowers
   GOSUB CreateGrid
   GOSUB ShowInitialValues
   GOTO CatchClick

'*
'* Dimension arrays and define many variables used in program.
'*
DefineVariables:
   DEFINT a - z                                      'use integers for speed
```

Figure 8-18. The complete pattern-generating program

more...

```
'*
'* Dimension arrays used by program.
'*
DIM bound(4)                                    'edges of grid location
DIM pattern(3)                                  'pattern array
DIM decimalValue(8)                             'decimal value of row
DIM displayWin(3)                               'edges of display window
DIM power(15)                                   'powers of 2
wide = 160                                      'width of pattern display window
high = 160                                      'height of pattern display window
increment = 20                                  'height of row, width of column
displayWin(0) = 0                               'top of pattern display window
displayWin(1) = 0                               'left
displayWin(2) = 160                             'bottom
displayWin(3) = 160                             'right
RETURN

'*
'* Create four windows used by program. Window #4 is background window,
'* window #3 displays decimal and hexadecimal values of each row,
'* window #2 displays pattern, and window #1 is used to create pattern.
'* Note that window #1 is modal (specified by negative type number),
'* meaning that it is only window in which user can make selection.
'*
CreateWindows:
   WINDOW 4, "Pattern Generator", (0, 38) - (512, 342), 1      'background
   MOVETO 85, 250
   PRINT "Hold down command key and press . to return to BASIC."
   WINDOW 3, , (200, 60) - (300, 220), 3                       'display values
   WINDOW 2, , (320, 60) - (480, 220), 3                       'display patterns
   WINDOW 1, , (20, 60) - (180, 220), -3                       'display pixels
   RETURN
'*
'* Initialize array containing powers of 2.
'* Used to convert from pixels to binary numbers:
'*    power(bit) = 2 ^ bit
'*
InitializePowers:
   FOR bit = 0 TO 14
      power(bit) = 2 ^ bit
   NEXT bit
   power(15) = -32768!
   RETURN
```

Figure 8-18. The complete pattern-generating program *(continued)*

more...

```
'*
'* Draw grid used to simulate 8 by 8 array of pixels.
'*
CreateGrid:
   FOR hLine = 1 TO 7                                    'draw horizontal lines
      MOVETO 0, hLine * increment
      LINETO wide, hLine * increment
   NEXT hLine

   FOR vLine = 1 TO 7                                    'draw vertical lines
      MOVETO vLine * increment, 0
      LINETO vLine * increment, high
   NEXT vLine
   RETURN

'*
'* Display initial decimal and hexadecimal values of each row of grid.
'*
ShowInitialValues:
   FOR row = 1 TO 8
      decimalValue(row) = 0
      GOSUB ShowNum
   NEXT row
   RETURN

'*
'* Wait for user to click in grid, then determine row/column location
'* of click, update values of grid and call subroutines to show
'* decimal and hexadecimal values and display pattern created.
'*
CatchClick:

   '*
   '* Wait for button press.
   '*
   WHILE MOUSE(0) = 0
   WEND
   xCord = MOUSE(1)                                  'current x coordinate of mouse
   yCord = MOUSE(2)                                  'current y coordinate of mouse
   row = ((8 * yCord \ high) + 1)                        'compute row number
   doubleRow = INT((row - 1) \ 2)                        'which double-row set
```

Figure 8-18. The complete pattern-generating program *(continued)*

more...

```
    IF row MOD 2 = 0 THEN offset = 8 ELSE offset = 16
    column = ((8 * xCord \ wide) + 1)                    'compute column number
    bitLocation = offset - column                        'which bit out of 16

    '*
    '* Set up array to describe selected grid location.
    '*
    top = (row -1) * increment                           'top border
    left = (column -1) * increment                       'left edge
    SetRelRect top, left
    INVERTRECT VARPTR(bound(0))                          'invert selection

    '*
    '* Update decimal value of row.
    '*
    decimalValue(row) = decimalValue(row) XOR power(8 - column)
    GOSUB ShowNum                                        'show number
    GOSUB PaintWin                                       'show pattern
    GOTO CatchClick                                      'wait for next click

'*
'* Print value of row in decimal and hexadecimal.
'*
ShowNum:
    WINDOW 3                                 'make value display window active
    MOVETO 15, (row * increment) - 5                     'get ready to print
    PRINT USING "###"; decimalValue(row);                'print value
    PRINT "= "; "&H"; HEX$ (decimalValue(row));
    WINDOW 1                                             'make window #1 active
    RETURN                                   'go back to end of CatchClick

'*
'* Call UpdatePattern to update pattern array,
'* then fill window #2 with new pattern.
'*
PaintWin:
    WINDOW 2                                             'make window #2 active
    GOSUB UpdatePattern                                  'compute pattern array
```

Figure 8-18. The complete pattern-generating program *(continued)*

more...

```
   '*
   '* Fill window with new pattern.
   '*
   FILLRECT VARPTR (displayWin(0)), VARPTR (pattern(0))
   WINDOW 1
   RETURN

'*
'* Pattern array is composed of four values, one for each double row in
'* grid. Next subroutine updates value for double row just clicked by
'* XORing current value with power of 2 of bit location clicked.
'*
UpdatePattern:
   pattern(doubleRow) = pattern(doubleRow) XOR power(bitLocation)
   RETURN

'*
'* SetRelRect is passed upper left corner of rectangle,
'* computes other two sides, and stores values in array.
'*
SUB SetRelRect(x, y) STATIC
   SHARED bound(), increment
   setRectangle bound(), (x), (y), x + increment, y + increment
END SUB

'*
'* Take pair of points and set rectangle so it encloses these points.
'*
SUB setRectangle(array(), y1, x1, y2, x2) STATIC
   array(0) = y1
   array(1) = x1
   array(2) = y2
   array(3) = x2
END SUB
```

Figure 8-18. The complete pattern-generating program *(continued)*

The MiniPaint Program

Chapter 9

Now that you understand how to create different patterns on the Macintosh screen, let's have a look at a program that uses these patterns. If you have a Macintosh, you are at least vaguely familiar with the MacPaint program created by Bill Atkinson. The diminutive version of MacPaint we're going to create in this chapter—MiniPaint—won't threaten Bill's position as the supreme master of Macintosh graphics, but it will demonstrate how easily you can include many of the various shapes and patterns in your own programs.

The MiniPaint program allows you to create empty and filled rectangular or oval frames and produce freehand drawings in the large center window of the work area shown in Figure 9-1 on the following page. By clicking the different options arranged around the edge, you can specify the shape, thickness, and pattern of the frame or drawing line and the fill pattern used.

There are few new commands used in this program, so we won't get too bogged down in explanation. As you read through the program, notice that almost all screen positions are expressed relative to the width or height of the active window, and that these dimensions are returned by the WINDOW(2) and WINDOW(3) functions. This approach allows you to experiment with different window sizes without rewriting the entire program.

The program consists of seven sections. Section one, shown in Figure 9-2 on the following page, routes the program through sections two through six to set up the initial screen. (I will discuss the ON DIALOG and DIALOG ON statements shortly, when we've taken a look at these first six sections.) Then the program flows into section seven, the main loop, where it will stay.

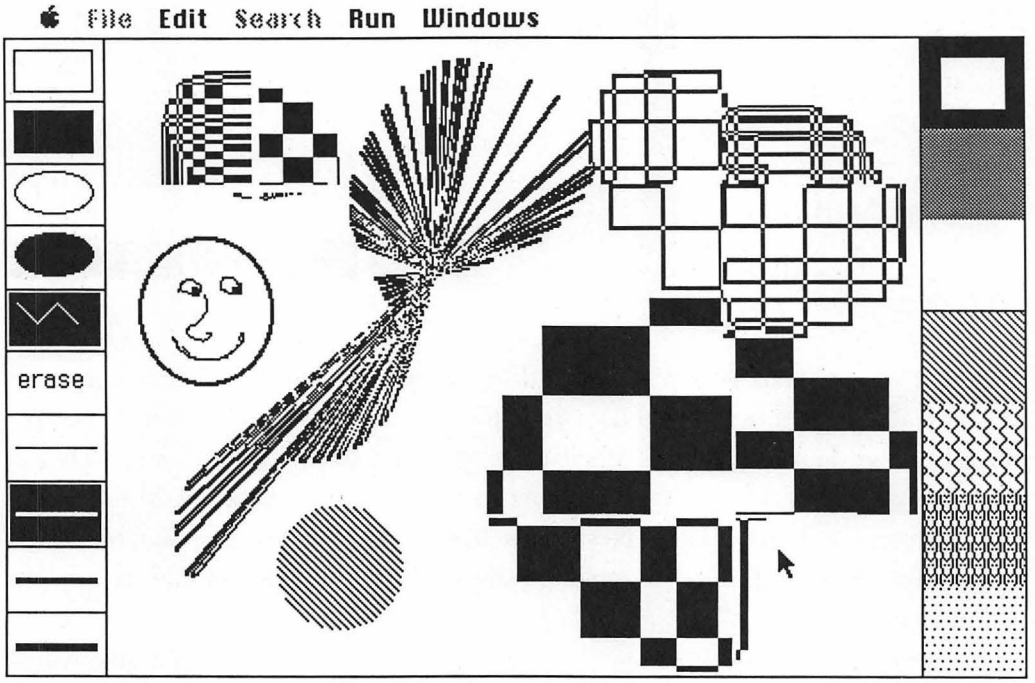


Figure 9-1. The MiniPaint work area

You will notice, if you study the full program listing in Figure 9-20, at the end of the chapter, that the sections of the program are grouped there not in the order I will discuss them in but by type: the subprograms in one group, the subroutines in another, and so on. This is only for convenience in finding different sections, and you can arrange your listing as you see fit.

```
GOSUB DefineVariables
GOSUB CreateWindows
GOSUB CreateSymbols
GOSUB CreatePatterns
GOSUB ShowDefaults
ON DIALOG GOSUB SelectWindow                    'if inactive window clicked
DIALOG ON
```

Figure 9-2. Calling subroutines to set up the MiniPaint work area

```
'*
'* Define variables and dimension arrays.
'*
DefineVariables:
   DEFINT a - z                          'make all variables integers for speed
   DIM div(3)                            'number of window division
   DIM but(3)                            'current button in each window
   DIM pat(28)                           'pattern definitions
   DIM bord(3)                           'top, left, bottom, and right borders
   DIM oldBord(3)
   false = 0
   true = -1
   div(1) = 6                            'number of divisions in shape window
   div(2) = 4                            'number of divisions in line-thickness window
   div(3) = 7                            'number of divisions in pattern window
   function = 1                          'default shape to draw--rectangle
   RETURN
```

Figure 9-3. Defining the variables

Defining the variables

Figure 9-3 lists the *DefineVariables* subroutine—the usual define-declare-and-dimension section. The values assigned to the variables *false* and *true* are the same as those generated by the logical operators: 0 for *false* and −1 for *true*. The values of the array elements *div(1)*, *div(2)*, and *div(3)* are the number of divisions in windows #1, #2, and #3, respectively.

Creating the windows

The *CreateWindows* subroutine, shown in Figure 9-4 on the following page, uses familiar WINDOW statements to produce the four windows you see on the MiniPaint screen: a main work area, two small windows on the left for shapes and line thicknesses, and one long window on the right for patterns.

```
'*
'* Create four windows.
'*
CreateWindows:
  WINDOW 1, , (0, 20) - (50, 209), 3          'shape window
  WINDOW 2, , (0, 210) - (50, 342) ,3         'line-thickness window
  WINDOW 3, , (460, 20) - (512, 342), 3       'pattern window
  WINDOW 4, , (51, 20) - (459, 342), 3        'work-area window
  RETURN
```

Figure 9-4. Creating the windows

Creating the symbols

Figure 9-5 shows the *CreateSymbols* subroutine, which draws the horizontal division lines in the shape, line, and pattern windows and then creates the symbol or pattern for each option and displays it. The *FOR windo = 1 TO 3* loop draws the horizontal division lines in each of the three windows. First the window is made active (*WINDOW windo*), and then the WINDOW(2) and WINDOW(3) functions are used to retrieve the width and height of the window and assign these values to the variables *wide* and *high*. These values are used, along with the number of divisions in the window, to compute the size of the boxes and hence their positions down the screen, and then to draw the lines. Nothing here should look particularly confusing to you if you have worked your way through the previous programs.

Creating the shapes

The method of creating the actual symbols or patterns varies somewhat for each window. To create the symbols in window #1, the height, width, and number of divisions in the window—*div()*—are passed to the *DrawFuncs* subprogram, shown in Figure 9-6 on page 116.

There are only a couple of things worth pointing out in this subprogram: the DIM statement at the beginning, the variety of ROM calls to frame and invert rectangles and ovals, and the LINE STEP statement that draws the crooked line.

The only reason the DIM statement is significant here is that it again points out the concept of variables that are unique to the subprogram. If you are using a variable or an array in a subprogram but not in the main program, there is no point in creating

```
'*
'* Create window divisions.
'*
CreateSymbols:
   FOR windo = 1 TO 3
      WINDOW windo                                    'make window windo active
      wide = WINDOW(2)                                'width of current window
      high = WINDOW(3)                                'height of current window
      FOR division = 1 TO div(windo) - 1              'draw division lines
         MOVETO 0, division * high \ div(windo)
         LINETO wide, division * high \ div(windo)
      NEXT division
   NEXT windo

   '*
   '* Draw symbols in shape window.
   '*
   WINDOW 1                                           'shape window
   high = WINDOW(3)
   wide = WINDOW(2)

   '*
   '* Call subprogram that creates six shapes.
   '*
   DrawFuncs wide, high, div(1)

   '*
   '* Create line-thickness window.
   '*
   WINDOW 2
   high = WINDOW(3)
   wide = WINDOW(2)

   FOR division = 1 TO 4
      PENSIZE 1, division
      MOVETO 5, (division - .5) * high \ div(2)       'note parentheses
      CALL LINE (wide - 10, 0)
   NEXT
   RETURN
```

Figure 9-5. Creating the symbols

```
'*
'* Draw symbols in shape window.
'*
SUB DrawFuncs (wide, high, numDivs) STATIC
   DIM Rect(3)
   boxWidth = wide - 10 : boxHeight = high \ numDivs - 9
   deltaY = high \ numDivs

   '*
   '* Draw rectangle.
   '*
   SetRelRect Rect(), 5, 5, boxWidth, boxHeight
   FRAMERECT VARPTR(Rect(0))

   '*
   '* Draw filled rectangle.
   '*
   SetRelRect Rect(), 5, deltaY + 5, boxWidth, boxHeight
   INVERTRECT VARPTR(Rect(0))

   '*
   '* Draw circle.
   '*
   SetRelRect Rect(), 5, 2 * deltaY + 5, boxWidth, boxHeight
   FRAMEOVAL VARPTR(Rect(0))

   '*
   '* Draw filled circle.
   '*
   SetRelRect Rect(), 5, 3 * deltaY + 5, boxWidth, boxHeight
   INVERTOVAL VARPTR(Rect(0))

   '*
   '* Draw crooked line.
   '*
   LINE (6, 4.3 * deltaY) - STEP (10, 10)
   LINE - STEP (10, -10)
   LINE - STEP (10, 10)
```

Figure 9-6. Creating the shape symbols with *DrawFuncs*

more...

```
    '*
    '* Draw eraser.
    '*
    MOVETO 6, 6 * deltaY - 10
    PRINT "erase";
END SUB
```

Figure 9-6. Creating the shape symbols with *DrawFuncs (continued)*

or defining it in the main program and then passing it to the subprogram: Even if the *rect* array existed in the main program, dimensioning a new array with the same name in the subprogram would have no effect on it. However, if you call the subprogram more than once, you will have to use the ERASE statement at the end of it to eliminate the array, so that dimensioning it the next time will not cause an error.

The ROM calls FRAMERECT, INVERTRECT, FRAMEOVAL, and INVERTOVAL all use the familiar format of pointing to the first element of an array containing the top, left, bottom, and right boundaries of the section of the screen where the shape will appear.

The LINE statement is another BASIC statement that has a variety of formats and uses. (There is also a LINE ROM call, which is always preceded by the CALL statement, to avoid confusion.) The syntax for the LINE statement is:

LINE [[STEP] (*x1,y1*)] – [STEP] (*x2,y2*)[,[*color*][, b[f]]]

It is used to draw a line or a box. Without the *b* and *f* options tacked onto the end, LINE draws a line from point (x1,y1) to point (x2,y2). If you add the *b*, it draws a box with opposite corners at those points; if you add the *f* option after the *b*, it also fills the box with the current pattern.

Although the word *color* does not seem applicable to the Macintosh screen—at least right now—the choice of black or white is one more choice than the purchasers of Henry Ford's first machine had. As then, the default is black. The number 30 in the *color* position causes both the line and the fill pattern to be white. The number 33 is for black, but since leaving the option blank will also produce a black line, there seems to be little point in specifying it. Note, however, that if you omit the *color* option but still want to use the *b* or *bf* options, you must include the commas that show where the *color* option would appear.

The STEP option changes the (x,y) coordinates from absolute pixel locations to relative pixel locations. In other words, a STEP location is x pixels horizontally and y pixels vertically relative to the previous position of the pen, not the corner of the window. STEP can be used with the first set of coordinates, the second set, or both. You can also omit the first set, as we do in *DrawFuncs* in the three statements used to draw the crooked line: The statement *LINE - STEP (x,y)* draws a line from the current pen position to the point (x,y) pixels away. The first LINE statement in this section draws the initial segment of the crooked line; the second and third statements step the line to the right and either up or down 10 pixels.

Creating the lines

Window #2 displays the line-width options that are available for drawing lines and frames. The different line thicknesses are set with the PENSIZE ROM call in the *CreateSymbols* subroutine:

```
PENSIZE 1, division
```

Pen width and height are in pixels. In this section we set the width equal to 1 and the height equal to the number of the window division in which the line is drawn. Since window #2 is divided into four parts, numbered 1 through 4, the pen heights will range from 1 to 4. We can leave the pen width at 1 here, since we are drawing only horizontal lines, but in the main body of the program, where we draw both horizontal and vertical lines in the work area, the width and height will be set to the same value.

Each line is drawn with the same sequence of statements. First the pen size is set and the pen is positioned five pixels in from the left edge and halfway down a division. Then the LINE ROM call is used to draw the line to another location relative to the first. So the LINE ROM call works just like the STEP option in the LINE statement, in this case drawing a line from the current pen location to a point 10 pixels less than the width of the window and on the same level.

There are two things of importance to note about the PENSIZE call: It applies only to the current output window, and it applies only to lines produced by other ROM graphic calls (not to BASIC LINE and CIRCLE statements). If you change the output

window, the pen size reverts to the last size specified for that window. If no size has specifically been set, the default size of 1,1 is used.

Creating the patterns

The *CreatePatterns* subroutine (Figure 9-7), which displays the seven available patterns, reads the data statements listed elsewhere in the program and stores the hex numbers found there in the *pat* array. Each hex number represents a double row of the pattern. The elements of the array are then taken four at a time, each set of four being used to define one of the seven patterns, which are displayed in window #3.

There are several ways to make information available to a program. You can store the information in a disk file and retrieve it with an INPUT$, INPUT#, or LINE INPUT# statement. Or you can ask the user to supply the information via the keyboard and retrieve it with an INPUT or LINE INPUT statement or the INKEY$ function. Or you can store the information in the program itself in the form of DATA statements that can be read as needed and assigned to variables. This last method (used in this subroutine) is particularly appropriate when the information is not subject to change.

```
'*
'* Read data for patterns.
'*
CreatePatterns:
   WINDOW 3                                          'make window #3 active
   wide = WINDOW(2)                                  'get its height and width
   high = WINDOW(3)
   FOR design = 0 TO 27
      READ pat(design)                               'read DATA statement
   NEXT
   countBy4 = 0                                      'initialize--used to count by fours
   FOR division = 0 TO div(3) - 1                    'fill pattern swatches
      SetRelRect bord(), 0, division * high \ div(3), wide, high \ div(3)
      FILLRECT VARPTR(bord(0)), VARPTR(pat(countBy4))
      countBy4 = countBy4 + 4                        'increment counter
   NEXT
   RETURN
```

Figure 9-7. Creating the patterns

CreatePatterns uses the READ statement to assign the hexadecimal numbers stored in the DATA statements to the 28 elements of the *pat* array. The syntax of the READ statement is:

READ *variable-list*

The *variable-list* can contain as many variables as you like, and they can be either numeric or string. The only restrictions are that there must be at least as many pieces of data to read as there are variables in the statement, and that each item must be of the same type (string or numeric) as the variable to which it is assigned.

The first FOR...NEXT loop in *CreatePatterns* is cycled through 28 times; at each pass it assigns a value from the DATA statements to an element in the *pat* array. After all 28 values have been read, the next FOR...NEXT loop passes every fourth element of the array to the FILLRECT ROM call, which uses it to create and display a pattern.

It is important to note that pattern arrays can consist of more than four elements. Most programs show a separate array for each pattern, and use the VARPTR function to point to the memory location of the first element of the array when creating the pattern. However, as you can see from this example, one pattern array can contain as many elements as you like: You simply point to the first of the four consecutive elements you want to use. This approach would be very useful if you wanted to create an evenly graduated gray scale for highlighting or shading graphics.

Highlighting the defaults

The *ShowDefaults* subroutine in Figure 9-8 highlights the default setting for each of the option windows. Later, when you select a different option, the highlight will have to be removed and applied to your new choice.

The selection is highlighted by inverting a smaller rectangle centered inside the rectangle holding the selected symbol. The inverted rectangle is made a little smaller than the selection it is highlighting in order to create a border around the highlight. The width of the border is determined by the value of the variable *inset.* The reason for making the inset a variable is that a wider border (10 pixels) is needed in window #3 to make it obvious which pattern is selected (an inverted white pattern looks just like a black pattern). After the inset is defined, each window is made active in turn, and the *SetRectangle* subprogram is called to create a rectangle with dimensions two

```
'*
'* Highlight default selection in each window.
'*
ShowDefaults:
  inset = 2                                          'size of border around highlight
  FOR windo = 1 TO 3
    IF windo = 3 THEN inset = 10
    WINDOW windo
    wide = WINDOW(2)
    high = WINDOW(3)
    SetRectangle bord(), inset, inset, wide - inset, high \ div(windo) - inset
    INVERTRECT VARPTR(bord(0))                       'invert center of selection
    but(windo) = 1                                   'store this window's button-press
  NEXT
  RETURN
```

Figure 9-8. Highlighting the defaults

times *inset* smaller than the width and height of a window division. The INVERTRECT ROM call is then invoked to highlight that area in the first division of the window (the default division).

The last action in this section assigns the number of the currently highlighted division to the variable *but(windo)*, which stands for the active "button" in that window. This is done so that when another option is selected, we will know which option has to have the highlight removed.

Changing options

Before moving into the main loop of the program to create shapes in window #4, we must make provisions for what to do if another window is clicked, which would indicate a desire to change options. We can do this with the ON DIALOG statement which I deferred discussing when we encountered it at the beginning of the program. This is ON DIALOG's general format:

ON DIALOG GOSUB *line*

The DIALOG function returns information about events that involve buttons, windows, and edit fields created by BASIC. The ON DIALOG GOSUB statement is an event trap that sends the program to a specified line if there has been a change in one

of the conditions monitored by the DIALOG function. Once you have specified where to go if a dialog event is trapped, you activate the trapping with the DIALOG ON statement. We will take a closer look at the DIALOG function soon.

The main loop

The *MainLoop* routine, shown in Figure 9-9, is a short section of code through which the program loops continuously while waiting for the mouse button to be pressed to indicate that a shape should be drawn. While waiting, the program is constantly updating the value of *yCord* with the current coordinate value of MOUSE(2). We will use this information a little later, when a different option is selected by clicking in a side window.

The sections that follow *MainLoop* contain the routines that actually draw rectangles, ovals, and lines as you drag the mouse around in window #4. Other than the difference in the actual ROM call that draws the shape, the routines for the rectangle and the oval are identical. Both have chunks of code that are used repeatedly, so these chunks have been assigned to subprograms, several of which you have already seen in other programs.

```
'*
'* Allow user to create lines and shapes, while waiting
'* for click in side windows.
'*
MainLoop:
  WINDOW 4                                          'make window #4 active
  WHILE MOUSE(0) = 0

    '*
    '* While waiting for mouse click, store current location of
    '* pointer. This information will be used if next click is outside
    '* work-area window.
    '*
    yCord = MOUSE(2)
  WEND
  ON function GOSUB Rect, Oval, Lin
  GOTO MainLoop
```

Figure 9-9. Waiting for the mouse button to be pressed

When the mouse button is pressed, the program branches to the most recently specified drawing routine (*Rect, Oval,* or *Lin*), as determined by the ON...GOSUB statement, which has this syntax:

ON *expression* GOSUB *line-list*

This is a "computed GOSUB" statement: The value of the expression is computed and the program branches to the subroutine whose label or line number is that far into *line-list.* For example, if the value of the expression is 2, the program goes to the second subroutine in the list; if the value is 7, the program goes to the seventh subroutine listed. If the value is 0, or if it is greater than the number of items in *line-list,* the program continues with the statement after ON...GOSUB. (A parallel statement that operates this way is ON...GOTO, which branches to a line other than the beginning of a subroutine.)

In our program, the expression to be evaluated is *function.* The program will branch to *Rect, Oval,* or *Lin,* depending upon the value assigned to *function.* The first time through *MainLoop,* the initially assigned value of 1 sends the program to the *Rect* subroutine. Each of these subroutines ends by returning to *MainLoop.*

Drawing rectangles

The subroutine labeled *Rect,* shown in Figure 9-10 on the following page, can create both framed and filled rectangles. The factor that determines which type is drawn is a variable named *fill,* which is set equal to *true* (−1) when either of the filled-shape options is selected from window #1.

The *Rect* subroutine has several distinctly separate stages. First it sets the pattern to black (using the *MakePattern* subprogram with an argument of 1) and draws the first rectangle (using *SetRectangle* and FRAMERECT). Then it checks to see whether the mouse button has been pressed once and is still being held down (*MOUSE(0)* = −*1*). If so, it constantly erases and redraws the rectangle as the mouse is dragged, keeping track of the coordinate information it needs using the *CopyRect* subprogram, and all the while waiting for the button to be released. Figure 9-11 on page 125 lists the three subprograms used by the *Rect* routine.

```
'*
'* Draw rectangle.
'*
Rect:
  PENMODE 10                                                    'XOR mode

  '*
  '* Draw first rectangle.
  '*
  MakePattern 1                                          'use black pattern for frame
  SetRectangle oldBord(), MOUSE(3), MOUSE(4), MOUSE(1), MOUSE(2)
  FRAMERECT VARPTR(oldBord(0))                                    'draw rectangle
  WHILE MOUSE(0) = -1                                 'while mouse button is pressed
    SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
    FRAMERECT VARPTR(oldBord(0))                    'erase rectangle (while dragging)
    FRAMERECT VARPTR(bord(0))                                   'draw new rectangle
    CopyRect oldBord(), bord()
  WEND

  '*
  '* Create filled rectangle if this symbol was selected.
  '*
  MakePattern but(3)                                       'reinstate stored pattern
  PENMODE 8                                                           'copy mode
  FRAMERECT VARPTR(bord(0))                                       'draw rectangle
  IF fill THEN PAINTRECT VARPTR (oldBord(0))

  '*
  '* And now for a little fun.
  '*
  WHILE MOUSE(0) = -2
    PENMODE 10
    SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
    FRAMERECT VARPTR(bord(0))
    IF fill THEN PAINTRECT VARPTR(bord(0))
  WEND
  PENMODE 8
  RETURN
```

Figure 9-10. Drawing rectangles with *Rect*

```
'*
'* Make one of stored patterns active.
'*
SUB MakePattern(patNum) STATIC
   SHARED pat()
   PENPAT VARPTR(pat(4 * patNum - 4))                    'Pat() is first of group of four
END SUB

'*
'* Copy one rectangle into another.
'*
SUB CopyRect(rect1(), rect2()) STATIC
   FOR count = 0 TO 3
      rect1(count) = rect2(count)
   NEXT
END SUB

'*
'* Take pair of points and set rectangle so it encloses these points.
'*
SUB SetRectangle(array(), x1, y1, x2, y2) STATIC
   array(0) = y1
   array(1) = x1
   array(2) = y2
   array(3) = x2
   IF x1 > x2 THEN SWAP array(1), array(3)
   IF y1 > y2 THEN SWAP array(0), array(2)
END SUB

'*
'* SetRelRect is just like SetRectangle except it takes  as input
'* top, left point and height and width.
'*
SUB SetRelRect(array(), x, y, wide, high) STATIC
   CALL SetRectangle (array(), (x), (y), x + wide, y + high)
END SUB
```

Figure 9-11. The drawing subprograms *MakePattern, CopyRect, SetRectangle,* and *SetRelRect*

The *MakePattern* subprogram points to the first of a set of four elements in the pattern array—in this case *pat(0)*—so that the rectangle that is drawn as you drag the mouse will always be in the same pattern. (I originally used whatever pattern was

currently selected to draw this rectangle, but found that if the white pattern was selected, the rectangle was a little difficult to detect against the white background.) When the mouse button is released, *MakePattern* is called again to reset the pattern to the one currently selected, before drawing the final rectangle.

The *SetRectangle* subprogram fills the array passed to it with the top, left, bottom, and right boundaries of the rectangle described by the mouse drag. The starting and current coordinates of the pointer are used to define the boundaries, and the direction in which the mouse is being dragged is checked by comparing the starting and current coordinates. If the drag is not down and to the right, the upper/lower or left/right boundaries are exchanged using the SWAP statement. (This swap is done in order to keep the coordinates in the order expected by the ROM call that draws the rectangle.)

The boundaries are stored in an array called *oldBord,* and a rectangle is drawn. The new current coordinates of the pointer are located and stored in an array called *bord,* the old rectangle is erased by drawing another rectangle on top of it, and a new rectangle is drawn using the original starting coordinates and the new coordinates of *bord* for the ending point. Then the *CopyRect* subprogram is called to move the elements of *bord* into the *oldBoard* array, freeing the *bord* array to receive the coordinates of the still-moving pointer.

The rapid drawing and erasing of rectangles has the effect of lightly tracing the changing shape of the rectangle as the mouse is dragged. After the button is released, one more rectangle is drawn, this time from the starting coordinate to the ending coordinate. It is at this point that *fill* is checked, and, if it is *true,* PAINTRECT is called.

If the mouse button is double clicked (*MOUSE(0)* = −2), the next loop is entered. Look closely at this loop to see if you can figure out what will happen as the mouse is dragged. The significant differences between this routine and the previous one are that the rectangle is not erased after each draw, and the pen mode is set to 10, which is XOR mode, rather than 8, which is Copy mode.

Drawing ovals

The subroutine labeled *Oval,* shown in Figure 9-12, does essentially the same thing as *Rect,* using the FRAMEOVAL and PAINTOVAL ROM calls.

```
'*
'* Draw oval.
'*
Oval:
  PENMODE 10
  MakePattern 1                                    'use black pattern for rectangle
  SetRectangle oldBord(), MOUSE(3), MOUSE(4), MOUSE(1), MOUSE(2)
  FRAMEOVAL VARPTR(oldBord(0))
  WHILE MOUSE(0) = -1
    SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
    FRAMEOVAL VARPTR(oldBord(0))
    FRAMEOVAL VARPTR(bord(0))
    CopyRect oldBord(), bord()
  WEND
  PENMODE 8
  MakePattern but(3)                               'reinstate selected pattern
  FRAMEOVAL VARPTR(bord(0))
  IF fill THEN PAINTOVAL VARPTR(oldBord(0))
  WHILE MOUSE(0) = -2
    PENMODE 10
    SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
    FRAMEOVAL VARPTR(bord(0))
    IF fill THEN PAINTOVAL VARPTR(bord(0))
  WEND
  PENMODE 8
  RETURN
```

Figure 9-12. Drawing ovals

Drawing lines

The third subroutine, shown in Figure 9-13 on the following page, is labeled *Lin* (notice that you have to abbreviate Line so that BASIC doesn't confuse the *Lin* subroutine with the LINE statement). *Lin* draws a line that follows the pointer around as long as the mouse button is held down after a single click. Dragging the mouse after a double click will continuously draw new lines from the starting point to the changing pointer location, giving a sunbeam effect.

```
'*
'* Draw line.
'*
Lin:

   '*
   '* Move pen point to beginning of drag.
   '*
   IF MOUSE(0) = -1 THEN MOVETO MOUSE(3), MOUSE(4)
   WHILE MOUSE(0) = -1                          'while mouse is being dragged
      point1 = MOUSE(5)                        'ending coordinate is also current
      point2 = MOUSE(6)                        'coordinate as long as button is down
      LINETO point1, point2                                        'draw line
      MOVETO point1, point2                                'move pen to end of line
   WEND                                                          'button released

   '*
   '* Mouse double clicked before drag.
   '*
   WHILE MOUSE(0) = -2
      PENMODE 10
      MOVETO MOUSE(3), MOUSE(4)
      LINETO MOUSE(5), MOUSE(6)
   WEND
   PENMODE 8
   RETURN
```

Figure 9-13. Drawing lines with *Lin*

Dialog event trapping

If the button is clicked while the pointer is over a window other than window #4, BASIC considers this a significant dialog event and traps it. The ON DIALOG GOSUB statement executed earlier then sends the program to the line labeled *SelectWindow* to execute the subroutine shown in Figure 9-14.

The DIALOG function is like the MOUSE function, in that DIALOG(0) returns the type of event that has occurred, and DIALOG(1) through DIALOG(5) give specific information about the event. In this case, the only dialog event we want to trap is when

```
'*
'* Routine branched to if dialog event is trapped
'*
SelectWindow:
   IF DIALOG(0) <> 3 THEN RETURN                    'inactive window clicked
   WINDOW DIALOG(3)                                 'make clicked window active
   windo = WINDOW(0)                                'active window
   wide = WINDOW(2)                                 'width
   high = WINDOW(3)                                 'height
   GOSUB SelectItem                                 'get option selection
   WINDOW 4                                         'make window #4 active
   ON windo GOTO SetShape, SetLine, SetPattern      'implement selection
```

Figure 9-14. Responding to dialog events with *SelectWindow*

an inactive window is clicked, which causes DIALOG(0) to return the value 3. So if DIALOG(0) is not equal to 3, the first line of this subroutine simply returns the program to where it was when the dialog event occurred. If DIALOG(0) is equal to 3, then DIALOG(3), which returns the ID number of the inactive window that was clicked, is used to make the clicked window active. The ID number, width, and height of the current window are returned by the WINDOW(0), WINDOW(2), and WINDOW(3) functions, and then the *SelectItem* subroutine determines which option in the new window was clicked.

Selecting options

The *SelectItem* subroutine shown in Figure 9-15 on the following page determines which option was clicked, removes the highlight from the old selection, highlights the new, and then assigns the number of the new selection to the variable *butSel*. This is where the constant updating of *yCord* while the program is waiting for the mouse click comes in handy. If we wait until the option window is clicked to check for the pointer location, we have to click twice: once to activate the window and once to spot the pointer. This is necessary because the MOUSE functions return information only about the location of the pointer relative to the active output window, so each time you change windows, the pointer coordinates are automatically reset, thus requiring the second click.

```
'*
'* Determine which option was clicked.
'*
SelectItem:
  inset = 2                                                    'border for highlight
  IF windo = 3 THEN inset = 10                                 'window #3
  IF windo = 2 THEN yCord = yCord - 190                        'window #2
  butSel = ((div(windo) * yCord \ high) + 1)                   'increment down from top
  IF windo = 1 AND butSel = 6 THEN RETURN                      'don't highlight erase button
  top = (but(windo) - 1) * high \ div(windo) + inset
  SetRelRect bord(), inset, top, wide - 2 * inset, high \ div(windo) - 2 * inset
  INVERTRECT VARPTR(bord(0))                                   'return previous selection to normal
  top = (butSel - 1) * high \ div(windo) + inset
  SetRelRect bord(), inset, top, wide - 2 * inset, high \ div(windo) - 2 * inset
  INVERTRECT VARPTR(bord(0))                                   'invert center of new selection
  but(windo) = butSel
  RETURN
```

Figure 9-15. Deselecting the old option and selecting the new with *SelectItem*

If the click was in window #1 or #3, the previously stored value of *yCord* is used directly to determine the window division selected. If the click was in window #2, which starts 190 pixels down from the top of window #4 (where we were when *yCord* was first stored), then 190 is subtracted from *yCord* to give it a value that is relative to the top of window #2.

Once the window division is determined, the rest of the subroutine simply sets up and calls a few familiar subprograms and ROM calls to remove the highlight from the old selection and highlight the new one, and then updates the value of *butSel.*

When the program returns from *SelectItem,* it makes window #4 active and then goes to the subroutine determined by the window in which the selection was made.

If the option window selected was window #1, the program branches to the *SetShape* subroutine, shown in Figure 9-16. This subroutine sets the values of *fill* and *function,* depending upon the item selected in the window. If either the filled rectangle or the filled oval is selected, the variable *fill* is set equal to *true,* which causes the relevant drawing routine to call PAINTRECT or PAINTOVAL rather than calling FRAMERECT or FRAMEOVAL. If the hollow rectangle or oval is selected, *fill* is set to *false* (in case it had previously been *true*).

```
'*
'* Select option from shape window.
'*
SetShape:
   IF butSel = 6 THEN CLS : RETURN
   IF butSel = 5 THEN function = 3 : RETURN
   fill = true
   IF butSel = 4 THEN function = 2 : RETURN
   IF butSel = 2 THEN function = 1 : RETURN
   fill = false
   IF butSel = 3 THEN function = 2 : RETURN
   function = 1 : RETURN
```

Figure 9-16. Selecting the shape with *SetShape*

If the option window selected was #2, the *SetLine* subroutine (Figure 9-17) uses the PENSIZE ROM call to set the size of the pen that the other ROM calls use to draw lines and frames. You will recall that this pen size does not affect BASIC drawing statements such as LINE and CIRCLE, and applies only to the current output window. This is why it was important to make window #4 active again before branching to this section. The width and height of the pen are given in pixels, and in this case are simply set equal to the number of the option selected in window #2.

And finally, if option window #3 was selected, the *SetPattern* subroutine, shown in Figure 9-18 on the next page, passes the number of the division selected within the window (*butSel*) to the *MakePattern* subprogram, which then sets the pattern.

```
'*
'* Set size of pen.
'*
SetLine:
   PENSIZE butSel, butSel                    'butSel equals 1, 2, 3, or 4
   RETURN
```

Figure 9-17. Setting the line width with *SetLine*

```
'*
'* Specify which pattern is to be used.
'*
SetPattern:
   MakePattern butSel
   RETURN
```

Figure 9-18. Selecting the pattern with *SetPattern*

The only thing left in the MiniPaint program is the set of DATA statements (Figure 9-19) that stores the hexadecimal numbers describing the optional patterns. These numbers are arranged in groups of four, for clarity, but as far as the program is concerned you can put as many on a line as you want—the program considers all DATA statements to be one large storage area. The first READ statement in the program reads the first piece of information, the second READ statement reads the second piece of information, and so on. To change one of the patterns used by the program, simply edit the information in the appropriate DATA statement here.

Suggestions for experimenting

If you feel you understand this program fairly well, you might try to integrate the pattern-generating program into it. Perhaps double clicking on a pattern could bring up the routine to create a new pattern that will replace the existing one. You would probably want to store the patterns in a disk file, rather than in DATA statements.

```
'*
'* Data statements for patterns.
'*
DATA &HFFFF, &HFFFF, &hFFFF, &hFFFF
DATA &H55AA,&H55AA, &H55AA, &H55AA
DATA &H0000,&H0000, &H0000, &H0000
DATA &H1188, &H4422, &H1188, &H4422
DATA &H8040, &H4090, &H0902, &H0201
DATA &H82AA, &H8244, &H4444, &HAA92
DATA &H0044, &H0000, &H0088, &H0000
```

Figure 9-19. The DATA statements that define the patterns

```
'* MiniPaint, a diminutive version of MacPaint
'*

   GOSUB DefineVariables
   GOSUB CreateWindows
   GOSUB CreateSymbols
   GOSUB CreatePatterns
   GOSUB ShowDefaults
   ON DIALOG GOSUB SelectWindow                    'if inactive window clicked
   DIALOG ON

'*
'* Allow user to create lines and shapes, while waiting
'* for click in side windows.
'*
MainLoop:
   WINDOW 4                                        'make window #4 active
   WHILE MOUSE(0) = 0

      '*
      '* While waiting for mouse click, store current location of
      '* pointer. This information will be used if next click is outside
      '* work-area window.
      '*
      yCord = MOUSE(2)
   WEND
   ON function GOSUB Rect, Oval, Lin
   GOTO MainLoop

'*
'* Draw rectangle.
'*
Rect:
   PENMODE 10                                      'XOR mode

   '*
   '* Draw first rectangle.
   '*
   MakePattern 1                                   'use black pattern for frame
   SetRectangle oldBord(), MOUSE(3), MOUSE(4), MOUSE(1), MOUSE(2)
   FRAMERECT VARPTR(oldBord(0))                    'draw rectangle
   WHILE MOUSE(0) = -1                             'while mouse button is pressed
      SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
```

Figure 9-20. The complete MiniPaint program

more...

```
    FRAMERECT VARPTR(oldBord(0))                    'erase rectangle (while dragging)
    FRAMERECT VARPTR(bord(0))                       'draw new rectangle
    CopyRect oldBord(), bord()
  WEND

  '*
  '* Create filled rectangle if this symbol was selected.
  '*
  MakePattern but(3)                                'reinstate stored pattern
  PENMODE 8                                         'copy mode
  FRAMERECT VARPTR(bord(0))                         'draw rectangle
  IF fill THEN PAINTRECT VARPTR (oldBord(0))

  '*
  '* And now for a little fun.
  '*
  WHILE MOUSE(0) = -2
    PENMODE 10
    SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
    FRAMERECT VARPTR(bord(0))
    IF fill THEN PAINTRECT VARPTR(bord(0))
  WEND
  PENMODE 8
  RETURN

'*
'* Draw oval.
'*
Oval:
  PENMODE 10
  MakePattern 1                                     'use black pattern for rectangle
  SetRectangle oldBord(), MOUSE(3), MOUSE(4), MOUSE(1), MOUSE(2)
  FRAMEOVAL VARPTR(oldBord(0))
  WHILE MOUSE(0) = -1
    SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
    FRAMEOVAL VARPTR(oldBord(0))
    FRAMEOVAL VARPTR(bord(0))
    CopyRect oldBord(), bord()
  WEND
  PENMODE 8
  MakePattern but(3)                                'reinstate selected pattern
  FRAMEOVAL VARPTR(bord(0))
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
    IF fill THEN PAINTOVAL VARPTR(oldBord(0))
    WHILE MOUSE(0) = -2
      PENMODE 10
      SetRectangle bord(), MOUSE(3), MOUSE(4), MOUSE(5), MOUSE(6)
      FRAMEOVAL VARPTR(bord(0))
      IF fill THEN PAINTOVAL VARPTR(bord(0))
    WEND
    PENMODE 8
    RETURN

'*
'* Draw line.
'*
Lin:

    '*
    '* Move pen point to beginning of drag.
    '*
    IF MOUSE(0) = -1 THEN MOVETO MOUSE(3), MOUSE(4)
    WHILE MOUSE(0) = -1                          'while mouse is being dragged
      point1 = MOUSE(5)                          'ending coordinate is also current
      point2 = MOUSE(6)                          'coordinate as long as button is down
      LINETO point1, point2                      'draw line
      MOVETO point1, point2                      'move pen to end of line
    WEND                                         'button released

    '*
    '* Mouse double clicked before drag.
    '*
    WHILE MOUSE(0) = -2
      PENMODE 10
      MOVETO MOUSE(3), MOUSE(4)
      LINETO MOUSE(5), MOUSE(6)
    WEND
    PENMODE 8
    RETURN

'*
'* Define variables and dimension arrays.
'*
DefineVariables:
    DEFINT a - z                                 'make all variables integers for speed
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
    DIM div(3)                                  'number of window division
    DIM but(3)                                  'current button in each window
    DIM pat(28)                                 'pattern definitions
    DIM bord(3)                                 'top, left, bottom, and right borders
    DIM oldBord(3)
    false = 0
    true = -1
    div(1) = 6                                  'number of divisions in shape window
    div(2) = 4                                  'number of divisions in line-thickness window
    div(3) = 7                                  'number of divisions in pattern window
    function = 1                                'default shape to draw--rectangle
    RETURN

'*
'* Create four windows.
'*
CreateWindows:
    WINDOW 1, , (0, 20) - (50, 209), 3          'shape window
    WINDOW 2, , (0, 210) - (50, 342) ,3         'line-thickness window
    WINDOW 3, , (460, 20) - (512, 342), 3       'pattern window
    WINDOW 4, , (51, 20) - (459, 342), 3        'work-area window
    RETURN

'*
'* Create window divisions.
'*
CreateSymbols:
    FOR windo = 1 TO 3
        WINDOW windo                            'make window windo active
        wide = WINDOW(2)                        'width of current window
        high = WINDOW(3)                        'height of current window
        FOR division = 1 TO div(windo) - 1      'draw division lines
            MOVETO 0, division * high \ div(windo)
            LINETO wide, division * high \ div(windo)
        NEXT division
    NEXT windo

    '*
    '* Draw symbols in shape window.
    '*
    WINDOW 1                                    'shape window
    high = WINDOW(3)
    wide = WINDOW(2)
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
'*
'* Call subprogram that creates six shapes.
'*
DrawFuncs wide, high, div(1)

'*
'* Create line-thickness window.
'*
WINDOW 2
high = WINDOW(3)
wide = WINDOW(2)

FOR division = 1 TO 4
   PENSIZE 1, division
   MOVETO 5, (division - .5) * high \ div(2)                    'note parentheses
   CALL LINE (wide - 10, 0)
NEXT
RETURN

'*
'* Read data for patterns.
'*
CreatePatterns:
   WINDOW 3                                                    'make window #3 active
   wide = WINDOW(2)                                            'get its height and width
   high = WINDOW(3)
   FOR design = 0 TO 27
      READ pat(design)                                         'read DATA statement
   NEXT
   countBy4 = 0                                          'initialize--used to count by fours
   FOR division = 0 TO div(3) - 1                                'fill patern swatches
      SetRelRect bord(), 0, division * high \ div(3), wide, high \ div(3)
      FILLRECT VARPTR(bord(0)), VARPTR(pat(countBy4))
      countBy4 = countBy4 + 4                                  'increment counter
   NEXT
   RETURN

'*
'* Highlight default selection in each window.
'*
ShowDefaults:
   inset = 2                                          'size of border around highlight
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
    FOR windo = 1 TO 3
      IF windo = 3 THEN inset = 10
      WINDOW windo
      wide = WINDOW(2)
      high = WINDOW(3)
      SetRectangle bord(), inset, inset, wide - inset, high \ div(windo) - inset
      INVERTRECT VARPTR(bord(0))                          'invert center of selection
      but(windo) = 1                                      'store this window's button-press
    NEXT
    RETURN

'*
'* Routine branched to if dialog event is trapped
'*
SelectWindow:
    IF DIALOG(0) <> 3 THEN RETURN                         'inactive window clicked
    WINDOW DIALOG(3)                                      'make clicked window active
    windo = WINDOW(0)                                     'active window
    wide = WINDOW(2)                                      'width
    high = WINDOW(3)                                      'height
    GOSUB SelectItem                                      'get option selection
    WINDOW 4                                              'make window #4 active
    ON windo GOTO SetShape, SetLine, SetPattern           'implement selection

'*
'* Select option from shape window.
'*
SetShape:
    IF butSel = 6 THEN CLS : RETURN
    IF butSel = 5 THEN function = 3 : RETURN
    fill = true
    IF butSel = 4 THEN function = 2 : RETURN
    IF butSel = 2 THEN function = 1 : RETURN
    fill = false
    IF butSel = 3 THEN function = 2 : RETURN
    function = 1 : RETURN

'*
'* Set size of pen.
'*
SetLine:
    PENSIZE butSel, butSel                                'butSel equals 1, 2, 3, or 4
    RETURN
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
'*
'* Specify which pattern is to be used.
'*
SetPattern:
   MakePattern butSel
   RETURN

'*
'* Determine which option was clicked.
'*
SelectItem:
   inset = 2                                                    'border for highlight
   IF windo = 3 THEN inset = 10                                 'window #3
   IF windo = 2 THEN yCord = yCord - 190                        'window #2
   butSel = ((div(windo) * yCord \ high) + 1)                   'increment down from top
   IF windo = 1 AND butSel = 6 THEN RETURN                      'don't highlight erase button
   top = (but(windo) - 1) * high \ div(windo) + inset
   SetRelRect bord(), inset, top, wide - 2 * inset, high \ div(windo) - 2 * inset
   INVERTRECT VARPTR(bord(0))                                   'return previous selection to normal
   top = (butSel - 1) * high \ div(windo) + inset
   SetRelRect bord(), inset, top, wide - 2 * inset, high \ div(windo) - 2 * inset
   INVERTRECT VARPTR(bord(0))                                   'invert center of new selection
   but(windo) = butSel
   RETURN

   '*
   '* Data statements for patterns.
   '*
   DATA &HFFFF, &HFFFF, &hFFFF, &hFFFF
   DATA &H55AA,&H55AA, &H55AA, &H55AA
   DATA &H0000,&H0000, &H0000, &H0000
   DATA &H1188, &H4422, &H1188, &H4422
   DATA &H8040, &H4090, &H0902, &H0201
   DATA &H82AA, &H8244, &H4444, &HAA92
   DATA &H0044, &H0000, &H0088, &H0000

'*
'* Start of subprogram section.
'*
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
'*
'* Make one of stored patterns active.
'*
SUB MakePattern(patNum) STATIC
   SHARED pat()
   PENPAT VARPTR(pat(4 * patNum - 4))                    'Pat() is first of group of four
END SUB

'*
'* Copy one rectangle into another.
'*
SUB CopyRect(rect1(), rect2()) STATIC
   FOR count = 0 TO 3
      rect1(count) = rect2(count)
   NEXT
END SUB

'*
'* Draw symbols in shape window.
'*
SUB DrawFuncs (wide, high, numDivs) STATIC
   DIM Rect(3)
   boxWidth = wide - 10 : boxHeight = high \ numDivs - 9
   deltaY = high \ numDivs

   '*
   '* Draw rectangle.
   '*
   SetRelRect Rect(), 5, 5, boxWidth, boxHeight
   FRAMERECT VARPTR(Rect(0))

   '*
   '* Draw filled rectangle.
   '*
   SetRelRect Rect(), 5, deltaY + 5, boxWidth, boxHeight
   INVERTRECT VARPTR(Rect(0))

   '*
   '* Draw circle.
   '*
   SetRelRect Rect(), 5, 2 * deltaY + 5, boxWidth, boxHeight
   FRAMEOVAL VARPTR(Rect(0))
```

Figure 9-20. The complete MiniPaint program *(continued)*

more...

```
    '*
    '* Draw filled circle.
    '*
    SetRelRect Rect(), 5, 3 * deltaY + 5, boxWidth, boxHeight
    INVERTOVAL VARPTR(Rect(0))

    '*
    '* Draw crooked line.
    '*
    LINE (6, 4.3 * deltaY) - STEP (10, 10)
    LINE - STEP (10, -10)
    LINE - STEP (10, 10)

    '*
    '* Draw eraser.
    '*
    MOVETO 6, 6 * deltaY - 10
    PRINT "erase";
END SUB

'*
'* Take pair of points and set rectangle so it encloses these points.
'*
SUB SetRectangle(array(), x1, y1, x2, y2) STATIC
    array(0) = y1
    array(1) = x1
    array(2) = y2
    array(3) = x2
    IF x1 > x2 THEN SWAP array(1), array(3)
    IF y1 > y2 THEN SWAP array(0), array(2)
END SUB

'*
'* SetRelRect is just like SetRectangle except it takes  as input
'* top, left point and height and width.
'*
SUB SetRelRect(array(), x, y, wide, high) STATIC
    CALL SetRectangle (array(), (x), (y), x + wide, y + high)
END SUB
```

Figure 9-20. The complete MiniPaint program *(continued)*

SECTION III

Communications

III

Introduction to Communications

Chapter 10

The application programs available for your Macintosh make it a useful and entertaining tool. But just as individuals can increase their knowledge and power through association with the rest of the community, the power of your Macintosh—and the scope of the information you can process with it—can be extended through communication with other computers. Connecting your computer to others allows you to search for and retrieve information from commercial databases, exchange electronic mail, send telegrams and telex messages, check airline fares and schedules and purchase tickets, manage your bank account, monitor and control conditions at remote sites, and participate in many other useful and entertaining activities.

The only items, in addition to your Macintosh, that you need to gain access to the world of electronic information and services are a telephone line, a modem, and communication software.

Modems and the modem port

Before getting too involved with the software that allows your computer to communicate, let's have a look at the hardware end of things, and at how you physically hook your Macintosh to a modem or another computer.

If you look at the back of your Macintosh, you will see four sockets, called ports, in a row at the bottom. As you undoubtedly know, these are for connecting, from left to right, the mouse, the external drive, the printer, and the modem. (The modem port is also called the communication port or COM1:.) The printer and modem ports are physically identical; either could be used to communicate with a printer, a modem, a hard disk, the AppleBus, the Microsoft MacEnhancer, or just about any other device that normally connects to the serial port of a computer. However, differences in the

way the operating system handles information received at the printer port restrict the speed at which you can communicate through it, so for practical purposes, telecommunication or communication with other computers is limited to the modem port.

There are two standard methods—serial and parallel—by which a computer communicates with other devices. Very briefly, serial communication sends a stream of single bits that are grouped together at the receiving end to make characters; parallel communication sends eight bits—the equivalent of one character—at a time. The Macintosh uses serial communication.

Now that the Mac has been out for a while, modems and other devices advertised as "Macintosh compatible" or "for the Macintosh" are appearing on the market. This sounds like a breakthrough, as if the manufacturers had to develop a special modem to allow your Macintosh to communicate; but the fact of the matter is that almost any serial device can be connected to the Macintosh. There are serial versions of almost every computer peripheral, including modems, printers, plotters, and data acquisition equipment. Your Macintosh can even connect directly to an IBM PC or any other computer that has a serial port. The only "interface" required between the Mac and the device with which it is communicating is a cable with the proper connector on each end.

The connectors

Physically, the Mac's serial-port connectors are somewhat smaller than the connectors you may be accustomed to seeing on the serial ports of other computers. The standard system for connecting serial devices has for years been the RS-232C protocol. (A communication protocol is a set of rules that establish a standard for interconnecting devices. Computer manufacturers are not *required* to follow any particular communication protocol when designing their equipment, but doing so usually adds to their product's popularity.) RS-232C devices connect through a 25-pin connector, commonly called a DB-25 connector.

The Macintosh is not an RS-232 device. Its serial ports follow the RS-422 protocol, which is an enhanced version of RS-232 that allows higher-speed communication and greater distance between devices. The connector used by the Macintosh has nine pins and is commonly called a DB-9 connector. It can easily be hooked to most

serial devices—even those using a different protocol—with the proper cable. Your Macintosh dealer can probably supply a cable with the proper connectors to hook to any other device you purchase, but if you are into being self-sufficient or saving money, you can assemble your own cables with relatively little effort.

The nine-pin connectors at each of the Macintosh serial ports have the following signals on them:

Pin	Signal
1	Ground
2	+5 volts
3	Ground
4	TXD+ (transmitted data)
5	TXD−
6	Filtered +12 volts
7	Handshake for printer or carrier detect for modem; also for external clock in synchronous communication mode
8	RXD+ (received data)
9	RXD−

Of these, the minimum required for communication are a ground, a transmit, and a receive signal. The signals available on pins 3, 5, and 9 of the Macintosh modem port satisfy this requirement, and correspond to pins 7, 2, and 3 on an RS-232 (DB-25) connector. Some devices also require a carrier-detect signal from pin 7 of the Macintosh. The connectors and wire required are readily available and not too expensive. Radio Shack, for example, stocks both the DB-9 and DB-25 connectors under the following part numbers:

DB-9	Male	276-1537
	Female	276-1538
DB-29	Male	276-1547
	Female	276-1548

The wiring diagrams in Figure 10-1 on the next page show the connections for cables used to hook the Macintosh to other specific devices.

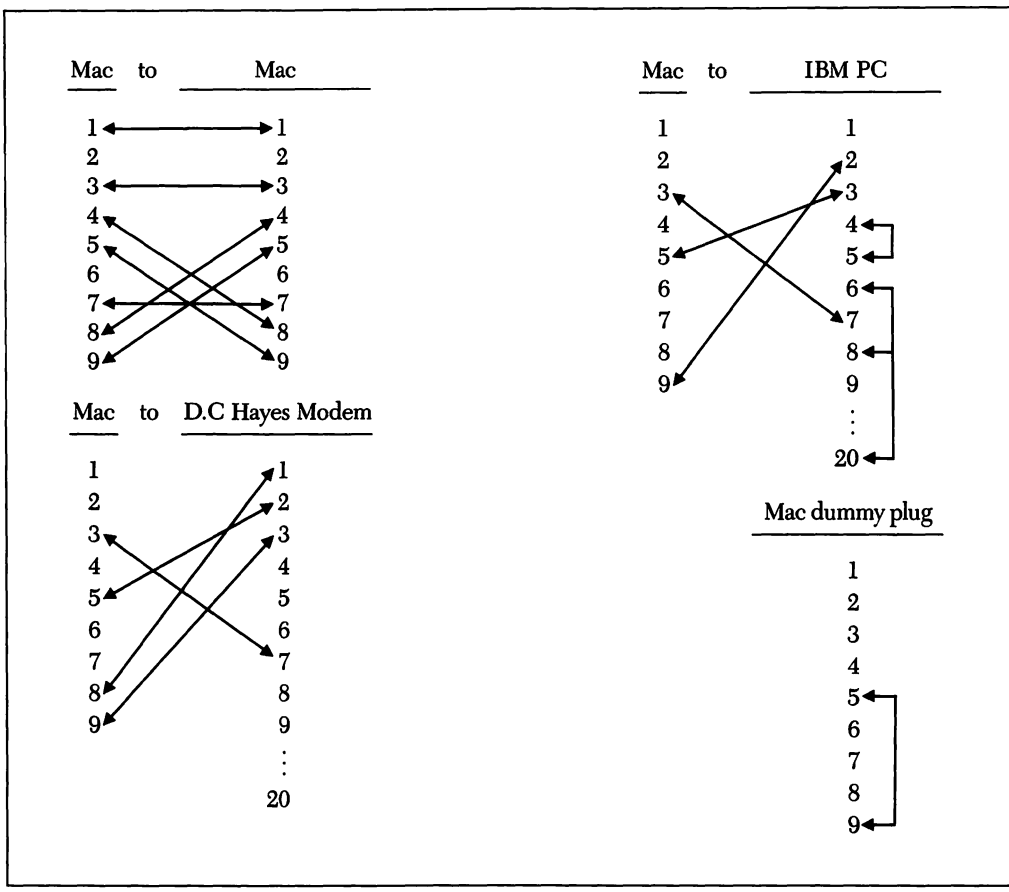


Figure 10-1. Various cable connections

Communication programs

Programs that allow the Macintosh to communicate with other computers come in many sizes and shapes, and at a variety of prices. They share the common quality of being able to send the characters you type at the keyboard out the modem port at the back of the Mac, and of transferring the characters that come in the modem port to the screen, but other than that, they may differ greatly.

To really take advantage of the power of your Macintosh, sophisticated software is usually required, but because this is a book about BASIC and not a book about communications per se, I'm going to limit the discussion of communication programs in

this chapter to the relatively simple terminal emulation program, called *Terminal,* that Microsoft has included on the BASIC disk. This program—written, naturally, in BASIC—does nothing more than allow your powerful Macintosh to act like a simple terminal. (A terminal is a keyboard/screen combination that has no computing power and no internal or external storage for the information that passes through it.)

There are many situations in which this level of communication is adequate. For example, if you can connect your Macintosh to the office computer and use its storage and computational power, then simply being able to enter information from home or while traveling may be useful. Also, many of the electronic services, such as directing your bank to pay a bill, or retrieving airline scheduling information, can be utilized with a simple terminal.

In addition to providing terminal emulation capability, I like to think that Terminal was provided to help you learn more about BASIC. And as a learning tool, it is ideal: It is well written and documented, it uses several commands unique to the Macintosh, it performs a simple task in an understandable manner, and it can be expanded almost infinitely. In Chapter 11, we will tear the Terminal program apart, and learn what it does and how it does it. Then in Chapter 12, we will add some features to make it more useful. By the time you finish Chapter 12, you should understand the Terminal program well enough to tailor it to your own needs. For now, let's finish this chapter with a brief discussion of some of the services that will be available to you once you have your modem connected and your communication program up and running.

Let your fingers do the walking

The electronic services to which you can connect your Macintosh vary in cost from nothing (or just the cost of a telephone call) to several hundred dollars an hour. I'll just mention the bargains.

Bulletin boards

The most common free services are bulletin boards—computerized versions of the cork-board at the neighborhood grocery store, where people post notices about things to buy or sell, and carry on "pen-pal" type relationships with other computerists. If you are new to telecommunication and would like a cheap way to try out your

system, and perhaps meet a few people with common interests, you should try out a few bulletin boards. The phone numbers of local bulletin boards can usually be obtained from someone at a computer store or in a users'-group meeting.

A communication sampler: *Online*

The commercial services (those that charge money) offer such a vast variety of information and service that it is difficult to decide which to subscribe to. If you would like to try out a few of these services for free, I recommend you buy my book: *Online* (Microsoft Press, 1985). Included with the purchase of this book are subscriptions to six major services, and free time on one more. As an example of what is available via a modem, here's a summary of the services offered through *Online.*

MCI Mail specializes in what is often called "store and forward" electronic mail delivery. Your subscription to MCI Mail allows you to send messages to other subscribers—anything from a short note to long documents created with your word processor. Mail sent electronically is instantly available in recipient MCI Mailboxes. The next time the people you have sent messages to log on, they are notified of mail waiting and can read it or transfer it directly to a computer file.

If the intended recipient is not a subscriber, MCI offers the option of telecommunicating your message to a facility as close to the recipient as possible, printing the message, placing it in an envelope, and then delivering it by courier or turning it over to the U.S. Postal system, as you desire.

CompuServe is an information utility, offering a broad range of topics at a low price. CompuServe's Executive Information Servise (EIS) offers in-depth coverage of the financial community, access to a variety of news services, statistical information, shopping, and communication—both electronic mail and an online computer conferencing service.

NewsNet offers the publishers of newsletters the opportunity to make their product instantly available online to readers. When you call NewsNet, you tap into a huge database containing years of subscriptions to hundreds of newsletters. Every key word of every newsletter is electronically indexed; enter a word or phrase that interests you and NewsNet will tell you which newsletter issues include references to it. You can then select and read specific articles or headlines.

If you have a continuing interest in a particular subject, NewsNet will monitor this subject for you and tell you, each time you log on, if new information has been added to the database since your last session.

Official Airline Guides (OAG) is a database containing fares and schedules for over 700 airlines throughout the world. With a quick call to OAG, you can find all direct or connecting flights between any two cities, including departure and arrival times and specific fares. You can discover the "specials" and excursion fares, and see what restrictions apply to them.

Western Union is a name that brings up visions of old men dinging the bells on their bicycles as they pedal through traffic to deliver urgent telegrams. But the delivery "boys" are gone: Western Union has enthusiastically entered the electronic era.

Your subscription to Western Union's EasyLink service provides a link to the people on the other side of 1.6 million telex machines; allows you to send telegrams, mailgrams, cablegrams, and ties you into an electronic mail network with 110,000 other EasyLink subscribers—all without leaving your computer keyboard. Western Union also provides a currrent affairs news service.

DIALOG is the granddaddy of all databases. Actually, DIALOG is a vendor of database information; it has gathered together hundreds of specialized databases, covering almost every imaginable subject, and provides an organized method to access all of them.

Dow Jones News/Retrieval Service is a subsidiary of the company that publishes *The Wall Street Journal.* Although it specializes in stock-market information, it also makes available to its subscribers an array of business and general-interest information and services—everything from Wall Street news to movie reviews.

These services are just a sample of what is available online. There are thousands of sources of information and services with which you can communicate using your computer. Now that you know *what* you are going to do with telecommunication, let's have a look at a simple program that helps you *do* it.

The Terminal Program

Chapter 11

The Terminal program supplied by Microsoft on the BASIC disk is a relatively simple program. It occupies about 4K bytes on your disk, and when printed fills about two and a half pages. By contrast, the powerful PC-Talk III, a communication program written in BASIC for the IBM PC, is about 45K bytes long and fills 19 pages. They both perform the same primary function: the exchange of information between two computers. PC-Talk III simply offers more refinements and options. In this chapter we will learn how the Terminal program operates, and in the next look at possible enhancements to it. By the time you finish this section, you should be able to create a communication program tailored exactly to your own needs.

Since the Terminal program is on your disk and ready to run, go ahead and try it out before we get into the explanation of how it works. (I assume, since you are reading this chapter, that you have either a modem or access to a second computer to which you can connect your Macintosh.)

The first thing you will notice as the Terminal program starts to run is that it replaces BASIC's menu bar with one of its own. There are only two selections on this menu bar, with only one item on each menu. Figure 11-1, on the next page, shows the screen after you choose Set Configuration Parameters from the Configure menu. The dialog box that appears allows you to set the baud rate, parity, number of data bits, and number of stop bits used to communicate, by clicking a button opposite each parameter. The default parameters are indicated by black dots in their buttons.

If, when you start the program with your modem connected and turned on, these default settings are satisfactory, you can simply dial the phone number of the computer you want to connect with and start communicating. Everything you type will go out the modem port, and anything that comes in the modem port will appear

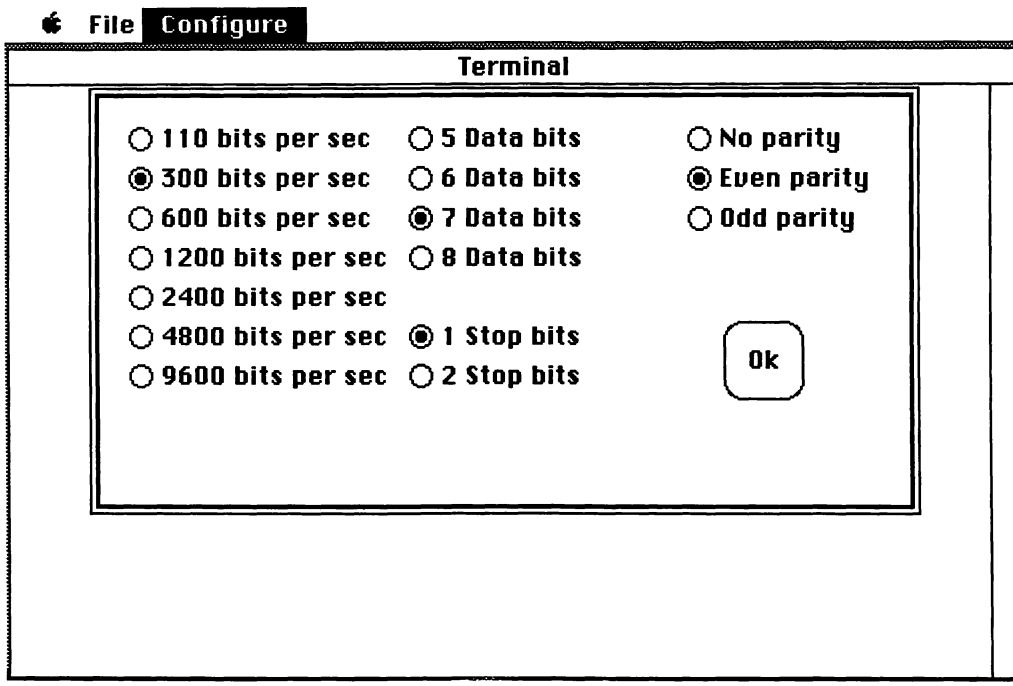


Figure 11-1. The Terminal configuration screen

on the screen. Normally, what you type will also appear on the screen, echoed back by either your modem or the computer with which you are communicating.

Other than the configuration dialog box and the text that you send or receive, there is not much to look at when you run this program, but it does its simple job well. Let's read through the program and discuss the new commands. After you understand how this fundamental program works, we will create a more powerful communication program based on the same concepts.

The complete Terminal program is shown in Figure 11-13 at the end of the chapter. For the purposes of this discussion, I have divided Terminal into sections, and in the next few pages, I list and briefly describe each section in the order in which it would typically be used, not the order in which it appears on your disk. Figure 11-2 shows the flow of the program. Once it enters the communication loop, it stays there unless it is interrupted to handle a menu request, which in turn leads it into one or more of the subroutines and possibly the subprogram. When the peripheral activities are taken care of, the program returns to the communication loop.

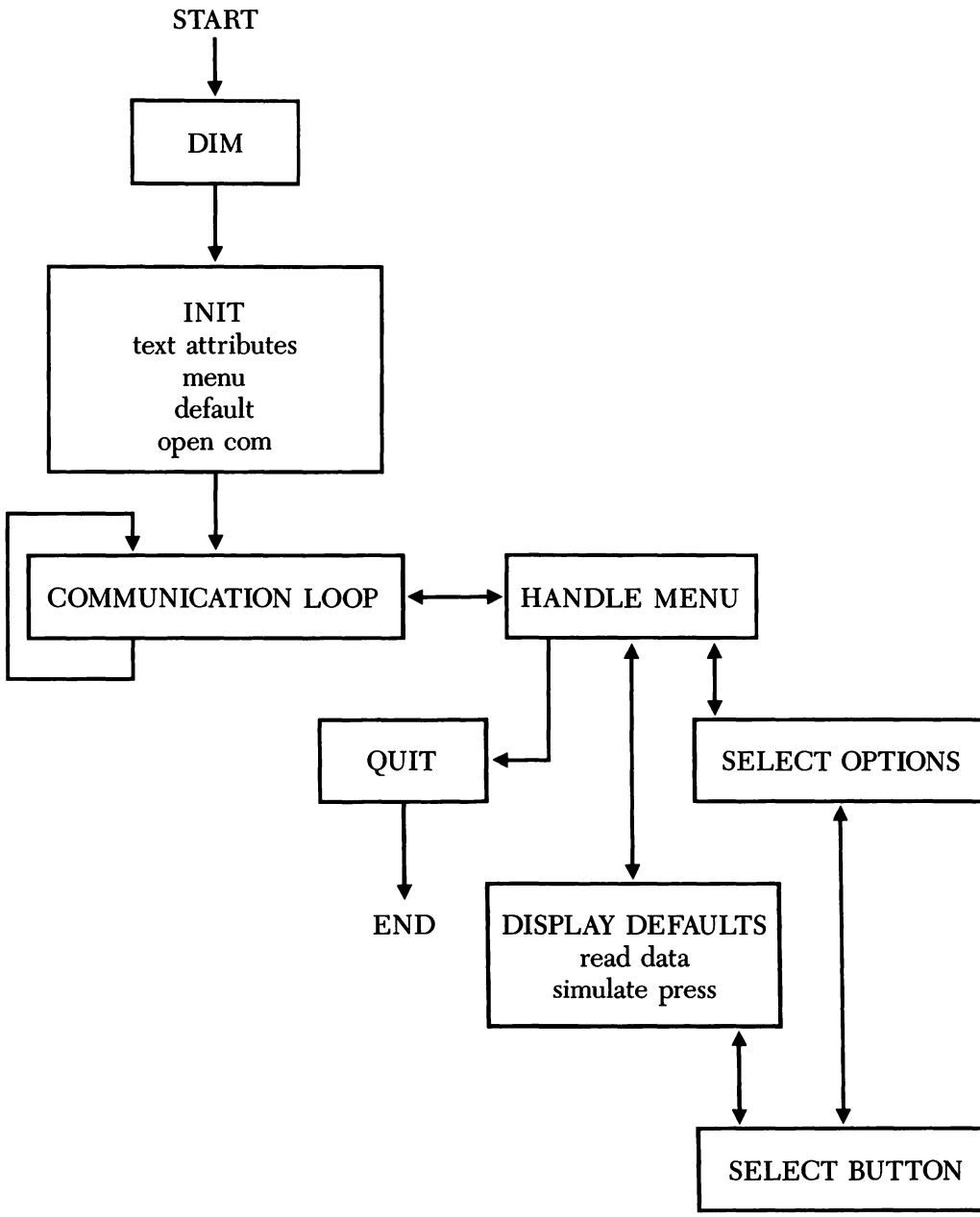


Figure 11-2. A flow chart of the Terminal communication program

```
REM --- Terminal
REM --- Terminal Emulation Program

  DIM nam$(17), group(17), choice(4), choice$(4)
  GOSUB Init
```

Figure 11-3. Dimensioning the arrays

You should be familiar with most of the commands in this program, and the comments will help refresh your memory. (Notice that REM statements are used for the comments in this program, instead of the single-quote/asterisk convention I usually use.) If, after glancing through the listing, you feel you already understand the commands used and would like to get on with enhancing Terminal to make it a more powerful program, feel free to skip ahead to the next chapter.

Dimensioning the arrays

Figure 11-3 lists the section of the Terminal program that identifies the program (in the initial comments) and sets aside storage space for array variables. As you can see, the program uses four arrays: *nam$(17), group(17), choice(4),* and *choice$(4).* Remember that the number in parentheses after each variable name is the highest subscript that may be used with that variable, and that 0 is the lowest, so you can have one more variable in each array than the value given in the DIM statement.

Once the arrays have been dimensioned, the GOSUB statement diverts program flow around the main communication loop (labeled *Loop:*) to the *Init* subroutine.

Initializing the program

The *Init* subroutine, as its label implies, sets up the initial, or default, conditions of the program. It establishes the text font that will be displayed on the screen, creates the menus and their selections, specifies the default communication parameters, and opens the communication port. Read through the listing in Figure 11-4 on the next page, and then we will take a closer look at each line.

```
Init:
   TEXTFONT 4                                          'mono-spaced font
   TEXTSIZE 9                                          'allows 80 characters per line
   TEXTMODE 1                                          'print mode = XOR, not COPY

   REM --- Setup menu
   MENU 1, 0, 1, "File"
   MENU 1, 1, 1, "Quit"
   MENU 2, 0, 1, "Configure"
   MENU 2, 1, 1, "Set configuration parameters"
   MENU 3, 0, 0, ""
   MENU 4, 0, 0, ""
   MENU 5, 0, 0, ""
   ON MENU GOSUB HandleMenu
   MENU ON

   REM --- Setup default options
   choice(1) = 2                                       '300 baud
   choice(2) = 9                                       'even parity
   choice(3) = 13                                      '7 data bits
   choice(4) = 15                                      '1 stop bit
   REM --- Open Communications port with 2000 byte input buffer
   OPEN "COM1: 300, e, 7, 1" AS 1 LEN = 2000
   RETURN
```

Figure 11-4. Initializing the program

You are familiar with the TEXTFONT and TEXTSIZE ROM calls from earlier chapters. They set the font and character size used for printing in the output window. The TEXTMODE call controls whether the pixels that make up new text on the Macintosh screen replace old pixels at the same location or are displayed on top of them in some manner. The two possible syntaxes for this call are:

CALL TEXTMODE (*mode*)

TEXTMODE *mode*

The *mode* argument is a numeric expression from 0 through 3. Figure 11-5, on the next page, demonstrates the effect of each mode.

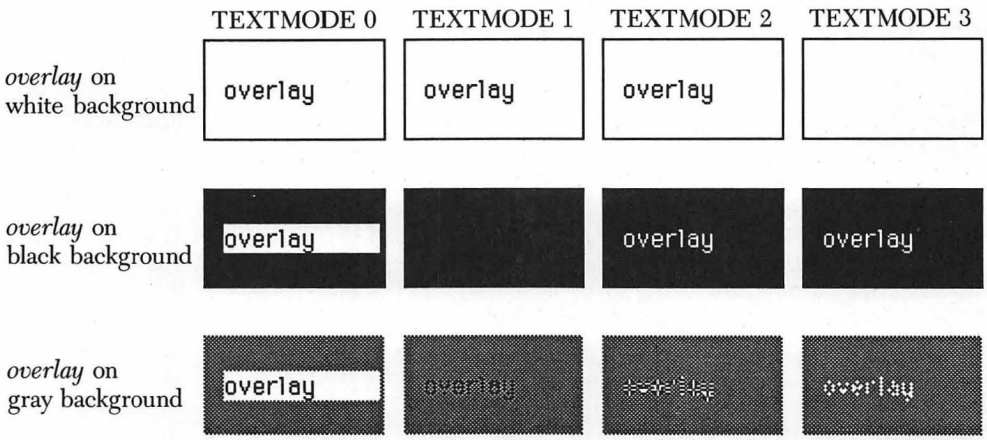


Figure 11-5. Text modes

If you don't specify a mode, BASIC defaults to mode 0, in which new text, black on a white background, replaces old. Mode 1 ORs new text with old, meaning that all pixels turned on for either new or old text remain turned on. Mode 2 XORs the new with the old, meaning that any pixel that would be turned on for the new text that has already been turned on for the old text is inverted in color (turned off) and appears white. Mode 3 changes only black pixels of the old text, making them white where the new text appears.

The next block of commands provides the power to personalize your program by creating a menu bar and dropdown menus. This part of the Mac's visual interface is a distinctive feature of commercially available programs for the Macintosh, and the BASIC MENU command allows you to create and control it.

A menu is actually dealt with in two stages. The first stage, which you see in this section, creates the menu and specifies the selections and the state (disabled or enabled) of each. The second stage, discussed in a moment, defines what happens when the user selects an item from the menu.

There are nine commands in BASIC that include the word MENU. Each has a different syntax and results in a different action being taken. Here, we use three MENU commands. The first of these is the statement that actually creates menu items. Its syntax is:

MENU *menu-ID, item-ID, state [title]*

Here is what the italicized arguments in this statement mean:

Argument	Value and meaning
menu-id	A number from 1 through 10, identifying position of menu in menu bar (from left to right).
item-id	A number from 0 through 20, identifying item in menu (menu title is item 0; selections in menu are numbered 1 through 20).
state	A number from 0 through 2 that determines whether item referenced in this statement is disabled (0), enabled (1), or enabled and preceded by check mark (2).
title	A string of text, enclosed in quotation marks, assigned to referenced item and appearing either as menu title or as item on menu.

Let's take a closer look at how the Terminal-program menus are created. When you ran the program, you saw that there were three menus—Apple, File, and Configure. The latter two are created by the program and each displays one item. (You can have as many as 10 BASIC-created menus with up to 20 items on each.) The desk accessory menu under the Apple icon is a permanent fixture that we can neither turn off nor control from BASIC, so it has no number. You issue this version of the MENU command once for each menu title and each selection item you want to define. Glancing at the block of MENU commands in the Terminal program, you can see that five menus, in addition to the Apple menu, are defined. The three that you did not see when you ran the program (numbers 3, 4, and 5), are "dummy" menus, included in the program as a way of turning off the standard BASIC menus in those locations (remember that this program is run from within BASIC, so the BASIC menu bar appears at the top of the screen until you change it).

Once a menu is created, the program uses another of the nine MENU commands, the ON MENU statement, to tell the Macintosh what to do when an item is selected from the menu. In this case *ON MENU GOSUB HandleMenu* sends the program off to a subroutine for more instructions. The ON MENU statement is one of a group of ON... statements that set up the action to be taken should some event occur in the future. The Macintosh stores this information and, once trapping for that event is

enabled, watches for it as the program runs, taking the specified action if the event occurs. The subsequent MENU ON statement, the third version of MENU that we'll meet in this chapter, enables menu event trapping. From this point on, if a menu item is chosen, program flow is diverted to the *HandleMenu* subroutine.

The next part of this section of code assigns values to four elements of the *choice* array, which was dimensioned in the first section. Later you will see how these settings are used to display the default communication parameters when we arrive at the *DisplayDefaults* subroutine.

The final step in the initialization process is to open the communication port. In addition to opening sequential and random-access files, you can use the OPEN command to open I/O devices: the keyboard (KYBD:), the Clipboard (CLIP:), the screen (SCRN:), the printer (LPT1:), and the communication port (COM1:). The syntax of the OPEN command when used to open the communication port includes information to set the baud rate, parity, number of data bits, and number of stop bits. The specific command given in the Terminal program—*Open "COM1: 300, e, 7, 1" AS 1 LEN=2000*—sets the baud rate to 300, the parity to even, the number of data bits to 7, and the number of stop bits to 1. The complete list of possible settings for these communication parameters is:

Parameter	Possible values
Baud rate	110, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, 19200, or 57600
Parity	Odd, even, or none
Data bits	5, 6, 7, or 8
Stop bits	1, 1.5, or 2

The *AS 1* portion of the command opens the port as file #1. The file number can be any integer from 1 through 255, and is essentially the address used when sending information to the file (PRINT *#filenumber*) or getting information from the file (INPUT$(*X, filenumber*)). The final part of the OPEN statement, *LEN=2000,* allocates 2000 bytes of memory as an input buffer in which to store data bits that come in faster than the program can accept them.

By the time the program has gotten this far, both it and the computer are initialized and ready to start communicating. So the RETURN statement sends the program back to the line following the GOSUB that called the *Init* subroutine, and the program flows on into the main communication loop, where it will remain until a menu item is chosen.

An exercise

One of the nice things about studying a working program already on your disk is that you don't have to type it in and worry about whether what you have typed will work. It is easy to load the program into BASIC and run it. You can then stop the program, list it, and make minor changes to check your understanding of the various commands. A simple change you can make to this section is to replace the menu titles with other words. You could even add a few more items to one of the menus, or add an entire new menu. As long as you don't save your changes when you quit, the original program won't be affected.

The main communication loop

Although the program is initially directed around the *Loop:* section by the *GOSUB Init* statement, *Loop* is the main part of the program and, as we will see later, can practically stand on its own as a useful program. The program accomplished one essential action during its diversion, however: It opened the communication port (COM1:) so that you can send and receive information.

The main communication loop, shown in Figure 11-6 on the next page, does just what the comment in its first line indicates. (Remember that remarks can be set off with either REM or a single quote mark.) It displays the characters arriving at COM1: on the screen, and sends the characters typed at the keyboard out the same port. The main loop is entered on the line labeled *Loop*. The next two lines monitor COM1: for incoming characters and print any that appear. The following two check to see if a key has been pressed since the last check, and if one has, the next-to-last line sends the character out COM1:. The last line directs the program back to *Loop* to repeat the sequence. As a matter of fact, it will repeat forever unless an outside event—in this case a selection from the menu—interrupts it.

```
'Display characters from COM1, send keystrokes to COM1
Loop:
   PRINT INPUT$(LOC(1), 1);
   IF LOC(1) > 0 THEN Loop
   k$ = INKEY$
   IF k$ = "" THEN Loop
   PRINT #1, k$;
   GOTO Loop
```

Figure 11-6. The main communication loop

Let's look more closely at the individual lines in this section, and I will explain the commands we haven't previously used.

```
PRINT INPUT$(LOC(1), 1);
```

You are already familiar with the PRINT statement. Used in this context, it will display the string represented by *INPUT$(LOC(1), 1)* on the screen. We already know that this string will be one or more characters waiting at COM1:, but how many characters are there and how are they collected? Well, you'll recall that the full syntax of the INPUT$ function is:

INPUT$(X[, [#]*filenumber*])

This function returns a string X characters long from the specified file (if no file number is specified, INPUT$ gathers its characters from the keyboard). In this case, the file specified is #1, the file number assigned to the communication port. The number of characters to gather, however, has been replaced by the function *LOC(1)*. The parenthetical number in this function is again a file number. The LOC function returns different types of information, depending upon the kind of file referenced by the file number. The number returned for COM1:, which we are using here, is the total number of characters waiting to be input (remember that the input buffer holds characters arriving faster than they can be displayed). So what this line in essence says, is: Find out how many characters are waiting at the communication port, bring them in, and print them on the screen.

The semicolon that ends the PRINT statement serves its usual purpose: It prevents an automatic carriage return, so that the next PRINT statement will display its characters directly after these, on the same line of the screen.

The next line instructs the computer to check the input port again and, if anything is waiting to come in, to direct the program flow back to *Loop,* which will input and print it.

```
IF LOC(1) > 0 THEN Loop
```

The purpose of this statement is to give incoming characters priority over those typed at the keyboard. If the previous line emptied the buffer but characters are still coming in, this line sends the program back to empty the buffer again. If there are no characters waiting to come in, the program continues to the next line.

```
k$ = INKEY$
```

This statement simply assigns the character returned by the INKEY$ function to the variable *k$.* INKEY$ returns either the next character from the keyboard buffer or, if the buffer is empty, a null string. Because it does not hold the waiting character, we assign the retrieved character to *k$* so that we can later test it and, if necessary, print it. The next line:

```
IF k$ = "" THEN Loop
```

instructs the computer to go back to *Loop* if there is no keystroke waiting in *k$.* (Double quotes with neither character nor space between them are the symbol for the null string.) If *k$* is not equal to the null string—in other words, if a character has been assigned to *k$*—the program goes on to the next line, without executing the THEN portion of the IF...THEN statement.

```
PRINT #1, k$;
```

Here we see yet another version of the PRINT statement. The *#1* included in this statement directs the printing to file #1, rather than to the screen. Again, note the closing semicolon which keeps the printed characters on the same line.

When you are using the Terminal program, it will seem that the characters you type are immediately printed on the screen. In reality, however, they are sent out the communication port to the computer you are hooked to, which (assuming it has been instructed to do so) echoes them back to your computer, in addition to whatever other uses it makes of them.

The final line of this section directs program flow back to the beginning of the loop, to repeat the cycle.

```
GOTO Loop
```

The constant checking of COM1: and the keyboard will continue until you stop the program by pressing Command-period, or make a selection from the menu, so you can see where this program will spend most of its time. The remainder of the program simply accomplishes the housekeeping tasks of displaying and managing the menus and allowing you to change the baud rate and other communication parameters.

Managing the menus

When an item is chosen from a menu, program flow is diverted to the *HandleMenu* subroutine shown in Figure 11-7. Since there is only one item on each menu, in this case determining the menu that was chosen also determines the item.

The MENU(0) function checks to see which menu was selected; if it was menu #1, the SYSTEM statement is issued to quit BASIC and return to the Macintosh desktop. If menu #1 was not selected, then #2 must have been, so the program assumes you want to change the communication parameters and closes file #1, displays a list of options, accepts your choices, and reopens the communication port with the new parameters. (It is unusual to have only two items to choose between in the entire menu.)

```
HandleMenu:
  IF MENU(0) = 1 THEN SYSTEM                              'got quit command
  CLOSE 1                                  'else it must be Set configuration parameters
  WINDOW 2,, (50, 50) - (450, 250), 2
  GOSUB DisplayDefaults
  GOSUB SelectOptions
  WINDOW CLOSE 2
  options$ = choice$(1) + "," + choice$(2) + "," + choice$(3) + "," + choice$(4)
  REM --- Open Communications port with 2000 byte input buffer
  OPEN "COM1:" + options$ AS 1 LEN = 2000
  CLS
  MENU 1, 0, 1
  RETURN
```

Figure 11-7. Managing the menus

The CLOSE statement puts away files opened with the OPEN statement. You can specify one or more files to close, separated by commas, or you can simply use CLOSE, with no file numbers, to close all open files. Files are closed automatically if you issue a CLEAR, END, NEW, RESET, or SYSTEM statement.

You have already used the WINDOW command in Chapter 4, so there is no need to discuss it in detail. The window created in this program is used as a dialog box, to display the communication-parameter options. These parameters are managed by the subroutines *DisplayDefaults* and *SelectOptions*, which will be discussed in a bit: They do essentially what their names imply. After the two GOSUBs, the program returns to *HandleMenu* and uses the statement *WINDOW CLOSE 2* to close the dialog window, which is no longer needed.

The two subroutines generate new values for elements 1 through 4 in the *choice$* array. These new values—baud rate, parity, number of data bits, and number of stop bits—are joined together as one long string named *option$*, which is then used to re-open COM1:. Creating one new string by using plus signs to "add" smaller strings together like this is called concatenation. The commas are characters in the new string, required to open the communication port, and are not a part of the concatenation process. If you selected 1200 baud, 8 data bits, 1 stop bit, and no parity, *option$* would look like this:

"1200,8,1,N"

Finally, in preparation for starting a new communication session, the screen is cleared with CLS and the previously highlighted menu title is reset to an active but unselected state.

Displaying the default parameters

HandleMenu calls the *DisplayDefaults* subroutine to display the communication-parameter options and mark the current selections in the dialog box created by its WINDOW statement. The values of these parameters are stored in the DATA statements that follow the subroutine. *DisplayDefaults* reads the DATA statements and assigns the values they contain to elements of the *nam$* and *group* arrays. It then uses these arrays to place the buttons in the dialog box. After the buttons are displayed, the *SelectButton* subprogram is called to simulate the selection of the buttons for the current communication parameters, *choice(1)* through *choice(4)*. Figure 11-8 shows the dialog box created, and Figure 11-9 lists the subroutine that produces the display.

You'll recall that there are several ways to provide a program with the information it needs to make decisions, create displays, and so on: You can store information in a disk file and retrieve it with an INPUT$, INPUT#, or LINE INPUT# statement; you can ask the user to supply it via the keyboard and retrieve it with an INPUT or

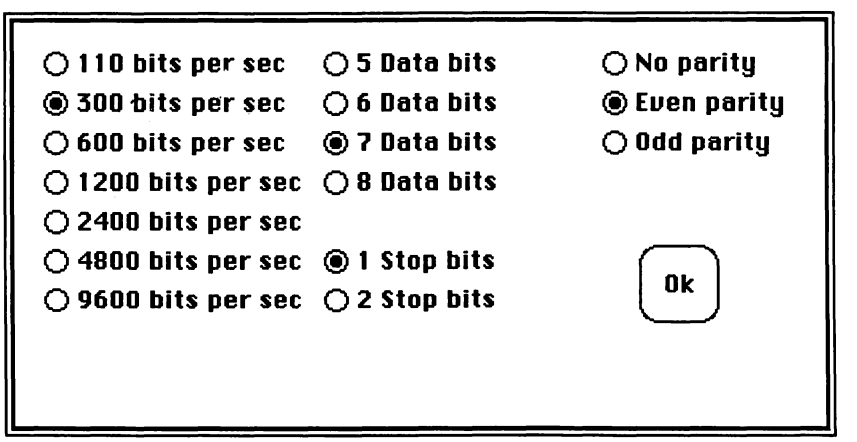


Figure 11-8. The default communication parameters

```
DisplayDefaults:
  REM *** Prompt user for Communications Parameters
  RESTORE
  FOR i = 1 TO 16
    READ x, y, group(i), nam$(i)
    BUTTON i, 1, nam$(i), (x, y) - (x + 135, y + 15), 3
    NEXT i
  BUTTON 17, 1, "Ok", (310, 110) - (350, 150)
  REM *** Simulate button pushes to highlight defaults
  FOR i=1 TO 4
    SelectButton choice(i)
    NEXT i
  RETURN
```

Figure 11-9. Displaying the default parameters

LINE INPUT statement or the INKEY$ function; or you can store it in the program itself in the form of DATA statements that can then be read as needed and assigned to variables. This third method is the one used here to store the default communication parameters and the (x, y) coordinates needed to properly position their buttons, since this information is not subject to change.

The RESTORE statement at the beginning of this subroutine determines the first DATA statement to be read. The syntax for this statement is:

RESTORE [*line*]

When used with neither line number nor label, as it is here, RESTORE causes the next READ statement to start at the first DATA statement in the program. If you include the line number or label of a specific DATA statement, the next READ will start with that statement.

After resetting the READ position to the beginning of the DATA statements, the program sets up a loop to read the 16 DATA statements and assign the four pieces of information in each to four variables. The 16 DATA statements, taken from top to bottom, correspond to the 16 buttons in the dialog box. You are already familiar with the FOR...NEXT statement: This one will cycle 16 times, executing the READ and BUTTON statements each time. The first time through the loop, *i* (the counter in the FOR...NEXT loop) is equal to 1. The READ statement, *READ x, y, group(i), nam$(i),*

will read the first DATA statement, *DATA 10, 10, 1, 110 bits per sec,* and make the following variable assignments:

$x = 10$

$y = 10$

group(1) = 1

nam$(1) = "110 bits per sec"

The READ statement is followed by the BUTTON statement, which uses three of the newly assigned variables to position the first button. You'll recall that the syntax for the BUTTON statement is:

BUTTON *ID, state*[, *title, rectangle*[, *type*]]

The BUTTON statement in the Terminal program reads:

```
BUTTON i, 1, nam$(i), (x, y) - (x + 135, y + 15), 3
```

so on the first time through the loop, it will look like this:

BUTTON 1, 1, "110 bits per sec", (10, 10) - (145, 25), 3

The values of x and y, brought in from the DATA statement, are used to define the upper left corner of the rectangle that determines the button's location. The x value is then increased by 135 and the y value by 15 to define the lower right corner.

After 16 times through the loop, there will be 16 active radio-type buttons. The line following the FOR...NEXT loop then creates one additional button, titled OK, for the user to click to institute the new communication parameters.

The second FOR...NEXT loop in this section calls the *SelectButton* subprogram four times to change each of the buttons that were stored in the *choice* array by the *Init* subroutine to the default state.

Following the *DisplayDefaults* subroutine are the DATA statements it reads (Figure 11-10). You could move this section to any other place in the program, or divide it into parts and scatter them throughout the program. As long as the statements appeared in the same order, moving them would not affect how the program runs.

```
REM --- x,y coordinate of button, groupid, title
  DATA 10, 10, 1, 110 bits per sec
  DATA 10, 30, 1, 300 bits per sec
  DATA 10, 50, 1, 600 bits per sec
  DATA 10, 70, 1, 1200 bits per sec
  DATA 10, 90, 1, 2400 bits per sec
  DATA 10, 110, 1, 4800 bits per sec
  DATA 10, 130, 1, 9600 bits per sec

  DATA 290, 10, 2,No parity
  DATA 290, 30, 2,Even parity
  DATA 290, 50, 2,Odd parity

  DATA 150, 10, 3, 5 Data bits
  DATA 150, 30, 3, 6 Data bits
  DATA 150, 50, 3, 7 Data bits
  DATA 150, 70, 3, 8 Data bits

  DATA 150, 110, 4, 1 Stop bits
  DATA 150, 130, 4, 2 Stop bits
```

Figure 11-10. The DATA statements

The individual items in a DATA statement are separated from each other by commas. There is no need to place quotation marks around an individual string unless the string itself contains commas, colons, or leading or trailing spaces that you want to preserve.

Managing the parameter options

Having returned from *DisplayDefaults*, *HandleMenu* branches to *SelectOptions* (Figure 11-11 on the next page), which waits for a button to be clicked. If a button with an ID of less than 17 is clicked, the number of the button is passed to the *SelectButton* subprogram, which changes the status of the selected button and stores the selection in the *choice* array. If the OK button (#17) is clicked, indicating that the user has finished changing parameters, then program flow returns to *HandleMenu*, where the options are set and the communication port reopened.

```
SelectOptions:
SelectLoop:
  dialogId = DIALOG(0)
  IF dialogId <> 1 THEN SelectLoop
  buttonId = DIALOG(1)
  IF buttonId < 17 THEN CALL SelectButton(buttonId) : GOTO SelectLoop
  RETURN
```

Figure 11-11. Managing the option selections

We discussed the DIALOG function when we first encountered it in Chapter 6. Then, as now, we used only two of the seven possible variations: DIALOG(0) and DIALOG(1). Recall that DIALOG(0) returns a 1 if a button is clicked (otherwise it returns a 0) and that DIALOG(1) returns the ID of the most recently clicked button. Since a function does not store the value returned, you must assign it to a variable in order to use the value more than once. At the beginning of this section DIALOG(0) is assigned to *dialogId,* and DIALOG(1) is assigned to *buttonId.* Once the program moves to the *SelectOptions* subroutine, it goes into a loop and waits for *dialogId* to become equal to 1, indicating that a button has been clicked in the window. As soon as a button is clicked, DIALOG(1), which has been set equal to *buttonId,* returns the button number, which will be used to set the new communication parameters.

Managing the buttons

The management of the parameter buttons is handled by the *SelectButton* subprogram. As was pointed out in Chapter 8, a subprogram is similar to a subroutine, except that the subprogram has its own set of variables, which remain unique to it unless you specifically instruct it to share some of the variables used in the main program.

SelectButton, shown in Figure 11-12, is called each time a button other than the OK button is pressed in the configuration dialog box. It determines which of the parameter buttons was pressed, removes the highlight from the previously selected button, and highlights the current selection. It then assigns the selected parameter to the array variable *choice$(groupId),* where *groupId* identifies the selected parameter's group (baud, parity, data bits, or stop bits). Since the array *choice$* is a shared variable, the new value will also be available to the main program.

```
REM --- The user has just pushed a button.  Highlight
REM --- that button and remember the selection in choice()
REM --- and choice$()
REM ---
SUB SelectButton(buttonId) STATIC
   SHARED nam$(), group(), choice(), choice$()

   groupId = group(buttonId)
   IF choice(groupId) > 0 THEN BUTTON choice(groupId), 1
   BUTTON buttonId, 2
   choice(groupId) = buttonId
   IF groupId = 2 THEN setParity
   choice$(groupId) = STR$(VAL(nam$(buttonId)))
   EXIT SUB
setParity:
   choice$(groupId) = LEFT$(nam$(buttonId), 1)
END SUB
```

Figure 11-12. Managing the parameter buttons

This subprogram is a little more complex than the subroutines we've looked at, so let's examine it one line at a time. The first line simply identifies the start of the subprogram and lists the formal parameters and storage class.

```
SUB SelectButton(buttonId) STATIC
```

The SHARED statement then identifies which of the variables are common to both the main progam and the subprogram.

```
SHARED nam$(), group(), choice(), choice$()
```

The next line assigns the group number of the button selected to the variable *groupId.* Baud-rate options (buttons 1 through 7) are group 1; parity options (buttons 8 through 10) are group 2; data bit options (buttons 11 through 14) are group 3; and stop bit options (buttons 15 and 16) are group 4.

```
groupId = group(buttonId)
```

Next, an IF...THEN statement checks the value of the shared array variable *choice* to confirm that a button was previously selected for this group, and then deselects it (removes the highlight by changing the button *status* to 1).

```
IF choice(groupId) > 0 THEN BUTTON choice(groupId), 1
```

The following two lines highlight the button that was just pressed and make it the choice for its group.

```
BUTTON buttonId, 2
choice(groupId) = buttonId
```

If the button is in group 2 (parity), an IF...THEN statement diverts program flow to the line labeled *SetParity* for reasons that will become clear in a moment. (It's worth pausing here to remind you that although variable names are unique within a subprogram, line labels are not. If you use a label within the subprogram that is also used in the main program, the program stops and the error message "Duplicate label" is displayed.) If the button is not in group 2, the program continues with this line:

```
choice$(groupId) = STR$(VAL(nam$(buttonId)))
```

The VAL(X$) function is typically used to convert a number that has been stored as a string variable back to a numeric variable, so that it can be manipulated mathematically. A secondary effect of this function is useful in this program: In performing the conversion, all leading blanks, tabs, and linefeeds are stripped from the argument (X$), and then, working from left to right, each remaining character is converted to a numeric value until a character is encountered that is not a number, at which time the

function terminates. The result of the VAL function is thus to convert a string such as *110 bits per sec,* which was stored in the *nam$* array by the *DisplayDefaults* subroutine earlier in the program, to the number *110.*

The STR$(X) function does just the opposite, converting numbers to strings. The result of the combination of these two functions is to pull the leading number from each of the strings that store the names of the parity, data-bit and stop-bit buttons (groups 1, 3, and 4) and convert it back to a string. This retrieved string is then assigned to the variable *choice$(groupId),* which will later be used as a parameter when opening the communication port.

If a parity button is chosen (group 2), the portion of the parameter that has to be assigned to *choice$(groupId)* is the first letter of the button name. Since the name consists entirely of letters—No parity, Even parity, Odd parity—the VAL function is of no use. However, we can use the LEFT$ function, which has the syntax:

LEFT$(X$, I)

to return the leftmost *I* characters of *X$.* In this case, it returns just the leftmost character of *nam$(buttonId)*—N, E, or O—which is then assigned to *choice$(groupId).*

```
choice$(groupId) = LEFT$(nam$(buttonId), 1)
```

If the button that was pressed to cause this subprogram to be called was in group 1, 3, or 4, the subprogram is terminated by the conditional EXIT SUB statement. If the button was in group 2, the subprogram is terminated by the END SUB statement. Either way, program flow returns to the statement following the CALL statement in the *SelectOptions* subroutine: *GOTO SelectLoop.* Flow returns from *SelectOptions* to *HandleMenu* when the OK button (#17) is clicked.

That completes our discussion of the Terminal program. If your Macintosh is hooked to a modem or to another computer, you may have used this program to send the characters you type and read the response, but you probably found that its usefulness is limited by its inability to store the information received. However, now that you understand how the program works, you can see that it would not be too difficult to open an extra file and, each time a character from the modem port is printed

on the screen, to also print it in the file. This could be accomplished by adding just a few lines to the program, but such a simple refinement wouldn't give me a chance to explain many new BASIC commands. Instead, I'll use the next chapter to add some extra bells and whistles to create a more useful communication program.

```
REM --- Terminal
REM --- Terminal Emulation Program

   DIM nam$(17), group(17), choice(4), choice$(4)
   GOSUB Init

'Display characters from COM1, send keystrokes to COM1
Loop:
   PRINT INPUT$(LOC(1), 1);
   IF LOC(1) > 0 THEN Loop
   k$ = INKEY$
   IF k$ = "" THEN Loop
   PRINT #1, k$;
   GOTO Loop

Init:
   TEXTFONT 4                                        'mono-spaced font
   TEXTSIZE 9                                  'allows 80 characters per line
   TEXTMODE 1                                  'print mode = XOR, not COPY

   REM --- Setup menu
   MENU 1, 0, 1, "File"
   MENU 1, 1, 1, "Quit"
   MENU 2, 0, 1, "Configure"
   MENU 2, 1, 1, "Set configuration parameters"
   MENU 3, 0, 0, ""
   MENU 4, 0, 0, ""
   MENU 5, 0, 0, ""
   ON MENU GOSUB HandleMenu
   MENU ON

   REM --- Setup default options
   choice(1) = 2                                          '300 baud
   choice(2) = 9                                          'even parity
```

more...

Figure 11-13. The complete Terminal program

```
    choice(3) = 13                                                    '7 data bits
    choice(4) = 15                                                    '1 stop bit
    REM --- Open Communications port with 2000 byte input buffer
    OPEN "COM1: 300, e, 7, 1" AS 1 LEN = 2000
    RETURN

HandleMenu:
    IF MENU(0) = 1 THEN SYSTEM                                        'got quit command
    CLOSE 1                                    'else it must be Set configuration parameters
    WINDOW 2,, (50, 50) - (450, 250), 2
    GOSUB DisplayDefaults
    GOSUB SelectOptions
    WINDOW CLOSE 2
    options$ = choice$(1) + "," + choice$(2) + "," + choice$(3) + "," + choice$(4)
    REM --- Open Communications port with 2000 byte input buffer
    OPEN "COM1: " + options$ AS 1 LEN = 2000
    CLS
    MENU 1, 0, 1
    RETURN

DisplayDefaults:
    REM *** Prompt user for Communications Parameters
    RESTORE
    FOR i = 1 TO 16
        READ x, y, group(i), nam$(i)
        BUTTON i, 1, nam$(i), (x, y) - (x + 135, y + 15), 3
        NEXT i
    BUTTON 17, 1, "Ok", (310, 110) - (350, 150)
    REM *** Simulate button pushes to highlight defaults
    FOR i=1 TO 4
        SelectButton choice(i)
        NEXT i
    RETURN

REM --- x,y coordinate of button, groupid, title
    DATA 10, 10, 1, 110 bits per sec
    DATA 10, 30, 1, 300 bits per sec
    DATA 10, 50, 1, 600 bits per sec
    DATA 10, 70, 1, 1200 bits per sec
    DATA 10, 90, 1, 2400 bits per sec
    DATA 10, 110, 1, 4800 bits per sec
    DATA 10, 130, 1, 9600 bits per sec
```

Figure 11-13. The complete Terminal program *(continued)*

more...

```
    DATA 290, 10, 2,No parity
    DATA 290, 30, 2,Even parity
    DATA 290, 50, 2,Odd parity

    DATA 150, 10, 3, 5 Data bits
    DATA 150, 30, 3, 6 Data bits
    DATA 150, 50, 3, 7 Data bits
    DATA 150, 70, 3, 8 Data bits

    DATA 150, 110, 4, 1 Stop bits
    DATA 150, 130, 4, 2 Stop bits

SelectOptions:
SelectLoop:
    dialogId = DIALOG(0)
    IF dialogId <> 1 THEN SelectLoop
    buttonId = DIALOG(1)
    IF buttonId < 17 THEN CALL SelectButton(buttonId) : GOTO SelectLoop
    RETURN

REM --- The user has just pushed a button.  Highlight
REM --- that button and remember the selection in choice()
REM --- and choice$()
REM ---
SUB SelectButton(buttonId) STATIC
    SHARED nam$(), group(), choice(), choice$()

    groupId = group(buttonId)
    IF choice(groupId) > 0 THEN BUTTON choice(groupId), 1
    BUTTON buttonId, 2
    choice(groupId) = buttonId
    IF groupId = 2 THEN setParity
    choice$(groupId) = STR$(VAL(nam$(buttonId)))
    EXIT SUB
setParity:
    choice$(groupId) = LEFT$(nam$(buttonId), 1)
END SUB
```

Figure 11-13. The complete Terminal program *(continued)*

The Expanded Communication Program

Chapter 12

The terminal emulation program supplied on your BASIC disk and explained in Chapter 11 is an excellent learning tool, and in certain circumstances has practical value. But it lacks many of the features considered standard in a communication program. In this chapter we will expand the terminal program to include the features I consider desirable. By the time we are through, you should understand the program well enough to tailor it precisely to your needs.

Before we get involved with enhancing the Terminal program, let's prove how simple communication is by stripping Terminal down to its bare essentials. Figure 12-1 shows a shorter version of this program that still manages to move information between computers. As you can see, it is really just the program we have been working with, without the menus and choices of communication parameters. With only very minor modifications, this program will run on almost any computer that runs some form of BASIC.

```
Init:
  OPEN "COM1:300, e, 7, 1" AS 1 LEN = 2000
Loop:
  PRINT INPUT$(LOC(1), 1)
  IF LOC(1) > 0 THEN Loop
  k$ = INKEY$
  IF k$ = "" THEN Loop
  PRINT #1, k$;
  GOTO Loop
```

Figure 12-1. A stripped-down communication program

You could consider this the core of the communication program we are about to develop. The program we worked with in Chapter 11 is the second stage: It added a few visual refinements and niceties. Now we will carry the development a little further by adding routines that give the user the ability to toggle on and off the saving of information that passes through the Macintosh's port and across its screen. We will also add the ability to upload and download files, and to store and edit a directory of telephone numbers and dial one of these numbers with a single command.

The flow chart in Figure 12-2 shows the major sections of the program we are going to build. Can you recognize the Terminal program buried in it? In designing this program, I analyzed the features of the Terminal program, made a list of features I wanted to add, and sketched a chart similar to this one. Since this program is quite a bit more complex than Terminal (not necessarily more difficult—just more of it), I added a "flow control" section consisting of a series of GOSUB statements that route the program through initialization subroutines that define key variables, create the screen display, set default communication parameters, open the communication port, and create the menu, before allowing it to flow into the main communication loop. Since each of the initialization subroutines is used only once, the program could simply be allowed to flow through them in a linear fashion, without all the GOSUBs and RETURNs, but this section is helpful to someone encountering the program for the first time, as it gives them a quick feeling for the flow and the significant sections.

In this chapter, I'm basically going to re-enact the process I went through to create the program you will find in Figure 12-63 (without all the mistakes and dead ends involved in the original task). If you are entering this program as you follow along, there is no point in including the comments, as they slow the program down appreciably. As a matter of fact, if you have a Macintosh with 128K of RAM, the comments would make this program too large to load.

After I wrote the flow-control GOSUB statements, I labeled each subroutine and stubbed it out with a RETURN statement, as shown in Figure 12-3 on the page after next. When the program returned from all the subroutines, I had it fall into the main communication loop, labeled *CommLoop:*, where I placed an infinite WHILE... WEND loop to hold it. Then I began filling in individual modules.

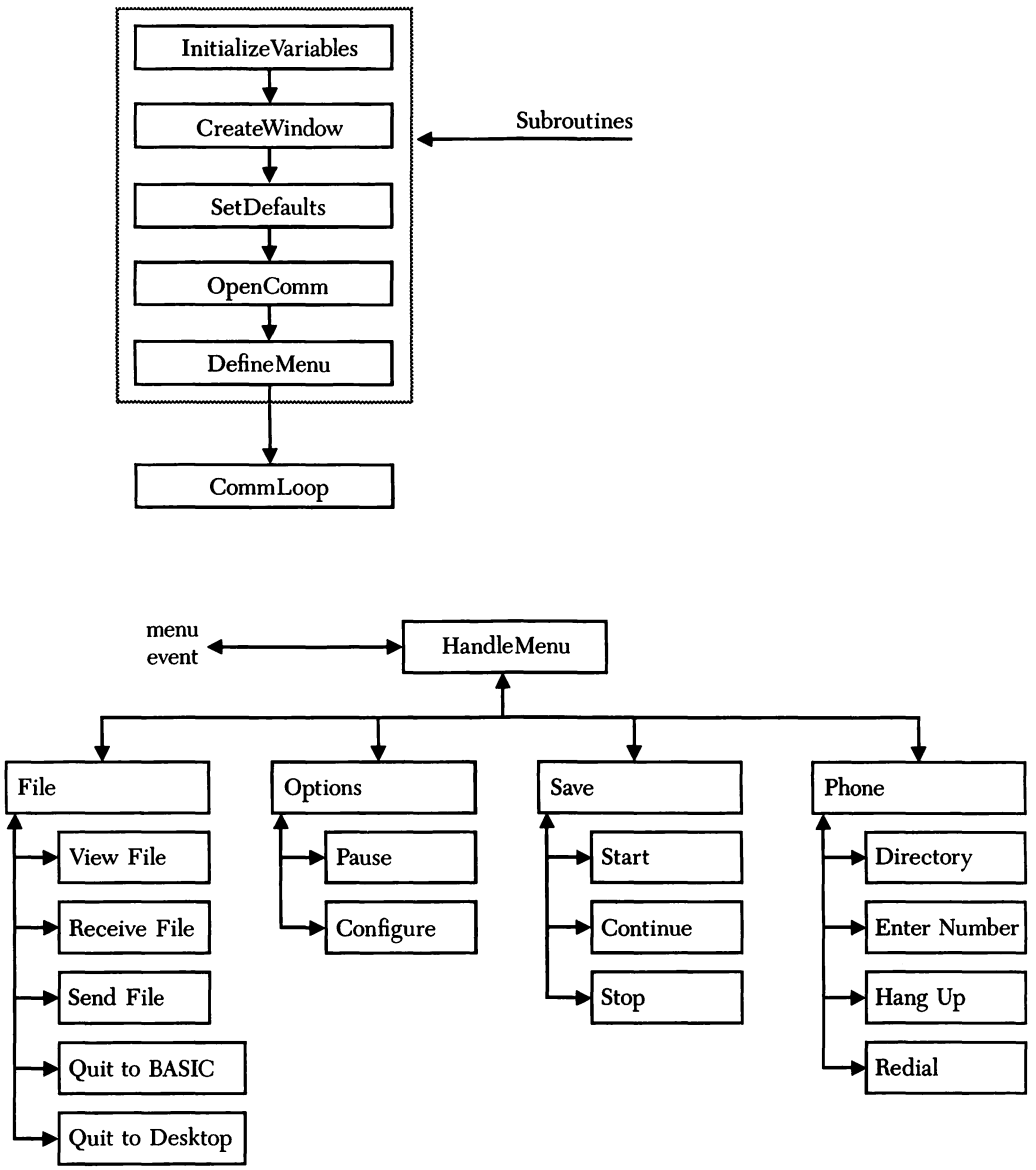


Figure 12-2. A flow chart of the enhanced communication program

```
'*
'* Execute all initializing subroutines.
'*
   GOSUB InitializeVariables
   GOSUB CreateWindow
   GOSUB SetDefaults
   GOSUB OpenComm
   GOSUB DefineMenu

CommLoop:
   WHILE 1 = 1
   WEND

InitializeVariables:
   RETURN

CreateWindow:
   RETURN

SetDefaults:
   RETURN

OpenComm:
   RETURN

DefineMenu:
   RETURN
```

Figure 12-3. The framework

Initializing the variables

The first section to add is the one that initializes the variables and dimensions the arrays. It is important to place this section near the beginning of the program, before any arrays are actually used.

As you can see in Figure 12-4, I have added only three arrays to those used in the Terminal program, but one of them is very large. The two small arrays, *num$(10)* and *who$(10),* are used in the subprogram that creates the telephone directory. The large array, *scrnsave(4889),* is used to refresh portions of the screen that are covered, then uncovered, as we display various dialog boxes.

```
'*
'* Dimension arrays and initialize variables.
'*
InitializeVariables:
   DEFINT a - z                                                    'for speed
   DIM choice(4)                                                   'selected parameters
   DIM choice$(4)                                                  'likewise
   DIM group(17)                                                   'parameter buttons
   DIM nam$(17)                                                    'likewise
   DIM num$(10)                                                    'directory phone number
   DIM scrnsave(4889)                                              'used to restore screen
   DIM who$(10)                                                    'directory name

   alert$ = CHR$(7)                                                'beep
   buffer = 2000                                                   'input buffer
   bufferLimit = .9 * buffer                                       'buffer limit--send XOFF
   choice(1) = 4                                                   '1200 baud
   choice(2) = 9                                                   'even parity
   choice(3) = 13                                                  '7 data bits
   choice(4) = 16                                                  '2 stop bits
   cr$ = CHR$(13)                                                  'carriage return
   dial$ = "ATDT"                                                  'D.C. Hayes dial command
   directFile$ = "directory file"                                  'file containing phone #'s
   endMessage$ = sp$ + cr$ + lf$ + alert$ + "End of Transmission" + cr$
   false = 0
   hangup$ = "~+++~ATH0"                                           'hangup command
   lf$ = CHR$(10)                                                  'linefeed
   pauseFlag = false                                               'activity paused
   posit$ = cr$ + "   "
   receiveFlag = false                                             'Receive File selected
   saveFlag = false                                                'save flag
   sendFlag = false                                                'Send File selected
   sp$ = CHR$(32)
   stopFlag = false                                                'XOFF sent flag
   true = -1
   viewFlag = false                                                'View file selected
   waitFlag = false                                                'XOFF received flag
   xoff$ = CHR$(19)                                                'stop sending
   xon$ = CHR$(17)                                                 'send more
   RETURN
```

Figure 12-4. Initializing the variables

I also chose to define quite a few variables in this section, and to specify that all variables will be integers unless individually defined otherwise. These variables were not the result of a tremendous amount of forethought. In fact, they weren't even here before I started fleshing out the program, but were added as I needed them. Since I now know what's needed here, we may as well list them all, and discuss each of them as they are used.

Setting aside a specific area of a program in which significant variables are defined as you add them can be useful in three ways: First, you then have one place where you can look up the values assigned to these variables. Second, if you include adequate remarks here, you will be able, a few months later, to look up variables with names you thought you'd never forget the meaning of to see what they mean. And third, if variables are subject to change between versions of the program, you can make one change here rather than searching and replacing throughout the program.

An example of this last point is the assignment of *"ATDT"* to *dial$*. *ATDT* is the D.C. Hayes command to dial a touch-tone telephone. By using *dial$* throughout the program whenever I want to dial the phone, I need only replace this one definition to change to a different modem or to a rotary phone.

Testing the module

There is really no operational test for this section, since it does nothing obvious. You can, however, practice a troubleshooting technique that may be handy in the future. Run the program, and then stop it and make the Command window active. Use the PRINT statement to display the values of the variables that you defined in *InitializeVariables*. To prove that the program is setting these variables, change one of them from the Command window, as with this command:

```
LET endMessage$ = "Goodbye"
```

Now print your new value, and then run the program and print it again.

Creating the window

The addition of an output window seems straightforward: You simply want something that will fill the available space and provide a background for text. It *seems* straightforward, but I experienced a little frustration before I settled on the window

created by the listing in Figure 12-5. The cause of my frustration was a desire to have the maximum amount of space possible for displaying text. The original window went from (0,20) to (512,342), and when I reached the point where the program was capable of displaying text, I found that it scrolled rather slowly and flickered a lot. It turns out that the contortions the Macintosh goes through to scroll a window that does not have square corners (as when the corners are cut off by the rounded corners of the screen) are considerably more involved than when the corners are square. A less obtrusive problem involved a narrow strip along the left edge of the window that disappeared after certain operations. This was just a pixel or two of the first character of each line, but it was noticeable. Moving the window in three pixels and up four solved both problems. You may want to enlarge the window later in the program, just to see what I was dealing with.

I chose a type 3 window, which does not have a title bar, so there is no point in including a *title* parameter in the WINDOW statement that begins this subroutine, but notice the two commas used to hold the space.

Next, PENMODE sets the graphic-call mode to XOR, which reverses the color of the pixels where a new graphic appears. The only graphic call in the program is the one to create the cursor: The XOR mode allows the cursor to be erased by drawing another cursor on top of it. The PENMODE, TEXTFONT, and TEXTSIZE ROM calls were placed in this section rather than in the *InitializeVariables* subroutine because they apply only to the current window, and you therefore have to wait until after the window is opened to call them.

```
'*
'* Create output window.
'*
CreateWindow:
  WINDOW 1, , (3, 20) - (511, 338), 3
  PENMODE 10
  TEXTFONT 4                                    'monospace font
  TEXTSIZE 9                                    '80 characters per line
  RETURN
```

Figure 12-5. Creating the output window

You can run the program again to test for typographical errors, but there still isn't much to look at. About all you can experiment with at this point is the window type, size, or title.

Setting the defaults

This section of the program uses the values you assigned to *choice(1)* through *choice(4)* in the *InitializeVariables* subroutine to actually set the default options used when the communication port is initially opened. This is a slight improvement on the Terminal program, which assigned the default values but used them only to simulate the button pushes in the configuration dialog box, thus requiring you to edit both the *choice* array and the statement that opened the port in order to change the default setting in the program. The method used in the *SetDefaults* subroutine (Figure 12-6) allows you to open COM1: with new parameters simply by changing values assigned to *choice(1)* through *choice(4)* in the *InitializeVariables* subroutine.

The FOR...NEXT loop in this subroutine reads through the 16 sets of DATA statements located a little further into the program and discards everything except 16 values that are assigned to the array variable *nam$* (the unwanted values are assigned to the variable *garbage*, which is never used). The four values assigned to the *choice* array in *InitializeVariables* are used to extract the default communication parameters (currently 1200 baud rate, even parity, 7 data bits, and 2 stop bits), which are then concatenated into *option$*.

The method used to extract the default parameters is the same as that used in the *SelectButton* subprogram we discussed in Chapter 11. You'll recall that each element of the *nam$* array holds more information than we need (for example, *110 bits per sec*), so *choice$(1)*, *choice$(3)*, and *choice$(4)* are extracted by using the VAL function to return the numerical value of *nam$* and then the STR$ function to convert this new value back to a string, while *choice$(2)* is extracted by using the LEFT$ function to return the leftmost character of *nam$*.

If you would like to test this section of the program now, you could add a PRINT statement after the concatenation of *option$*, to display the options on the screen. Then, after running the program, you could edit the values assigned to the *choice* array in the *InitializeVariables* subroutine and run the program again. This time *option$* should indicate the new defaults. (Don't forget to delete the PRINT statement after completing the test.)

```
'*
'* Set up default options.
'*
'* choice(1) is baud rate: 1  --  110
'*                         2  --  300
'*                         3  --  600
'*                         4  -- 1200
'*                         5  -- 2400
'*                         6  -- 4800
'*                         7  -- 9600
'*
'* choice(2) is parity:    8  -- None
'*                         9  -- Even
'*                        10  -- Odd
'*
'* choice(3) is data bits: 11 -- 5
'*                         12 -- 6
'*                         13 -- 7
'*                         14 -- 8

'* choice(4) is stop bits: 15 -- 1
'*                         16 -- 2
'*


'*
'* Changing default choices here will control initial parameters.
'*
SetDefaults:
   FOR count = 1 TO 16
      READ garbage, garbage, garbage, nam$(count)                    'discard garbage
   NEXT
   choice$(1) = STR$(VAL(nam$(choice(1))))
   choice$(2) = LEFT$(nam$(choice(2)), 1)
   choice$(3) = STR$(VAL(nam$(choice(3))))
   choice$(4) = STR$(VAL(nam$(choice(4))))
   options$ = choice$(1) + "," + choice$(2) + "," + choice$(3) + "," + choice$(4)
   RETURN
```

Figure 12-6. Setting the defaults

Opening the port

The *OpenComm* subroutine (Figure 12-7) opens the communication port with our default parameters. I decided to add the *Flush* subroutine, which simply grabs any garbage that might be in the input buffer and discards it, to ensure that the first characters printed on the screen will be those actually received through the port.

Setting up the menus

This section of the program should be a little more satisfying than the preceding ones, since it does something you can actually see. I discussed the three MENU statements used here while explaining the Terminal program. When you finish typing *DefineMenu* (Figure 12-8), your program will have four new menus (the *MENU 5, 0, 0,""* statement simply creates a blank menu to "erase" the fifth menu from BASIC's menu bar) with a total of 14 items from which to select. The menu items with a *status* parameter of 0 (Continue, Stop, and Redial) are initially disabled, since it is not reasonable to select them before something else happens. For example, there is no point in enabling Redial until after the user has dialed a number.

Now you need to stub out the *HandleMenu* subroutine with a RETURN, just as you did for each of the initialization subroutines, so that you can run the program and look at all your new menus. It is a little disappointing to have nothing happen when you select from a menu, so let's try another troubleshooting technique. Go back to the

```
'*
'* Open communications port.
'*
OpenComm:
   OPEN "COM1:" + options$ AS 1 LEN = buffer

'*
'* Clear input buffer.
'*
Flush:
   garbage$ = INPUT$ (LOC(1), 1)
   RETURN
```

Figure 12-7. Opening the communication port

```
'*
'* Set up custom menu.
'*
DefineMenu:
   MENU 1, 0, 1, "File"
   MENU 1, 1, 1, "View File"
   MENU 1, 2, 1, "Receive File"
   MENU 1, 3, 1, "Send File"
   MENU 1, 4, 1, "Quit to BASIC"
   MENU 1, 5, 1, "Quit to Desktop"
   MENU 2, 0, 1, "Options"
   MENU 2, 1, 1, "Pause"
   MENU 2, 3, 1, "Set Configuration Parameters"
   MENU 3, 0, 1, "Save"
   MENU 3, 1, 1, "Start"
   MENU 3, 2, 0, "Continue"
   MENU 3, 3, 0, "Stop"
   MENU 4, 0, 1, "Phone"
   MENU 4, 1, 1, "Directory"
   MENU 4, 2, 1, "Enter Number"
   MENU 4, 3, 1, "Hang Up"
   MENU 4, 4, 0, "Redial"
   MENU 5, 0, 0, ""
   ON MENU GOSUB HandleMenu                    'if menu item chosen
   MENU ON                                     'activate trapping
   RETURN

'*
'* Decide which menu item selected and take action.
'*
HandleMenu:
   RETURN
```

Figure 12-8. Setting up the menus

stubbed-out *HandleMenu* subroutine and insert a statement that does something—I usually use the BEEP statement, which sounds a short tone. You could also type:

```
PRINT alert$
```

which uses the variable *alert$* that we defined to be CHR$(7)—the beep character—in the *InitializeVariables* subroutine. Now when you run your program, you will hear a beep each time you select an item from the menu.

When you have finished testing this part of the program, you will have to hold down the Command key while pressing the period key to quit to BASIC, since we have replaced the BASIC menu and have not yet activated Quit on our new menu. When the Command window appears on your screen, type *MENU RESET* to restore the BASIC

```
'*
'* Display characters from COM1:, send keystrokes to COM1:.
'*
CommLoop:
  WHILE true
    WHILE pauseFlag : WEND
    IF (LOC(1) = 0) AND stopFlag THEN PRINT #1, xon$; : stopFlag = false
    WHILE LOC(1) = 0                                    'nothing waiting to come in
      SendKey                                           'send key typed to file #1
      MENU STOP                                         'don't get sidetracked
      IF (sendFlag OR viewFlag) AND NOT waitFlag THEN CALL SendLine
      IF endViewFlag THEN GOSUB EndFile
      MENU ON                                           'see if anything selected
    WEND
    IF LOC(1) > bufferLimit THEN PRINT #1, xoff$; : stopFlag = true
    lin$ = INPUT$(LOC(1), 1)                            'get everything waiting
    RemoveChars lin$, lf$                               'strip linefeeds
    RemoveChars lin$, xon$
    IF sendFlag AND waitFlag AND removeFlag THEN waitFlag = false
    RemoveChars lin$, xoff$
    IF sendFlag AND removeFlag THEN waitFlag = true
    MENU STOP
    PrintString lin$
    MENU ON
    IF endSendFlag THEN GOSUB EndFile

    '*
    '* File #3 is automatically named file that stores input when
    '* Start or Continue is chosen from Save menu. File #2
    '* stores received file after asking for name to store it under.
    '*
    IF saveFlag THEN PRINT #3, lin$;
    IF receiveFlag THEN PRINT #2, lin$;
  WEND
```

Figure 12-9. The main communication loop

menu and choose Show List from the Windows menu to redisplay the program in the List window.

The main communication loop

The main communication loop is the heart of this program, just as it was in the Terminal program. Although the final version of *CommLoop,* shown in Figure 12-9, is the result of a lot of refinement that went on as sections were added to the program, the highlighted statements that make up its core are very similar to those in the Terminal program.

The logic in this section is a little difficult to follow if you don't understand what is going on in the rest of the program. I use a set of variables ending in *Flag*—*pauseFlag, stopFlag, viewFlag,* and so on—to indicate that a selection has been made elsewhere in the program that affects how *CommLoop* should deal with the text it processes. To avoid having to explain all these functions before their time, let's start with just the core loop shown in Figure 12-10.

This listing is made up of the highlighted statements from the previous listing, except for the *SendKey* and *PrintString* subprograms, which have been replaced by simple statements. As soon as you are certain this core loop works, we will add the subprograms.

You can test the communication loop by hooking your modem to the Macintosh, running the program, and typing something. As with the Terminal program, what you

```
'*
'* Initial CommLoop
'*
   WHILE true
      WHILE LOC(1) = 0
         keyTyped$ = INKEY$
         IF keyTyped$ <> "" THEN PRINT #1, keyTyped$
      WEND
      lin$ = INPUT$(LOC(1), 1)
      PRINT lin$
   WEND
```

Figure 12-10. The core communication loop

type can get to your screen only by going out the communication port and being reflected back by either a modem or another computer. If you do not have a modem or another computer to hook to, and still feel compelled to type and test this program, you can simulate a modem by placing a jumper between the transmitted-data and received-data pins on the communication-port connector. Since this is a female connector, a lightweight paper clip straightened out and then rebent to fit between sockets 5 and 9, as shown in Figure 12-11, will do the trick.

CAUTION: If you try this, be careful not to force too large a paper clip into the hole, and BE EVEN MORE CAREFUL NOT TO GO INTO THE WRONG HOLES. Pin 2 is tied to the Mac's +5-volt power supply, and pin 6 to the +12-volt power supply. Shorting either of these to ground, which is on neighboring pins 1 and 3, will add a little unexpected excitement to your life.

When you've successfully tested the loop, go on to the next section, where we will replace the *PRINT lin$* statement with the *PrintString* subprogram, to add a cursor to the screen at the next print location.

Adding a cursor

Adding and managing the cursor requires a couple of subprograms and the use of two new functions—WINDOW(4) and WINDOW(5). WINDOW(4) returns the x coordinate of the location in the current window where the next character will be drawn, and WINDOW(5) returns the y coordinate. Since this is where you want the cursor to be drawn, this will be a handy set of functions.

In the next few minutes we will create four subprograms, so this seems like a good time to discuss how we want to store all of the subprograms in this application. I like to arrange them in alphabetical order after the main program, as you will see in the full listing in Figure 12-63, but there are undoubtedly other methods, and you are

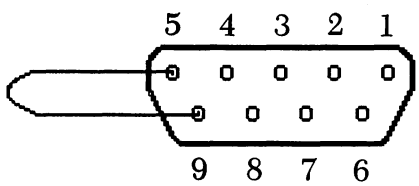


Figure 12-11. A simple dummy plug

welcome to choose the one that suits you. However you arrange them, I think it is important that they be in some logical order, so that people reading your program can locate them easily.

The first subprogram we will discuss, *ShowCur*, is listed in Figure 12-12. This subprogram first assigns the x and y coordinates of the current pen location to the variables *curX* and *curY*. It next uses the LINE ROM call to draw a line (the cursor) from the current pen location to a point five pixels to the right and on the same level (5,0), and then returns the pen to its previous position, at the front of the cursor.

You can see the result of this subprogram by returning to *CommLoop* and calling *ShowCur* right after the *PRINT lin$* statement. When you run the program, a cursor appears after each character you type. There are a few minor cosmetic problems, however: For one thing, the first line of text is printed just above the output window, so all that shows is the cursor, which hangs a little below the line. Then you'll notice that pressing the Return key leaves a cursor at the end of each line. This is because each character you type overwrites the previous cursor (due to the default text mode of 0), but no character is printed for a carriage return. Also, there is no initial cursor on the blank screen, since this subprogram is not called until something is printed.

The solutions to these problems are not particularly difficult. Let's start by getting the first line down where we can see it. It's out of sight because when a window is created or cleared, the pen is automatically moved to location (0,0). As you will recall, ROM calls like LINE are drawn below the pen location, and BASIC statements like PRINT are displayed above the pen location. So the text is out of sight, but we can see

```
'*
'* Show cursor at end of current line.
'*
SUB ShowCur STATIC
  SHARED curX, curY
  curX = WINDOW(4)                    'horizontal location of next character
  curY = WINDOW(5)                    'vertical location
  CALL LINE (5, 0)                    'draw cursor
  MOVETO curX, curY                   'put print location back where it was
END SUB
```

Figure 12-12. The *ShowCur* subprogram

```
'*
'* Place cursor in upper left corner.
'*
SUB RestoreCur STATIC
   MOVETO 0, 10
   ShowCur
END SUB
```

Figure 12-13. The *RestoreCur* subprogram

the cursor. We have to move the pen location down a bit before we call *ShowCur* the first time after creating or clearing the window. Let's use another subprogram to do this: We'll call this one *RestoreCur.* It consists of the few lines shown in Figure 12-13.

Make sense? Now return to the *CreateWindow* subroutine and add a call to this subprogram between the TEXTSIZE call and the RETURN statement. This is the only addition we'll make to *CreateWindow,* which should now look like Figure 12-14. If you run the program again, the first line of text will now be visible.

Now we'll get rid of the extra cursor at the end of each line. We can do this by erasing the cursor just before each new line is printed. This means erasing the cursor many times when it isn't really necessary, as the text would overwrite it anyway, but testing each character to see if it is a carriage return, and erasing the cursor only if it is, slows the program substantially.

```
'*
'* Create output window.
'*
CreateWindow:
   WINDOW 1, , (3, 20) - (511, 338), 3
   PENMODE 10
   TEXTFONT 4                                   'monospace font
   TEXTSIZE 9                                   '80 characters per line
   RestoreCur
   RETURN
```

Figure 12-14. The final *CreateWindow* subroutine

```
'*
'* Print string of text.
'*
SUB PrintString(text$) STATIC
   SHARED curX, curY
   CALL LINE (5, 0)                                    'erase cursor
   MOVETO curX, curY                                    'move back
   PRINT text$;
   ShowCur                                             'show new cursor
END SUB
```

Figure 12-15. The *PrintString* subprogram

Now is the time to replace the *PRINT lin$* statement (and the *ShowCur* call) in *CommLoop* with *PrintString lin$*, a new call to the *PrintString* subprogram shown in Figure 12-15.

You can see immediately that this subprogram will print the *text$* characters passed to it, and then call *ShowCur*, but do you understand why it causes the cursor to disappear? The last time *ShowCur* was called, it left the pen at the beginning of the cursor. So when we call *LINE (5,0)* again, we are drawing a second cursor on top of the first one. Since we specified *PENMODE 10* in the *CreateWindow* subroutine, the new pixels drawn are XORed with the old, reversing their color to white.

Managing the menus

Now that we have the cursor under control, let's flesh out the File menu, so that you can at least quit in a civilized manner. First, let's replace the BEEP statement in the *HandleMenu* subroutine with the subroutine in Figure 12-16 on the next page, which controls where the program branches when an item is chosen from the menu.

The MENU(0) and MENU(1) functions, two new variations of the MENU command, return the number of the selected menu and the number of the item chosen from that menu, which are assigned here to the variables *MenuSel* and *ItemSel*, respectively. *HandleMenu* then uses a computed GOSUB statement to branch to the first (*FileMenu*), second (*OptionMenu*), third (*SaveMenu*), or fourth (*PhoneMenu*) subroutine listed, depending upon the value (1 through 4) returned by MENU(0) and assigned to *MenuSel*.

```
'*
'* Decide which menu item selected and take action.
'*
HandleMenu:
   MenuSel = MENU(0)                                    'get number of menu
   ItemSel = MENU(1)                                    'get number of item

   '*
   '* Go to appropriate subroutine for menu selected.
   '*
   ON MenuSel GOSUB FileMenu, OptionMenu, SaveMenu, PhoneMenu
   MENU
   RETURN
```

Figure 12-16. Managing the menus

Upon returning from the menu subroutine, the program encounters the MENU statement, used this time with no arguments. This statement changes the title of the menu in the menu bar from the highlighted state back to normal black-on-white.

After typing the *HandleMenu* subroutine, stub out the menu subroutines. Now when you select an item from your new menu, the title is highlighted, the program takes a short side trip to your stub, the title is returned to normal by the MENU statement, and the program returns to where it was when the selection was made.

Each of the menu subroutines branched to from *HandleMenu* starts with another computed GOSUB statement, to direct the program to the proper subroutine for the specific item chosen. In the program listing, I have organized these subroutines into four groups, labeled A through D, for the four menus, and then further divided each group into the number of items on that menu. For example, A(1) contains the subroutine for the first item on the first menu, C(2) contains the subroutine for the second item on the third menu, and so on. Now let's get on with the process of fleshing out the File menu.

The File menu

Type the computed GOSUB statement shown in Figure 12-17, and stub out the five menu items. Then drop down to *DoneBas* and *DoneDesk* and type the routines shown in Figure 12-18.

```
'*
'* File menu was selected.
'*
FileMenu:
   ON ItemSel GOSUB ViewFile, ReceiveFile, SendFile, DoneBas, DoneDesk
   RETURN

ViewFile:
   RETURN

ReceiveFile:
   RETURN

SendFile:
   RETURN

DoneBas:

DoneDesk:
```

Figure 12-17. The *FileMenu* subroutine

```
'*
'* A(4): Return to BASIC.
'*
DoneBas:
   MENU RESET                                    'restore BASIC menu
   CLOSE                                         'close all open files
   END                                           'return to BASIC

'*
'* A(5): Return to Macintosh desktop.
'*
DoneDesk:
   CLOSE
   SYSTEM
```

Figure 12-18. The *DoneBas* and *DoneDesk* routines

Notice that these aren't subroutines, in that they end the program rather than returning to the main program. *DoneBas* provides an orderly method of retreating to BASIC. The advantage of using this method over pressing Command-period is that this routine restores the BASIC menu before closing all open files and returning to BASIC. *DoneDesk* issues the SYSTEM statement, to quit BASIC and return to the Macintosh Finder.

The *ViewFile*, *ReceiveFile*, and *SendFile* subroutines

The next three features operate in much the same manner, with very similar subroutines. The object in each case is to move information from one place to another: *ViewFile* transfers information from a disk file to the screen; *ReceiveFile* transfers information received at the communication port to a disk file, simultaneously displaying it on the screen; and *SendFile* transfers information from a disk file to the communication port so it can be sent to another computer. Selecting any one of the three puts a check mark in front of it on the menu and disables the other two. When the file transfer is complete, the check mark is removed and the disabled item is enabled. Selecting the checked item again, before the file transfer is compete, aborts the transfer and returns everything to normal. Since the procedures for each selection are almost identical, I will explain the one for *ViewFile* and then point out the differences for the other selections. (Because they use so many of the same routines, you may as well enter all three before we do a test.)

Selecting View File from the File menu causes the program to branch to the *ViewFile* subroutine (Figure 12-19), which allows you to view a text file while running the communication program. You may want to do this before sending the file, or to check a file after receiving it.

Since the program calls *ViewFile* both when you ask to start viewing a file and when you ask to stop viewing before the end of a file, the first thing the *ViewFile* subroutine does is check the value of *startFlag*, to see whether you are starting to view a file or already in the process of viewing one. If this flag is not *true*, then the program continues with the subroutine, where viewing is enabled and *startFlag*, among other flags, is set to *true*. If *startFlag* is already *true* when *ViewFile* is called, the program jumps to *EndFile*, a common ending routine for *ViewFile*, *ReceiveFile*, and *SendFile*.

```
'*
'* A(1): View file before sending or after receiving.
'*
ViewFile:
   IF startFlag = true THEN GOTO EndFile                    'terminating ViewFile
   SaveScreen
   filename$ = FILES$(1, "TEXT")
   RestoreScreen
   IF filename$ = "" THEN RETURN
   MENU 1, 2, 0                                             'disable Receive File
   MENU 1, 3, 0                                          'put check mark by Send File
   MENU 1, 1, 2                                          'put check mark by View File
   MENU 2, 0, 0                                             'disable Options menu
   MENU 3, 0, 0                                             'disable Save menu
   MENU 4, 0, 0                                             'disable Phone menu
   OPEN filename$ FOR INPUT AS #2
   oldSaveFlag = saveFlag                                   'store state of save flag
   saveFlag = false                                         'if previously saving, stop
   viewFlag = true
   startFlag = true
   RETURN
```

Figure 12-19. The *ViewFile* subroutine

The next thing we want to do is find out which file the user wants to view (or send, in the case of the *SendFile* subroutine). You already know how to get this information, using the FILES$(1) function. A new twist introduced in this program is the idea of saving the information that is already on the screen, where the FILES$ dialog box will appear. This is done with the same GET and PUT statements you used in Chapter 7 to move images around the screen.

Since restoring the screen after a dialog box has been displayed is frequently required during the program, I have written two subprograms that work together to accomplish the task: *SaveScreen* and *RestoreScreen* (see Figure 12-20 on the next page). These two subprograms capture the image on a portion of the screen and return it to the same area. Rather than use a different subprogram for each dialog box, or pass screen coordinates in order to save exactly the area covered, I always save the area covered by the largest dialog box used, after making sure that all the rest of the dialog boxes are displayed within that area.

```
'*
'* Save maximum screen area used by any dialog box, before displaying dialog box.
'*
SUB SaveScreen STATIC
  SHARED scrnsave()
  GET (39, 7) - (455, 187), scrnsave(0)
END SUB

'*
'* Display previously saved screen area after removing dialog box.
'*
SUB RestoreScreen STATIC
  SHARED scrnsave()
  PUT (39, 7), scrnsave(0), PSET
END SUB
```

Figure 12-20. Saving and restoring the screen

There is one critical difference between the format used for the graphic calls here and that used in Chapter 7. The syntax of the PUT statement allows an optional *action-verb* at its end:

PUT (*x1,y1*) [- (*x2,y2*)], *array* [(*INDEX*[(, *index*, ... *index*])][, *action-verb*]

This verb, which can be PSET, PRESET, AND, OR, or XOR, determines the interaction between the stored image and the one on the screen. If the option is ignored, as it was in our previous programs, the XOR default is used. This would still work if we covered and then uncovered exactly the area we saved, but if you either save more than you cover or cover more than you save, you will end up with a white border around the text when it is returned to the screen. The *action-verb* PSET solves this problem. Try the different options after you get this section running, just to see what happens.

Back to viewing the file.... Once the *ViewFile* subroutine has saved the screen, retrieved a file name, and restored the screen, it disables the other two menu items and puts a check mark by itself on the menu. With all three routines, the entire Options, Save, and Phone menus are disabled and *saveFlag* is set to *false* after its current

condition is stored. (The selections available from the Save menu allow you to toggle the saving of text on and off as the text passes across the screen. Saved text is appended to a file that is automatically opened with a file name consisting of the time of day that you first started saving. There is no point in saving information in this manner if you are viewing, receiving, or sending a file, since you either have it saved on disk already, or are about to save it.) The *ViewFile* and *SendFile* subroutines then open the selected file for input, while the *ReceiveFile* subroutine opens it for output. Then all three subroutines set the *viewFlag*, *receiveFlag*, or *sendFlag* (as appropriate) and the *startFlag* to *true*.

The only differences in the *ReceiveFile* subroutine (Figure 12-21) are that FILES$(0) is used to prompt the user for a name under which to store the file, and different menus and flags are set.

```
'*
'* A(2): Transfer information received at COM1: to disk file.
'*
ReceiveFile:
   IF startFlag = true THEN EndFile                    'terminating ReceiveFile
   SaveScreen
   filename$ = FILES$(0, "Name to save file under")
   RestoreScreen
   IF filename$ = "" THEN RETURN
   MENU 1, 3, 0                                        'disable Send File
   MENU 1, 2, 2                                'put check mark by Receive File
   MENU 1, 1, 0                                        'disable View File
   MENU 2, 0, 0                                    'disable Options menu
   MENU 3, 0, 0                                        'disable Save menu
   MENU 4, 0, 0                                        'disable Phone menu
   OPEN "O", #2, filename$
   oldSaveFlag = saveFlag
   saveFlag = false
   receiveFlag = true                                  'turn on receiving
   startFlag = true
   RETURN
```

Figure 12-21. The *ReceiveFile* subroutine

The *SendFile* subroutine, shown in Figure 12-22, is also similar to *ViewFile,* except that after FILES$(1) is used to get the name of an existing file, the file is opened for input and *sendFlag* is set to *true.*

The *EndFile* subroutine

The *EndFile* subroutine, shown in Figure 12-23, is common to all three file operations. It is the routine branched to when you quit in the middle of a file operation by choosing the checked selection from the File menu.

This subroutine begins by closing file #2, which has been opened by one of the preceding File subroutines to either get or receive information. Then it puts the cursor back on the screen, since we have been ignoring that task, and prints *endMessage$.* Finally, it restores the menus and flags to their normal conditions and sets three new

```
'*
'* A(3): Transmit file stored on disk.
'*
SendFile:
   IF startFlag = true THEN EndFile                    'terminating SendFile
   SaveScreen
   filename$ = FILES$(1, "TEXT")
   RestoreScreen
   IF filename$ = "" THEN RETURN
   MENU 1, 2, 0                                        'disable Receive File
   MENU 1, 3, 2                                  'put check mark by Send File
   MENU 1, 1, 0                                        'disable View File
   MENU 2, 0, 0                                     'disable Options menu
   MENU 3, 0, 0                                        'disable Save menu
   MENU 4, 0, 0                                       'disable Phone menu
   OPEN "I", #2, filename$
   oldSaveFlag = saveFlag
   saveFlag = false
   sendFlag = true                                          'turn on sending
   startFlag = true
   RETURN
```

Figure 12-22. The *SendFile* subroutine

```
'*
'* Close file and re-enable various competing menus
'*
EndFile:
   CLOSE #2
   ShowCur
   PrintString endMessage$
   MENU 1, 2, 1                          'enable Receive File option
   MENU 1, 3, 1                          'enable Send File option
   MENU 1, 1, 1                          'remove check mark
   MENU 2, 0, 1                          'enable Options menu
   MENU 3, 0, 1                          'enable Save menu
   MENU 4, 0, 1                          'enable Phone menu
   saveFlag = oldSaveFlag                'restore state of Save
   viewFlag = false                      'turn off viewing
   sendFlag = false
   receiveFlag = false
   startFlag = false
   endFlag = false
   endSendFlag = false
   endViewFlag = false
   RETURN
```

Figure 12-23. The *EndFile* subroutine

flags: *endFlag*, *endSendFlag*, and *endViewFlag*. You will see how each of these flags is used as we work our way through *CommLoop*.

We have added a whole bunch of lines to our core program, just to do something as simple as look at a file, but if you were paying close attention, you may have noticed that not one routine included the most important lines of all: the ones to read the file and print its contents. We could add these features to each of the subroutines, but since we already have most of the commands we need built into *CommLoop*, we might as well make them do double duty.

Modifying *CommLoop*

Let's return to the simple *CommLoop* routine and add a call to the short subprogram we'll create in a moment to allow you to actually view a file. Right after the

line that prints *keyTyped$* (*IF keyTyped$ <> "" THEN PRINT #1, keyTyped$*), add these lines:

```
MENU STOP
IF viewFlag THEN CALL SendLine
IF endViewFlag THEN GOSUB EndFile
MENU ON
```

I hope you appreciate the ease and speed with which you were able to enter those lines. They are the result of an hour or so of frustrating experimentation on my part, trying to understand the cause of seemingly inexplicable error messages, after starting with the line:

```
IF viewFlag THEN SendLine
```

It turns out that this is one case in which the CALL statement is not optional: If you call a subprogram from the THEN or ELSE portion of an IF...THEN...ELSE statement, CALL is required to make it clear that the name refers to a subprogram, not a line label. An exception to this exception seems to occur if you also pass an argument to the subprogram, in which case BASIC recognizes it as a subprogram regardless of where it is. These situations are not documented in the BASIC manual, so you might make special note of them.

The reason for testing *endViewFlag* is to see if the end of the file has been reached, and if so to branch to the *EndFile* subroutine. We will add a similar test for *endSendFlag*, but at a different location in the program. We differentiate between the endings of these two processes in order to allow the last characters sent out the communication port to be retrieved and printed on the screen (viewed) before the end message is printed.

MENU STOP, another variation of the MENU command, must be added at the beginning of this sequence to suspend the practice of branching to a subroutine when a menu event is trapped (a record of events trapped will be kept, and they will be responded to when the MENU ON statement is issued). Here's why we suspend event

```
'*
'* View or send a file.
'*
SUB SendLine STATIC
   SHARED viewFlag, endViewFlag
   SHARED true, false
   LINE INPUT #2, lin$                                    'get line from file
   PRINT lin$                                             'send it someplace
   IF NOT EOF(2) THEN EXIT SUB
   endViewFlag = true
   viewFlag = false
END SUB
```

Figure 12-24. The *SendLine* subprogram

trapping: The obvious menu event to trap while viewing a file is the reselection of View File to abort viewing, which closes file #2. If this reselection is made between the time the *SendLine* subprogram is called and the time it tries to input a line from file #2, the program is stopped and a Bad File Number error message is generated.

Finally, go to your subprogram section and add the *SendLine* subprogram, shown in Figure 12-24.

Now you can fire up the program and choose View File from the File menu (don't choose Receive File or Send File yet, since we haven't added the last bit of code they need). If you don't have a text file on the disk, you can use the options in the FILES$(1) dialog box to change drives or to eject the disk and insert one containing a text file. After you select a file, it should scroll across your screen; at the end of the file (or if you abort viewing from the menu), the end message will be printed.

Finishing up: Sending

Let's go ahead and wrap up the other two sections. Finishing the Send File option is pretty easy: Go back to the lines you just added to *CommLoop* and change the second one to:

```
IF (viewFlag OR sendFlag) THEN CALL SendLine
```

To avoid the same problems that I had in the View File section, you will also want to drop down to *PrintString lin$* and add a few lines on either side of it, so that it looks like this:

```
MENU STOP
PrintString lin$
MENU ON
IF endSendFlag THEN GOSUB EndFile
```

and then go on down to the *SendLine* subprogram and add the highlighted items in Figure 12-25.

You should now be able to use this communication program to send a file. Test it by sending a file to another computer, or by simply turning your modem on and sending the file without calling anyone. If the file appears on your screen, it must have made it out the communication port and back in.

The only other feature we will add to the Send File option is something variously called flow control, handshaking, or XON/XOFF. All of these terms refer to the ability of the receiving computer to tell the sending computer to stop sending for a while. This is usually done when information is arriving at the computer faster than it can be

```
'*
'* View or send a file.
'*
SUB SendLine STATIC
   SHARED viewFlag, endViewFlag
   SHARED sendFlag, endSendFlag, true, false
   LINE INPUT #2, lin$                                    'get line from file
   IF sendFlag THEN PRINT #1, lin$ ELSE PRINT lin$        'send it someplace
   IF NOT EOF(2) THEN EXIT SUB
   IF viewFlag THEN endViewFlag = true
   IF sendFlag THEN endSendFlag = true
   viewFlag = false : sendFlag = false
END SUB
```

Figure 12-25. The final *SendLine* subroutine

brought in and processed. When the input buffer fills up to a certain point, the receiving computer sends an XOFF signal and the communication program in the other computer stops sending information. When the buffer is emptied, the receiving computer sends an XON signal and transmission resumes. We have already defined *xon$* and *xoff$* as CHR$(17) and CHR$(19) in the *InitializeVariables* subroutine: Now we have to decide when to send them and what to do when we receive them.

If we are sending a file, we have to watch the incoming data for an XOFF. If we receive an XOFF, we have to stop sending until we receive an XON. So far so good. The next step is figuring out how to recognize these specific characters in the input, when we often bring in a whole bunch of characters at a time. At this point, some programs revert to bringing in characters one at a time and testing each against a list of "special" characters (XON, XOFF, carriage return, and so on). This works well up to about 300 baud, but beyond that it slows down communication too much, so we will develop a subprogram that searches a string (the line read in from the input port) for a specific character, removes it if found, and then sets a flag to tell us it was there. We can use this subprogram as a general-purpose character stripper, and it will also serve to tell us if we receive an XON or XOFF, which we would always want to strip anyway. Figure 12-26 shows *RemoveChars,* the subprogram that will do the job. When we call *RemoveChars,* we will pass it the line to be searched and the character to remove.

```
'*
'* Remove passed character from passed line.
'*
SUB RemoveChars(lin$, char$) STATIC
   SHARED removeFlag
   removeFlag = false                                          'reset flag
   position = INSTR(lin$, char$)                    'where is first offensive character
   IF position = 0 THEN EXIT SUB                                'there wasn't one
   removeFlag = true                                             'there was one
   WHILE position > 0                                 'remove first and check for more
      lin$ = LEFT$(lin$, position - 1) + RIGHT$(lin$, LEN(lin$) - position)
      position = INSTR(lin$, char$)
   WEND
END SUB
```

Figure 12-26. The *RemoveChars* subprogram

When we leave this subprogram, *removeFlag* will be set to *true* if the character was found, so the first thing we have to do is set it to *false*, in case the previous pass set it to *true*. Then we use the INSTR function to see if the string contains the character we are looking for. This is a very useful function; its full syntax is:

INSTR([I,]*X$*, *Y$*)

INSTR searches for the first occurrence of *Y$* in *X$*. If you don't want to start the search with the first character of *X$*, you can use the optional offset, I, to specify where to start. If *Y$* is not found in *X$*, the function returns 0; otherwise it returns the character position at which the match is found. In our subprogram, we assign the returned value to the variable *position*, and then test *position* to see if it is 0. If it is, meaning the character was not in the string, we use EXIT SUB to return to the main program. If, on the other hand, *position* is greater than 0, we set *removeFlag* to *true* and continue through the subprogram.

Within the WHILE...WEND loop, *lin$* is restructured by setting it equal to the left portion of *lin$*, up to the character before *position*, plus the right portion from

```
'*
'* Display characters from COM1:, send keystrokes to COM1:.
'*
CommLoop:
  WHILE true
    WHILE LOC(1) = 0                                    'nothing waiting to come in
      keyTyped = INKEY$                                 'send key typed to file #1
      IF keyTyped <> "" THEN PRINT #1, keyTyped$
      MENU STOP                                         'don't get sidetracked
      IF (sendFlag OR viewFlag) THEN CALL SendLine
      IF endViewFlag THEN GOSUB EndFile
      MENU ON                                           'see if anything selected
    WEND
    lin$ = INPUT$(LOC(1), 1)                            'get everything waiting
    MENU STOP
    PrintString lin$
    MENU ON
    IF endSendFlag THEN GOSUB EndFile
  WEND
```

Figure 12-27. The modified *CommLoop* routine

the character after *position* to the end of the string. Then the string is tested again. If there is another "special" character, the process is repeated; otherwise the subprogram ends. Add *RemoveChars* to your subprogram section, and then we'll return to *CommLoop* and figure out when we need to call it. (*CommLoop* should now look like Figure 12-27.)

Since we want to remove the extra character between the time we receive it and the time we'd otherwise print it, we have to call our new subprogram between the lines *lin$ = INPUT$(LOC(1), 1)* and *MENU STOP*. So insert the line:

```
RemoveChars lin$, xoff$
```

between those two lines. Then you will immediately want to test *removeFlag*, to see if an XOFF was in fact removed (actually, you don't always have to check *removeFlag*—only when you are sending a file). This next line will do the trick:

```
IF sendFlag AND removeFlag THEN waitFlag = true
```

So, if we do receive an XOFF and set *waitFlag* to *true*, where is the best place to test *waitFlag* to prevent any more lines from being sent? Let's go back up a few lines to where we call *SendLine* if *viewFlag* and *sendFlag* are *true* and insert a check for *waitFlag* there. The line will look like this:

```
IF sendFlag OR viewFlag AND NOT waitFlag THEN CALL SendLine
```

That takes care of stopping the transmission if we receive an XOFF. Now, we need to turn it back on when we receive an XON. Sounds like another job for *RemoveChars*. Insert these two lines just above or below the two lines you added for XOFF:

```
RemoveChars lin$, xon$
IF sendFlag AND waitFlag AND removeFlag THEN waitFlag = false
```

```
'*
'* Get keystroke from keyboard and send it out COM1:.
'*
SUB SendKey STATIC
  keyTyped$ = INKEY$
  IF keyTyped$ <> "" THEN PRINT #1, keyTyped$;
END SUB
```

Figure 12-28. The *SendKey* subprogram

Before we tackle the receive side of flow control, let's update *CommLoop* again and make sure it works. First, to cut down on the clutter let's convert to a subprogram the two lines near the top that send a character typed at the keyboard out the communication port. We'll call the subprogram *SendKey,* and it will look like Figure 12-28. Nothing new here, just clean-up.

There is one more character we will almost always want to remove, since the Macintosh automatically inserts it when a carriage return is received, and that character is the linefeed. To take care of this, insert the line:

```
RemoveChars lin$, lf$
```

above, below, or between the two sets of lines that remove and test for XON and XOFF. Just don't put the new line between a *RemoveChars* call and the line that tests *removeFlag,* or *waitFlag* will be set or reset whenever a linefeed is received.

Now for a quick test. If you have added a call to remove linefeeds, replace *lf$* with some letter of the alphabet, enclosed in quotation marks. Start the program and "send" a file to your modem—no need to be connected to another computer. The text that is reflected back to your screen should not contain the character you passed to the *RemoveChars* subprogram.

If you would like to experiment a bit more, write another subprogram, similar to *RemoveChars,* that replaces one character with another. Pass to the subprogram the

string, the character to remove, and its replacement. A line similar to this next one will probably play an important role in the subprogram.

```
lin$ = LEFT$(lin$, position - 1) + newChar$ + RIGHT$(lin$, LEN(lin$) - position)
```

Finishing up: Receiving

Now that you can easily send the files on your disk to other computers, let's finish off the process of receiving a file from someone else and storing it in a disk file. When Receive File is chosen from the File menu, the program goes to a subroutine that solicits a name under which to store the file, opens that file as #2, and sets *receiveFlag* to *true.* It then returns to *CommLoop,* where characters are brought in from the communication port and displayed on the screen. If we want to find out if those characters should also be put in a file, we can have *CommLoop* check the status of *receiveFlag.* A good time to do this is right after each line is displayed on the screen (after *PrintString lin$*). This next line, inserted at that point, will print the line in file #2 if *receiveFlag* is *true.*

```
IF receiveFlag THEN PRINT #2, lin$
```

That was pretty easy. Now, what do we do if data is coming in faster than we can process it? Well, best tell the other computer to stop for a moment—sounds like just the job for *xoff$.*

When we opened COM1:, we specified a buffer length of 2000 characters. All incoming data goes into the buffer and *CommLoop* retrieves it from the buffer with *lin$ = INPUT$(LOC(1), 1).* At modem transmittal speeds (300 and 1200 baud), we will probably never receive data faster than it can be retrieved and processed, but with a higher-speed connection to another computer, it is likely we would lose data if we couldn't stop transmission every now and then. If we wait until the buffer is full to tell the other computer to stop, we will lose what came in between the time the buffer

filled and the time the other computer got around to interpreting our XOFF and responding to it. So when we defined the variable *bufferLimit* in the *InitializeVariables* section, we set it equal to a percentage of the buffer. Now we can test *bufferLimit* and start the XOFF procedure as soon as it is exceeded.

The best place to test the buffer is just before the line that inputs all characters from the buffer and assigns them to *lin$* (*lin$ = INPUT$(LOC(1), 1)*, since that is when it will contain the most characters. Here is the line you want to insert:

```
IF LOC(1) > bufferLimit THEN PRINT #1, xoff$; : stopFlag = true
```

And as you know from working out flow control for the sending routine, after you finish processing the data in the buffer you have to tell the other computer to resume sending. This next line will do that job. A good place to insert it is just before the *WHILE LOC(1) = 0...WEND* loop near the top of *CommLoop*.

```
IF LOC(1) = 0 AND stopFlag THEN PRINT #1, xon$; : stopFlag = false
```

This takes care of the entire viewing, receiving, and sending sections of our program. The *CommLoop* routine should now look like Figure 12-29. (The lines that we have entered since the last complete listing are highlighted.) There will be a few minor additions to *CommLoop*, when we add the Pause and Save features, and then we will be through with it.

The Options menu

This section of code includes a short subroutine and the section from the Terminal program that sets communication parameters. The new item is a Pause selection, which stops the processing of data until Pause is selected again. The stubbed version of *OptionMenu* is shown in Figure 12-30.

```
'*
'* Display characters from COM1:, send keystrokes to COM1:.
'*
CommLoop:
   WHILE true
      IF (LOC(1) = 0) AND stopFlag THEN PRINT #1, xon$; : stopFlag = false
      WHILE LOC(1) = 0                                          'nothing waiting to come in
         SendKey                                                'send key typed to file #1
         MENU STOP                                              'don't get sidetracked
         IF sendFlag OR viewFlag AND NOT waitFlag THEN CALL SendLine
         IF endViewFlag THEN GOSUB EndFile
         MENU ON                                                'see if anything selected
      WEND
      IF LOC(1) > bufferLimit THEN PRINT #1, xoff$; : stopFlag = true
      lin$ = INPUT$(LOC(1), 1)                                  'get everything waiting
      RemoveChars lin$, lf$                                     'strip linefeeds
      RemoveChars lin$, xon$
      IF sendFlag AND waitFlag AND removeFlag THEN waitFlag = false
      RemoveChars lin$, xoff$
      IF sendFlag AND removeFlag THEN waitFlag = true
      MENU STOP
      PrintString lin$
      MENU ON
      IF endSendFlag THEN GOSUB EndFile
      IF receiveFlag THEN PRINT #2, lin$;
   WEND
```

Figure 12-29. The updated *CommLoop*

```
'*
'* Options Menu was selected.
'*
OptionMenu:
   ON ItemSel GOSUB Pause, Config
   RETURN

Pause:
   RETURN

Config:
   RETURN
```

Figure 12-30. The *OptionMenu* subroutine

The *Pause* subroutine

The first time Pause (Figure 12-31) is chosen from the Options menu, it stops the flow of whatever data was being viewed, received, or sent. The next time it is chosen, it starts things back up, right where they left off. To accomplish this it toggles the condition of *pauseFlag* between *false* and *true* (0 and −1) by setting *pauseFlag* equal to itself XORed with −1 (the value of *true*). The Pause menu selection itself is also toggled between an unchecked state and a checked state by using the value of *pauseFlag* to control the value of the *state* argument (*state* is set equal to −1 * *pauseFlag* + 1).

The final action of this subroutine, before returning, is to send an XOFF if *Pause* is being started or an XON if it is being stopped, so that the buffer won't be filling up while we aren't removing anything.

To prevent the program from continuing to loop through *CommLoop*, we'll need to add a line. Just after *WHILE true*, insert:

```
WHILE pauseFlag : WEND
```

This definitely wraps up *CommLoop;* I promise not to drag you back here for any more additions. Although this program is much larger than the Terminal program, all the real work takes place in *CommLoop* (which should now look like Figure 12-32), just as it did in Terminal. If you understand what this section does, then you understand the essence of telecommunication, and if you understand how BASIC performs this communication, then you understand more about BASIC than most people do.

```
'*
'* B(1): Stop processing of data.
'*
Pause:
   pauseFlag = pauseFlag XOR -1
   MENU 2, 1, -1 * pauseFlag + 1
   IF pauseFlag THEN PRINT #1, xon$; ELSE PRINT #1, xoff$;
   RETURN
```

Figure 12-31. The *Pause* subroutine

```
'*
'* Display characters from COM1:, send keystrokes to COM1:.
'*
CommLoop:
  WHILE true
    WHILE pauseFlag : WEND
    IF (LOC(1) = 0) AND stopFlag THEN PRINT #1, xon$; : stopFlag = false
    WHILE LOC(1) = 0                                    'nothing waiting to come in
      SendKey                                           'send key typed to file #1
      MENU STOP                                         'don't get sidetracked
      IF (sendFlag OR viewFlag) AND NOT waitFlag THEN CALL SendLine
      IF endViewFlag THEN GOSUB EndFile
      MENU ON                                           'see if anything selected
    WEND
    IF LOC(1) > bufferLimit THEN PRINT #1, xoff$; : stopFlag = true
    lin$ = INPUT$(LOC(1), 1)                            'get everything waiting
    RemoveChars lin$, lf$                               'strip linefeeds
    RemoveChars lin$, xon$
    IF sendFlag AND waitFlag AND removeFlag THEN waitFlag = false
    RemoveChars lin$, xoff$
    IF sendFlag AND removeFlag THEN waitFlag = true
    MENU STOP
    PrintString lin$
    MENU ON
    IF endSendFlag THEN GOSUB EndFile

    '*
    '* File #3 is automatically named file that stores input when
    '* Start or Continue is chosen from Save menu. File #2
    '* stores received file after asking for name to store it under.
    '*
    IF saveFlag THEN PRINT #3, lin$;
    IF receiveFlag THEN PRINT #2, lin$;
  WEND
```

Figure 12-32. The final *CommLoop*

The *Config* subroutine

The Config subroutine used here is almost identical to the one used by the Terminal program. The main differences are that a few of the variable names are longer, and the buttons in the window are closer together. Figure 12-33 shows the new listing. The changes are so minor that the comments should explain them adequately.

```
'*
'* Set communication parameters.
'*
Config:
   CLOSE 1                                                    'close COM1:
   SaveScreen
   WINDOW 2, , (50, 35) - (450, 185), 2                       'open new window
   GOSUB DisplayDefaults                                      'show parameters
   GOSUB SelectOptions                                        'get selection
   options$ = choice$(1) + "," + choice$(2) + "," + choice$(3) + "," + choice$(4)

   '*
   '* Open communications port.
   '*
   OPEN "COM1:" + options$ AS 1 LEN = buffer
   WINDOW CLOSE 2
   RestoreScreen
   RETURN

DisplayDefaults:
   RESTORE                                                    'to read data statements from start
   FOR count = 1 TO 16
      READ x, y, group(count), nam$(count)
      BUTTON count, 1, nam$(count), (x, y) - (x + 130, y + 15), 3
   NEXT count
   BUTTON 17, 1, "OK", (320, 110) - (380, 135)

   '*
   '* Simulate button pushes to highlight defaults.
   '*
   FOR count = 1 TO 4
      ButtonSelect choice(count)
   NEXT
   RETURN

'*
'* Here is x,y coordinate of button, groupNum, title.
'*
   DATA 10, 10, 1, 110 bits per sec
   DATA 10, 30, 1, 300 bits per sec
   DATA 10, 50, 1, 600 bits per sec
   DATA 10, 70, 1, 1200 bits per sec
   DATA 10, 90, 1, 2400 bits per sec
```

Figure 12-33. The *Config* subroutine

more...

```
   DATA 10, 110, 1, 4800 bits per sec
   DATA 10, 130, 1, 9600 bits per sec

   DATA 292, 10, 2, No parity
   DATA 292, 30, 2, Even parity
   DATA 292, 50, 2, Odd parity

   DATA 162, 10, 3, 5 Data bits
   DATA 162, 30, 3, 6 Data bits
   DATA 162, 50, 3, 7 Data bits
   DATA 162, 70, 3, 8 Data bits

   DATA 162, 110, 4, 1 Stop bits
   DATA 162, 130, 4, 2 Stop bits

SelectOptions:
   eventTrapped = DIALOG(0)                                    'dialog event
   IF eventTrapped <> 1 THEN SelectOptions                     'watch for button press
   buttonPushed = DIALOG(1)                                    'which button

   '*
   '* Get button number if through making selections (clicked OK).
   '*
   IF buttonPushed < 17 THEN ButtonSelect buttonPushed : GOTO SelectOptions
   RETURN

'*
'* Subprogram ButtonSelect(buttonPushed) is called from DisplayDefaults and
'* SelectOptions subroutines. User has just pushed button. Highlight
'* that button and remember selection in choice() and choice$().
'*
SUB ButtonSelect(buttonPushed) STATIC
   SHARED nam$(), group(), choice(), choice$()
   groupNum = group(buttonPushed)                              'group 1, 2, 3, or 4
   BUTTON choice(groupNum), 1                                  'reset old selection
   BUTTON buttonPushed, 2                                      'set new selection
   choice(groupNum) = buttonPushed                             '1 through 16
   IF groupNum = 2 THEN SetParity                              'from parity group
   choice$(groupNum) = STR$(VAL(nam$(buttonPushed)))           'from other group
   EXIT SUB
SetParity:
   choice$(groupNum) = LEFT$(nam$(buttonPushed), 1)            'get first letter
END SUB
```

Figure 12-33. The *Config* subroutine *(continued)*

The Save menu

The three selections on the Save menu allow the user to start, stop, or continue saving information that passes across the screen. These menu items are obviously interdependent—you can't stop saving something you haven't started saving, or continue when you haven't started and then stopped—so we will use BASIC's ability to enable and disable menu items to let the user know which selection is available at any moment. Figure 12-34 shows the standard computed GOSUB used to branch to the routine appropriate to the item selected from the menu.

Start saving

In order to start saving information without bothering the user with a request for a file name, some sort of unique file name must automatically be assigned. As shown in Figure 12-35, the *StartSave* subroutine does this by using the TIME$ function to retrieve the current time, and then using that value, preceded by the word *Saved,* as the file name. The only difficulty in doing this is that the time is expressed with colons separating the hours, minutes, and seconds, and the Macintosh uses a colon to separate the volume name (the name assigned to the disk) from the file name. We could

```
'*
'* Save Menu was selected.
'*
SaveMenu:
   ON ItemSel GOSUB StartSave, ContSave, StopSave
   RETURN

StartSave:
   RETURN

ContSave:
   RETURN

StopSave:
   RETURN
```

Figure 12-34. The *SaveMenu* subroutine

```
'*
'* C(1): Start saving.
'*
StartSave:
   tim$ = TIME$                                    'get current time for filename
   FOR count = 3 TO 6 STEP 3                          'change : to /
      MID$(tim$, count) = "/"
   NEXT count
   filename$ = "Saved " + tim$                        'assign filename
   OPEN "A", #3, filename$                            'open new file
   saveFlag = true                                 'set save flag to true
   MENU 3, 1, 0                                       'disable Start
   MENU 3, 3, 1                                       'enable Stop
   RETURN
```

Figure 12-35. The *StartSave* subroutine

write a loop to test each character and replace it if it is a colon, but since we know which characters are always colons, we can use an easier method to replace them automatically: a FOR...NEXT loop that uses the MID$ statement to replace the third and sixth characters with slashes. The syntax of the MID$ statement is:

MID$(*string-exp1, n*[, *m*]) = *string-exp2*

This statement replaces *m* characters in *string-exp1* with the same number of characters from *string-exp2,* starting with the character in position *n* in *string-exp1.* The variable *m* is optional; if omitted, as our program does, all of *string-exp2* will be used in place of *n.*

Before returning, *StartSave* also sets *saveFlag* to *true,* disables the Start menu selection, and enables the Stop menu selection.

Continue saving

The menu item that leads to the *ContSave* subroutine can be selected only if saving has been started and then stopped. If those conditions exist and Continue is chosen from the Save menu, *ContSave* (Figure 12-36 on the next page) sets *saveFlag* to *true,* disables itself, and enables the Stop selection.

```
'*
'*C(2): Continue saving (if stopped).
'*
ContSave:
   saveFlag = true
   MENU 3, 2, 0                                    'disable Continue
   MENU 3, 3, 1                                     'enable Stop
   RETURN
```

Figure 12-36. The *ContSave* subroutine

Stop saving

The *StopSave* subroutine, shown in Figure 12-37, is essentially the same as the previous section, except that it sets *saveFlag* to *false,* enables the Continue selection, and disables itself.

That's all there is to the Save menu. Its selections are useful when trying to conserve disk storage space while connected to a remote computer that has a little useful information buried in a bunch of irrelevant material. You simply toggle saving on as you get into the interesting parts, and toggle it off as you leave them.

Testing the Save section

The obvious test for the Save menu is to fire the program up and save portions of a communication session. You can also try interrupting this save routine to receive a file under a name you assign by choosing the Save option from the File menu.

```
'*
'*C(3): Stop saving.
'*
StopSave:
   saveFlag = false                          'set save flag to false
   MENU 3, 2, 1                               'enable Continue
   MENU 3, 3, 0                                'disable Stop
   RETURN
```

Figure 12-37. The *StopSave* subroutine

```
'*
'* Phone Menu was selected.
'*
PhoneMenu:
  ON ItemSel GOSUB Directory, EnterNumber, Disconnect, Redial
  RETURN

Directory:
  RETURN

EnterNumber:
  RETURN

Disconnect:
  RETURN

Redial:
  RETURN
```

Figure 12-38. The *PhoneMenu* subroutine

The Phone menu

The items on the Phone menu handle the various telephone-related functions provided by the program. They allow the user to enter a telephone number that the computer will then dial, or to hang up the phone, redial, or select a number from a list stored on disk. The stubbed *PhoneMenu* subroutines are shown Figure 12-38. The only new concept introduced in the subroutines for the first three options is edit fields: the text-input function common to many Macintosh dialog boxes.

Entering a number

The section branched to when the Enter Number option is chosen from the Phone menu consists of three subroutines. The first, *EnterNumber,* creates the dialog box shown in Figure 12-39 on the next page, to ask the user for a phone number. The other two subroutines, *EnterLoop* and *EnterContinue,* retrieve the number and pass it to the *SendToModem* subprogram, which dials it.

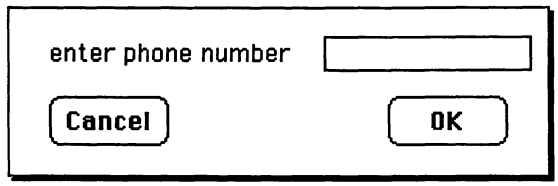


Figure 12-39. The number-entry dialog box

The *EnterNumber* subroutine

EnterNumber, which opens a window, prints some instructions, and then creates an edit field and two buttons, is shown in Figure 12-40.

The new statement in this section of code, EDIT FIELD, belongs in the same class as DIALOG, WINDOW, and MOUSE: They all pack a lot of power. Here is the generic sytax for EDIT FIELD:

EDIT FIELD *ID*[, *default, rectangle*[, [*type*][, *justify*]]]

The *ID* is a number greater than zero used to identify a particular edit field in a window. Typically, the fields are numbered consecutively from 1. Just as with buttons, these ID numbers are unique to the window in which they are used, so an edit field with the same number in two different windows will not cause a conflict.

The optional *default* is the text to be edited. If you provide this text, it will automatically appear in the edit field. The *default* can be specified as actual text, enclosed

```
'*
'* D(2): Create dialog box.
'*
EnterNumber:
   SaveScreen
   WINDOW 4, , (100, 100) - (370, 185), -4
   MOVETO 20, 27
   PRINT "enter phone number"
   EDIT FIELD 1, phoNum$, (160, 15) - (260, 30)                    'create edit field
   BUTTON 1, 1, "Cancel", (20, 45) - (80, 70)
   BUTTON 2, 1, "OK", (190, 45) - (250, 70)
```

Figure 12-40. The *EnterNumber* subroutine

in quotation marks, or it can be a string variable defined elsewhere (if you want to display a numeric variable, use the STR$ function, as in *EDIT FIELD 1, STR$(age),...*). If you don't include *default* text, you must still include the quotation marks.

The *rectangle* argument is the same type of upper left, lower right coordinate description used to define windows and boxes in other BASIC statements, and *type* is a number from 1 to 4 that determines the editing format as follows:

Value	Meaning
1	The default. Draws a box around the rectangle to be edited; does not allow Returns in the edit field (most applications trap the Return and interpret it the same as a click of the OK button)
2	Boxed; allows Return key
3	No box; does not allow Return key
4	No box; allows Return key

Another optional argument is *justify,* a number from 1 to 3, specifying the location of text within the edit-field rectangle:

Value	Meaning
1	The default. Left-justifies text
2	Centers text
3	Right-justifies text

When a window contains more than one edit field, the fields are created in the order they are listed in the program. The contents of the last edit field created are always highlighted, and standard Macintosh Cut, Copy, and Paste techniques can be used to make changes. These edit features aren't necessary with the edit field created to input a phone number, but they can be useful elsewhere.

NOTE: Even if the Edit menu is replaced or turned off by your BASIC program, as it is by this one, the Command-key equivalents of the Cut, Copy, and Paste commands will still work.

The *EnterLoop* subroutine

Once the dialog box has been created by *EnterNumber*, the program goes into the *EnterLoop* subroutine, shown in Figure 12-41, to wait for the user to click a button or press the Return key to signal that the number has been entered.

In previous programs, we have trapped DIALOG(0) to see when a button was pushed, but this time we are interested in two dialog events, not just one:

Value	Meaning
DIALOG(0) = 1	Button in active output window selected with mouse; number of button is returned by DIALOG(1) function, and this number is used to determine which subroutine should respond to the event
DIALOG(0) = 6	Return key pressed in active window that has button or edit field that cannot accept Return as input text; treated same as click of the OK button in that window

Pressing the Return key or clicking the OK button causes the program to move on to *EnterContinue*. Clicking Cancel terminates the *EnterLoop* subroutine simply by not branching out of it before the end.

```
'*
'* Wait for user to finish entering number.
'*
EnterLoop:
   eventTrapped = DIALOG(0)
   IF eventTrapped = 6 THEN GOTO EnterContinue          'Return key pressed
   IF eventTrapped <> 1 THEN GOTO EnterLoop            'button wasn't selected
   buttonSel = DIALOG(1)
   IF buttonSel <> 1 THEN GOTO EnterContinue
   WINDOW CLOSE 4                                       'Cancel button clicked
   RestoreScreen
   RETURN
```

Figure 12-41. The *EnterLoop* subroutine

```
'*
'* Retrieve number from dialog box.
'*
EnterContinue:
   phoNum$ = EDIT$(1)                                    'retrieve number
   phoNam$ = "manually entered number"
   WINDOW CLOSE 4
   RestoreScreen
   num$ = dial$ + phoNum$
   SendToModem num$                                       'dial number
   MENU 4, 4, 1                                          'enable Redial
   RETURN
```

Figure 12-42. The *EnterContinue* subroutine

The *EnterContinue* subroutine

This subroutine, shown in Figure 12-42, uses the EDIT$ function to retrieve the contents of the edit field. The syntax for this function is:

EDIT$(*ID*)

The *ID* used here must be the same number used to open the edit field you want to retrieve information from.

You use the EDIT$ function by setting it equal to a string variable—in this case, *phoNum$*—thereby assigning the number entered in the edit field to the variable. (We deal with the phone number as a string to avoid problems with the non-numeric symbols, such as dashes and parentheses, often included in phone numbers.) The second line of the subroutine assigns the phrase "manually entered number" to the variable *phoNam$*, which is used in other parts of the program to store the name of the party being called. When the number is actually dialed, the text assigned to *phoNam$* is displayed on the screen. Once the number is retrieved, the dial command for your modem (assigned to the variable *dial$* in the *InitializeVariables* subroutine) is placed in front of it and they are passed to the *SendToModem* subprogram, shown in Figure 12-43 on the following page. After the number is dialed, the menu is reset and the program returns to *CommLoop*.

```
'*
'* Send passed string to modem, one character at a time with pause
'* between characters. Used to pass commands and phone numbers to modem.
'*
SUB SendToModem(out$) STATIC
   FOR position = 1 TO LEN(out$)                          'take numbers one by one
      code$ = MID$(out$, position, 1)
      Delay 500                                              'don't send too fast
      IF code$ = "~" THEN Delay 8000 : GOTO SkipCode              'long pause
      PRINT #1, code$;
SkipCode:
   NEXT
   Delay 5000
   PRINT #1, cr$                                          'end with carriage return
END SUB

'*
'* Delay.
'*
SUB Delay(count) STATIC
   FOR hold = 1 TO count
   NEXT
END SUB
```

Figure 12-43. The *SendToModem* subprogram

Hanging up

If the user selects Hang Up from the Phone menu, the short subroutine shown in Figure 12-44 passes *hangup$* to *SendToModem.* This variable was defined in *InitializeVariables* as *"~+++~ATH0"*, but it can easily be changed to match the modem in use (though the D.C. Hayes command used here works with most modems). The tildes (~) on either side of the three plus signs, which are used to put the modem in the command mode, cause a pause of about two seconds, so that the modem will be ready to respond to the *ATH0* that follows.

```
'*
'* D(3): Tell modem to hang up.
'*
Disconnect:
   SendToModem hangup$
   RETURN
```

Figure 12-44. The *Disconnect* subroutine

Redialing

Selecting Redial from the Phone menu causes the *Redial* subroutine, shown in Figure 12-45, to send the last number dialed (whether it was entered manually or from the directory) and your modem dial command (assigned to *dial$* in *InitializeVariables*) to *SendToModem*.

Dialing from a directory

When Directory is selected from the Phone menu, the *Directory* subroutine, shown in Figure 12-46 on the next page, calls the *Direct* subprogram, which in turn displays a list of numbers and allows the user to select one to dial. Notice that before the *Direct* subprogram is called, *Directory* sets the variable *dialFlag* to *false*. This variable is shared with the subprogram, which sets it to *true*, and assigns values to *phoNum$* and *phoNam$* if the user selects a number to dial. Upon returning from the

```
'*
'* D(4): Send number last dialed to modem.
'*
Redial:
   PRINT "Dialing "; phoNam$; ": "                    'tell us who we're calling
   num$ = dial$ + phoNum$
   SendToModem num$
   RETURN
```

Figure 12-45. The *Redial* subroutine

```
'*
'* D(1): Call Direct subprogram.
'*
Directory:
   SaveScreen
   dialFlag = false                          'set dialFlag going into subprogram
   Direct                                    'call directory subprogram
   RestoreScreen
   IF NOT dialFlag THEN RETURN               'check dialFlag on return
   PrintString "Dialing " + phoNam$ + cr$    'tell us who we're calling
   num$ = dial$ + phoNum$                    'add modem dial command
   SendToModem num$                          'send dial command and number
   RETURN
```

Figure 12-46. The *Directory* subroutine

subprogram, the *Directory* subroutine tests *dialFlag* and, if it is *true,* prints the name and dials the number just as was done when the number was entered by hand.

The *Direct* subprogram

The *Direct* subprogram produces a display similar to that shown in Figure 12-47, which is a list of phone numbers that can be edited or dialed. This subprogram is really a fairly major program which, with a few modifications, could be split off to

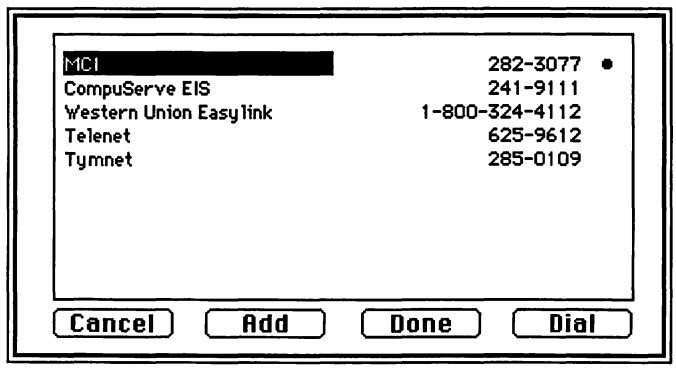


Figure 12-47. The directory display

maintain your address book or some similar list. Since it can stand on its own, I suggest you type *Direct* as a separate program and, when everything works smoothly, merge it with the main program. To make it an independent program, simply drop the first three lines and change the END SUB and EXIT SUB statements to END statements. For test purposes, you may want to include the *SendToModem* subprogram and change the *Dial* subroutine at the very end of the subprogram so that it passes the phone number to *SendToModem,* rather than back to the main program.

We will stumble across a few new concepts in this section of code, but most of it should look pretty familiar to you by now. I will list the program in sections, and comment on new items and problem areas as we come to them.

Initializing the subprogram

The first section, shown in Figure 12-48, contains the standard subprogram statement and a few GOSUBs to initialize the directory display. Notice that there are two SHARED statements: You can use as many as you need to list all the shared variables. They aren't required to be at the beginning of the subprogram, but you will avoid problems if you always put them there.

A second thing to notice in this section is the subroutine labels. I tend to use the same labels for the same functions in different programs. For example, I usually use *InitializeVariables* for the subroutine that does what the first one here does. However, in this case doing so would result in a duplicate-label error, since that label is already used in the main program. This doesn't seem to be quite in keeping with the theory of

```
'*
'* Subprogram Direct called from Directory. Display telephone
'* directory and allow user to edit directory or select number to dial.
'*
SUB Direct STATIC
  SHARED phoNum$, phoNam$, dialFlag%, scrnsave%(), directFile$
  SHARED num$(), who$(), cr$, false, true
  GOSUB DefineVariables
  GOSUB ReadData
  GOSUB ShowWindow
```

Figure 12-48. Initializing the subprogram

unique variable names that makes subprograms portable between programs, but there are exceptions to every rule, and there is probably a good reason for this one.

Defining the variables

The *DefineVariables* subroutine (Figure 12-49) is included primarily to keep the format consistent, since most variables pertinent to this section were already defined in the main program and are included in the SHARED statements. One thing is worth pointing out, though: You do need to define all variables as integers again, since the DEFINT statement in the main program does not apply here. The variable *yDist* is used here to establish the vertical distance between lines in the directory display.

Reading data

The *ReadData* subroutine shown in Figure 12-50 attempts to open a disk file containing the names and numbers to be displayed. If it has a problem doing so, it branches to an error-handling routine that should take care of the problem. Once the file is opened, the names and numbers are read and assigned to the elements of the arrays *who$* and *num$*.

I set the maximum number of entries to be read at 10. My original version of this directory allowed you to page through unlimited entries, but I decided I was adding complexity and length to the program without really explaining anything new. (A simple method of storing more numbers is to use multiple directory files and add a FILES$(1) statement to allow you to choose one of them.)

```
'*
'* Define variables (most are shared with main program).
'*
DefineVariables:
   DEFINT a - z
   yDist = 12                                   'vertical distance between entries
   RETURN
```

Figure 12-49. The *DefineVariables* subroutine

```
'*
'* Open directory file and assign entries to who$() and num$().
'*
ReadData:
  ON ERROR GOTO NoFileError                          'enable error trapping
  OPEN directFile$ FOR INPUT AS #4
  IF errQuitFlag THEN EXIT SUB
  ON ERROR GOTO 0                                    'disable error trapping
  entry = 0
  WHILE (entry < 11) AND (NOT EOF(4))
    entry = entry + 1                                  'number of entries
    INPUT #4, who$(entry), num$(entry)
  WEND
  last = entry                                                'last entry
  CLOSE #4                                          'we're through with it
  RETURN
```

Figure 12-50. The *ReadData* subroutine

The ON ERROR GOTO statement at the beginning of this section puts error trapping into effect, instructing the program to branch to *NoFileError* should any error occur later in the program. Of course, since we are about to open a file, we suspect that if there is an error now it will be a File Not Found error (number 53).

Let's assume for the moment that the file is there and that we have opened it. We will bypass *errQuitFlag*, which is set by the error-handling routine. The *ON ERROR GOTO 0* statement turns error trapping off, so that the program won't branch to the *NoFileError* routine if it later encounters an error.

The next line initializes the variable *entry* to 0. This variable is used to keep track of the line occupied by each name and phone number on the display. It would be unnecessary to initialize *entry* to 0 if we ran this subprogram only once, since all numeric variables are initially set to 0 unless we specify otherwise. However, when a subprogram is rerun, all variables that are not shared with the main program have exactly the value they had when the subprogram last ended (assuming that you have not stopped running the main program between calls), so we need to take the precaution of re-initializing the variable here.

Next, a WHILE...WEND loop repeats until the number of entries exceeds 10 or the end of the file is reached. The test for the number of entries is included to avoid an

error (Subscript Out of Range, number 9) if the user opens a file containing too many entries—perhaps one created by another program. After the WHILE...WEND loop, we again test the number of entries, this time to see if the file we opened was empty. If it was, *who$(1)* is assigned the string *"empty directory"*, so that the user will at least know the file was opened and checked. The last value of *entry* (which is incremented by 1 each time through the loop) is assigned to the variable *last*, which is used to determine how many edit fields to display.

The *NoFileError* subroutine

If the directory file was not on the startup disk, the program branches to the *NoFileError* routine shown here in Figure 12-51. This routine makes sure the error trapped was number 53 (File Not Found) and then gives the user a choice of solutions (Figure 12-52). If it was some other error, then the *ON ERROR GOTO 0* statement stops the program and prints the error number. However, by placing the error trap where we did, it is unlikely we will trap any other error.

```
'*
'* Attempt made to open directory file that wasn't on startup disk.
'*
NoFileError:

   '*
   '* Crash if error other than File Not Found was trapped.
   '*
   IF ERR <> 53 THEN ON ERROR GOTO 0
   BEEP                                              'get some attention
   SaveScreen
   WINDOW 3, , (50, 50) - (375, 185), -2
   PRINT "The file containing the telephone directory"
   PRINT "is not on the startup disk"
   BUTTON 1, 1, "Load from another disk", (20, 50) - (200, 70), 2
   BUTTON 2, 1, "Use default settings", (20, 80) - (200, 100), 2
   BUTTON 3, 1, "Cancel", (250, 95) - (310, 120)
   WHILE DIALOG(0) <> 1
   WEND                                              'wait for some action
   butPush = DIALOG(1)                               'which button was clicked
```

Figure 12-51. The *NoFileError* subroutine

more...

```
    IF butPush = 3 THEN WINDOW CLOSE 3 : RestoreScreen
    IF butPush = 3 THEN errQuitFlag = true : RESUME NEXT
    IF butPush = 2 THEN GOTO DefaultDirectory
    WINDOW CLOSE 3
    RestoreScreen
    directFile$ = FILES$(1, "TEXT")                    'open file dialog box
    IF directFile$ = "" THEN GOTO NoFileError          'no selection
    OPEN directFile$ FOR INPUT AS #4
    RESUME NEXT

DefaultDirectory:
    OPEN directFile$ FOR OUTPUT AS #4                  'create new file
    WRITE #4, "New Directory", "number"                'store something in it
    CLOSE #4                                           'close it
    WINDOW CLOSE 3
    RestoreScreen
    RESUME
```

Figure 12-51. The *NoFileError* subroutine *(continued)*

One very important point you should note about error trapping from within a subprogram is that the subroutine branched to when the error is encountered must be located outside the subprogram. If you try to branch to an error-handling subroutine inside the subprogram, you will be greeted by an Undefined Label error. (We will discover one more little quirk when we get ready to return to the subprogram from this error routine.)

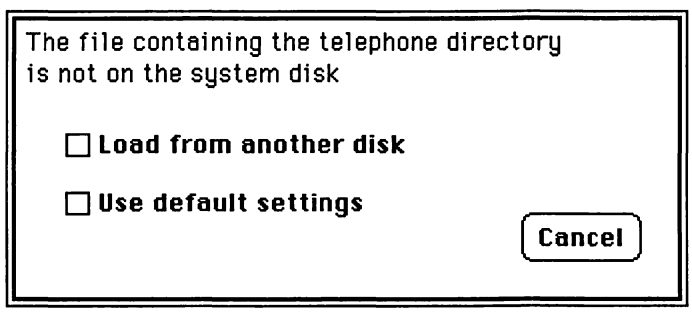


Figure 12-52. The *NoFileError* dialog box

If the user decides to cancel the directory request, the program has to do a little fancy footwork to get you back to the main program. Normally, a RESUME statement with one of these four syntaxes is used to exit the error-handling routine:

Syntax	Action
RESUME	Execution resumes at statement that caused error
RESUME 0	Same as RESUME
RESUME NEXT	Execution resumes at statement immediately following one that caused error
RESUME *line*	Execution resumes at *line*

The one additional quirk I referred to earlier is that you cannot use the RESUME statement to return to a specific label in a subprogram because BASIC, at this point, decides not to recognize subprogram labels. You also can't just jump back into the main program, as that would leave an unENDed subprogram hanging around. The solution is to set a flag that tells the subprogram you want to go back to the main program, then RESUME with the next statement in the subprogram, which should be a test of that flag. So in this program, if the user chooses to cancel the directory request by clicking the Cancel button in the error dialog box, we set *errQuitFlag* to *true* and issue a RESUME NEXT statement.

If the user decides to use the default directory, the subprogram opens a new directory file, *directFile$*, and stores the strings *"New directory"* and *"number"* in it. The RESUME statement returns the program to the OPEN statement in the subprogram, which can now re-open this new directory file and retrieve the information that was just stored.

If the user decides to use a different directory, the subprogram displays an OPEN dialog box, opens the requested file, and then resumes with the next line in the subprogram.

Displaying data

The *ShowWindow* subroutine, shown in Figure 12-53, opens a window, creates an edit field for each name and phone number read from the directory file, and displays the option buttons.

```
'*
'* Open directory display window.
'*
ShowWindow:
  WINDOW 3, "Phone Directory", (50, 35) - (367, 200), -2
  LINE (15, 5) - (305, 140), , b                                    'inside box
  TEXTMODE 1                                                        'keep selection dot clean
  TEXTSIZE 9                                                        'keep dialog box small

  '*
  '* List names and numbers.
  '*
  FOR entry = 1 TO last
    yPos = entry * yDist                                            'to shorten following lines

    '*
    '* Create name and number edit fields.
    '*
    EDIT FIELD 2 * entry - 1, who$(entry), (20, 2 + yPos) - (155, 15 + yPos), 3
    EDIT FIELD 2 * entry, num$(entry), (170, 2 + yPos) - (280, 15 + yPos), 3, 3
  NEXT
  edField = 1 : EDIT FIELD edField                                  'make first edit field active
  GOSUB PutDot                                                      'show which entry is active

  '*
  '* Create command buttons at bottom.
  '*
  BUTTON 1, 1, "Cancel", (15, 145) - (75, 160)
  BUTTON 2, 1, "Add", (91, 145) - (151, 160)
  BUTTON 3, 1, "Done", (169, 145) - (229, 160)
  BUTTON 4, 1, "Dial", (245, 145) - (305, 160)
  RETURN
```

Figure 12-53. The *ShowWindow* subroutine

This method of creating a number of edit fields is similar to that used to create buttons in the configuration dialog box in the last chapter. In this case, the position of each edit field is expressed relative to the entry number and the variable *yDist*. The edit-field ID numbers are also expressed relative to the entry, but since there are two fields per entry line, we have to do a little manipulation here (I think anyone who has written more than two or three programs starts to develop a knack for expressing a variable in terms of one or more other variables).

```
'*
'* Place dot at right end of currently selected entry.
'*
PutDot:
   entry = (edField + 1) \ 2
   LINE (290, 16) - (304, 139), 30, bf                    'white box erases old dot
   MOVETO 290, 12 + entry * yDist                          'active entry
   PRINT CHR$(165);                                        'print dot
   RETURN
```

Figure 12-54. The *PutDot* subroutine

Notice that the edit fields for both name and number are type 3, which means that there will be no box drawn around them. It also means that Return keys are not allowed, though in this case I simply ignore Returns, rather than trapping them and taking some action, because it does not seem to me that any particular action is intuitively correct in this situation (unlike many cases, where Return would signal the end of an edit action and therefore a desire to return to the main program).

Next, the subroutine *PutDot,* shown in Figure 12-54, places a dot to the right of the line that contains the currently selected edit field. As far as the program is concerned, the entire line is selected and clicking the Dial button will dial its number.

PutDot determines the selected line by evaluating the current edit field, which is always assigned to *edField.* It then draws a filled white box over the area where the dots are displayed, to erase the current dot. (We can get away with this tactic because speed isn't really important at this point. If it were, we would use a ROM call to XOR the dot into oblivion.) The new dot is produced by printing CHR$(165) to the right of the current line. You can substitute any other ASCII code you like for 165: perhaps CHR$(60) + CHR$(45) to produce this <− arrow, or CHR$(199) to indicate the selected line with this << chevron.

Direct's main loop

The *MainLoop* routine (Figure 12-55) is a forever-loop (*WHILE true...WEND*) that the program falls into after the initialization subroutines. The purpose of this loop is to branch to an appropriate subroutine if a command button or edit field is

```
'*
'* Trap button and edit-field actions, and branch to appropriate subroutine.
'*
MainLoop:
   WHILE true
      eventTrapped = DIALOG(0)
      IF eventTrapped = 1 THEN GOSUB ButPush                    'button clicked
      IF eventTrapped = 2 THEN GOSUB EdFldClk                   'edit field clicked
      IF eventTrapped = 7 THEN GOSUB EdFldTb                    'Tab key pressed
   WEND
```

Figure 12-55. The *MainLoop* routine

clicked or a Tab key pressed, as indicated by the value returned when the DIALOG(0) function is used.

Tabbing

Pressing the Tab key while one edit field is active typically (in Macintosh applications) deactivates that field and makes the next available edit field active. In our program the *EdFldTb* subroutine (Figure 12-56) performs this task. Since we plan to move from one edit field to another, the contents of the current field are first checked for change. If the program is not already in the last edit-field position, *EdFldTb* advances it one position; if it is, then the subroutine moves it to the first edit-field position, so you can cycle through the entries again.

```
'*
'* Tab key was pressed; move to next edit field.
'*
EdFldTb:
   GOSUB CheckEdit
   IF edField < 2 * last THEN edField = edField + 1 ELSE edField = 1
   EDIT FIELD edField                                           'make it active
   GOSUB PutDot
   RETURN
```

Figure 12-56. The *EdFldTb* subroutine

```
'*
'* Inactive edit field was clicked; determine which one and make it active.
'*
EdFldClk:
   GOSUB CheckEdit                                    'see if current field has changed
   edField = DIALOG(2) : EDIT FIELD edField           'make new field active
   GOSUB PutDot
   RETURN
```

Figure 12-57. The *EdFldClk* subroutine

Clicking an edit field

The *EdFldClk* subroutine, shown in Figure 12-57, assigns the value returned by DIALOG(2) to the variable *edField,* then makes *edField* the current edit field. This allows the user to select an edit field by clicking in it.

Clicking a button

The *ButPush* subroutine, shown in Figure 12-58, assigns the value returned by DIALOG(1)—the number of the button clicked—to the variable *buttonPushed.* It then tests *buttonPushed* and takes the action appropriate to its value.

If Button #1 (Cancel) is clicked, *changeFlag* is set to *false,* meaning that the directory file won't be updated, and the program branches to the closing routine (*Fini*).

```
'*
'* Button was clicked; determine which one and take action.
'*
ButPush:
   buttonPushed = DIALOG(1)
   IF buttonPushed = 1 THEN changeFlag = false : GOTO Fini          'Cancel clicked
   IF buttonPushed = 2 THEN GOSUB CheckEdit : GOSUB AddEntry        'Add clicked
   IF buttonPushed = 3 THEN GOSUB CheckEdit : GOTO Fini             'Done clicked
   IF buttonPushed = 4 THEN GOSUB CheckEdit : GOTO Dial             'Dial clicked
   RETURN
```

Figure 12-58. The *ButPush* subroutine

If Button #2, #3, or #4 is clicked, the *ButPush* subroutine passes the program to the *CheckEdit* subroutine, which checks the contents of the current edit field to see if they have changed.

Checking for changes

The *CheckEdit* subroutine, shown in Figure 12-59, first determines the number of the active line, then determines whether the active edit field is the first (name) or second (number) field on that line. If it is the first field, the program branches to the line labeled *CheckName;* otherwise, it continues its normal flow. Either way, the original value of that edit field is compared to the current value. If they aren't the same, *changeFlag* is set and the new value is assigned to the appropriate element of either the *who$* array or the *num$* array.

Upon returning from *CheckEdit,* the program goes to *AddEntry* if the Add button was clicked, to *Fini* if the Done button was clicked, or to *Dial* if the Dial button was clicked.

Adding new entries

The *AddEntry* subroutine, shown in Figure 12-60 on the next page, adds a new line to the list, if there is room for it, and puts two edit fields on the line.

```
'*
'* Check if any changes have been made to edit field before
'* quitting, dialing, or moving to another edit field.
'*
CheckEdit:
   entry = (edField + 1) \ 2
   IF (edField MOD 2 = 1) THEN CheckNam
   IF num$(entry) <> EDIT$(edField) THEN changeFlag = true
   IF num$(entry) <> EDIT$(edField) THEN num$(entry) = EDIT$(edField)
   RETURN
CheckNam:
   IF who$(entry) <> EDIT$(edField) THEN changeFlag = true
   IF who$(entry) <> EDIT$(edField) THEN who$(entry) = EDIT$(edField)
   RETURN
```

Figure 12-59. The *CheckEdit* subroutine

```
'*
'* Activate new edit field below last one.
'*
AddEntry:
   changeFlag = true                                    'update file when quitting
   last = last + 1
   IF last = 10 THEN BUTTON 2, 0

   '*
   '* Create right edit field first, so left one ends up active.
   '*
   EDIT FIELD 2 * last, "number", (170, 2 + last * yDist) - (280, 15 + last * yDist), 3, 3
   EDIT FIELD 2 * last - 1, "name", (20, 2 + last * yDist) - (155, 15 + last * yDist), 3
   edField = 2 * last - 1
   num$(last) = "number"                                'assign temporary values
   who$(last) = "name"
   GOSUB PutDot
   RETURN
```

Figure 12-60. The *AddEntry* subroutine

Since something is being added to the directory, *changeFlag* is set to *true,* so that the disk file will be updated when the subprogram ends, and the value of *last* is incremented and then tested. If this brings the number of entries to 10, the Add button is disabled. The edit fields, with the default entries of *"name"* and *"number"*, are created in the same way as in the *ShowWindow* subroutine, except that the edit field on the right is created first, in order to leave the one on the left active without having to specifically set it. As usual, the dot is displayed to indicate this is the active line.

Finishing up

The *Fini* routine (Figure 12-61) checks *changeFlag* to see if any changes have been made. If not, the window is closed and the EXIT SUB statement issued to leave the subprogram and return to the main program. If changes have been made, the directory file is opened and the elements of the *who$* and *num$* arrays are written to it. During this process, any lines that have had the contents of both fields deleted are deleted from the file, so if there is a blank line when you quit, it won't be there when you next look at the directory. After storing the entries, the file is closed, the window is closed, and the subprogram is ended.

```
'*
'* Return to main program.
'*
Fini:
   IF changeFlag <> true THEN WINDOW CLOSE 3 : EXIT SUB          'no changes
   OPEN directFile$ FOR OUTPUT AS #4
   FOR entry = 1 TO last
      IF who$(entry) = "" AND num$(entry) = "" THEN PrintSkip    'compress
      WRITE #4, who$(entry); num$(entry)
PrintSkip:
   NEXT
   CLOSE #4                                                      'close file
   WINDOW CLOSE 3
END SUB
```

Figure 12-61. The *Fini* routine

Dialing a number

If the Dial button is clicked, the program goes to the *Dial* routine (see Figure 12-62), which is located just above the *Fini* routine, so that after flowing through *Dial* the program automatically goes to *Fini*.

Dial assigns the current values of *who$* and *num$* to the variables *phoNam$* and *phoNum$*, which the main program will use, and sets *dialFlag* to *true* to tell the main program to dial a number.

That pretty well covers the communication program. In modifying the Terminal program supplied on your BASIC disk, I added the features that *I* wanted. By now, you should understand both the initial program and its modifications well enough to add

```
'*
'* Dial button was pressed; get number.
'*
Dial:
   phoNam$ = who$(entry)                          'who are we going to call
   phoNum$ = num$(entry)                          'their phone number
   dialFlag = true                                'tell main program
```

Figure 12-62. The *Dial* routine

any features that are important to *you*. You may want to add a routine to change one character to another or a routine to automatically send your password and the logon sequence for an online service. The beauty of BASIC is that you are not limited to the features someone else feels you should have in a program.

If you have not previously used a computer to sample the online services available and are about to do so, you have a treat in store. You will find a fascinating variety of information and services at your fingertips.

```
'* Enhanced Terminal Emulation Program
'*

'*
'* Execute all initializing subroutines.
'*
   GOSUB InitializeVariables
   GOSUB CreateWindow
   GOSUB SetDefaults
   GOSUB OpenComm
   GOSUB DefineMenu

'*
'* Fall through to main communication loop after completing all initializing
'* subroutines and remain there until selection is made from menu.
'*

'*
'* Display characters from COM1:, send keystrokes to COM1:.
'*
CommLoop:
   WHILE true
      WHILE pauseFlag : WEND
      IF (LOC(1) = 0) AND stopFlag THEN PRINT #1, xon$; : stopFlag = false
      WHILE LOC(1) = 0                                    'nothing waiting to come in
         SendKey                                          'send key typed to file #1
         MENU STOP                                        'don't get sidetracked
         IF (sendFlag OR viewFlag) AND NOT waitFlag THEN CALL SendLine
         IF endViewFlag THEN GOSUB EndFile
         MENU ON                                          'see if anything selected
      WEND
```

Figure 12-63. The complete enhanced communication program

more...

```
    IF LOC(1) > bufferLimit THEN PRINT #1, xoff$; : stopFlag = true
    lin$ = INPUT$(LOC(1), 1)                              'get everything waiting
    RemoveChars lin$, lf$                                      'strip linefeeds
    RemoveChars lin$, xon$
    IF sendFlag AND waitFlag AND removeFlag THEN waitFlag = false
    RemoveChars lin$, xoff$
    IF sendFlag AND removeFlag THEN waitFlag = true
    MENU STOP
    PrintString lin$
    MENU ON
    IF endSendFlag THEN GOSUB EndFile

    '*
    '* File #3 is automatically named file that stores input when
    '* Start or Continue is chosen from Save menu. File #2
    '* stores received file after asking for name to store it under.
    '*
    IF saveFlag THEN PRINT #3, lin$;
    IF receiveFlag THEN PRINT #2, lin$;
  WEND

'*
'* Dimension arrays and initialize variables.
'*
InitializeVariables:
  DEFINT a - z                                                  'for speed
  DIM choice(4)                                        'selected parameters
  DIM choice$(4)                                                'likewise
  DIM group(17)                                       'parameter buttons
  DIM nam$(17)                                                  'likewise
  DIM num$(10)                                      'directory phone number
  DIM scrnsave(4889)                                 'used to restore screen
  DIM who$(10)                                            'directory name

  alert$ = CHR$(7)                                                'beep
  buffer = 2000                                                'input buffer
  bufferLimit = .9 * buffer                          'buffer limit--send XOFF
  choice(1) = 4                                                '1200 baud
  choice(2) = 9                                                'even parity
  choice(3) = 13                                               '7 data bits
  choice(4) = 16                                               '2 stop bits
  cr$ = CHR$(13)                                             'carriage return
  Dial$ = "ATDT"                                   'D.C. Hayes dial command
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
    directFile$ = "directory file"                          'file containing phone #'s
    endMessage$ = sp$ + cr$ + lf$ + alert$ + "End of Transmission" + cr$
    false = 0
    hangup$ = "~+++~ATH0"                                   'hangup command
    lf$ = CHR$(10)                                          'linefeed
    pauseFlag = false                                       'activity paused
    posit$ = cr$ + "  "
    receiveFlag = false                                     'Receive File selected
    saveFlag = false                                        'save flag
    sendFlag = false                                        'Send File selected
    sp$ = CHR$(32)
    stopFlag = false                                        'XOFF sent flag
    true = -1
    viewFlag = false                                        'View file selected
    waitFlag = false                                        'XOFF received flag
    xoff$ = CHR$(19)                                        'stop sending
    xon$ = CHR$(17)                                         'send more
    RETURN

'*
'* Create output window.
'*
CreateWindow:
    WINDOW 1, , (3, 20) - (511, 338), 3
    PENMODE 10
    TEXTFONT 4                                              'monospace font
    TEXTSIZE 9                                              '80 characters per line
    RestoreCur
    RETURN

'*
'* Set up default options.
'*
'* choice(1) is baud rate: 1  --  110
'*                         2  --  300
'*                         3  --  600
'*                         4  -- 1200
'*                         5  -- 2400
'*                         6  -- 4800
'*                         7  -- 9600
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* choice(2) is parity:     8 -- None
'*                          9 -- Even
'*                         10 -- Odd
'*
'* choice(3) is data bits: 11 -- 5
'*                         12 -- 6
'*                         13 -- 7
'*                         14 -- 8
'*
'* choice(4) is stop bits: 15 -- 1
'*                         16 -- 2
'*


'*
'* Changing default choices here will control initial parameters.
'*
SetDefaults:
   FOR count = 1 TO 16
      READ garbage, garbage, garbage, nam$(count)                    'discard garbage
   NEXT
   choice$(1) = STR$(VAL(nam$(choice(1))))
   choice$(2) = LEFT$(nam$(choice(2)), 1)
   choice$(3) = STR$(VAL(nam$(choice(3))))
   choice$(4) = STR$(VAL(nam$(choice(4))))
   options$ = choice$(1) + "," + choice$(2) + "," + choice$(3) + "," + choice$(4)
   RETURN

'*
'* Open communications port.
'*
OpenComm:
   OPEN "COM1:" + options$ AS 1 LEN = buffer

'*
'* Clear input buffer.
'*
Flush:
   garbage$ = INPUT$ (LOC(1), 1)
   RETURN
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* Set up custom menu.
'*
DefineMenu:
   MENU 1, 0, 1, "File"
   MENU 1, 1, 1, "View File"
   MENU 1, 2, 1, "Receive File"
   MENU 1, 3, 1, "Send File"
   MENU 1, 4, 1, "Quit to BASIC"
   MENU 1, 5, 1, "Quit to Desktop"
   MENU 2, 0, 1, "Options"
   MENU 2, 1, 1, "Pause"
   MENU 2, 3, 1, "Set Configuration Parameters"
   MENU 3, 0, 1, "Save"
   MENU 3, 1, 1, "Start"
   MENU 3, 2, 0, "Continue"
   MENU 3, 3, 0, "Stop"
   MENU 4, 0, 1, "Phone"
   MENU 4, 1, 1, "Directory"
   MENU 4, 2, 1, "Enter Number"
   MENU 4, 3, 1, "Hang Up"
   MENU 4, 4, 0, "Redial"
   MENU 5, 0, 0, ""
   ON MENU GOSUB HandleMenu                    'if menu item chosen
   MENU ON                                      'activate trapping
   RETURN

'*
'* Decide which menu item selected and take action.
'*
HandleMenu:
   MenuSel = MENU(0)                           'get number of menu
   ItemSel = MENU(1)                           'get number of item

   '*
   '* Go to appropriate subroutine for menu selected.
   '*
   ON MenuSel GOSUB FileMenu, OptionMenu, SaveMenu, PhoneMenu
   MENU
   RETURN
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* File menu was selected.
'*
FileMenu:
   ON ItemSel GOSUB ViewFile, ReceiveFile, SendFile, DoneBas, DoneDesk
   RETURN

'*
'* A(1): View file before sending or after receiving.
'*
ViewFile:
   IF startFlag = true THEN GOTO EndFile                    'terminating ViewFile
   SaveScreen
   filename$ = FILES$(1, "TEXT")
   RestoreScreen
   IF filename$ = "" THEN RETURN
   MENU 1, 2, 0                                          'disable Receive File
   MENU 1, 3, 0                                      'put check mark by Send File
   MENU 1, 1, 2                                      'put check mark by View File
   MENU 2, 0, 0                                          'disable Options menu
   MENU 3, 0, 0                                            'disable Save menu
   MENU 4, 0, 0                                          'disable Phone menu
   OPEN filename$ FOR INPUT AS #2
   oldSaveFlag = saveFlag                                 'store state of save flag
   saveFlag = false                                      'if previously saving, stop
   viewFlag = true
   startFlag = true
   RETURN

'*
'* A(2): Transfer information received at COM1: to disk file.
'*
ReceiveFile:
   IF startFlag = true THEN EndFile                       'terminating ReceiveFile
   SaveScreen
   filename$ = FILES$(0, "Name to save file under")
   RestoreScreen
   IF filename$ = "" THEN RETURN
   MENU 1, 3, 0                                            'disable Send File
   MENU 1, 2, 2                                      'put check mark by Receive File
   MENU 1, 1, 0                                            'disable View File
   MENU 2, 0, 0                                          'disable Options menu
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
   MENU 3, 0, 0                                    'disable Save menu
   MENU 4, 0, 0                                    'disable Phone menu
   OPEN "O", #2, filename$
   oldSaveFlag = saveFlag
   saveFlag = false
   receiveFlag = true                              'turn on receiving
   startFlag = true
   RETURN

'*
'* A(3): Transmit file stored on disk.
'*
SendFile:
   IF startFlag = true THEN EndFile                'terminating SendFile
   SaveScreen
   filename$ = FILES$(1, "TEXT")
   RestoreScreen
   IF filename$ = "" THEN RETURN
   MENU 1, 2, 0                                    'disable Receive File
   MENU 1, 3, 2                                    'put check mark by Send File
   MENU 1, 1, 0                                    'disable View File
   MENU 2, 0, 0                                    'disable Options menu
   MENU 3, 0, 0                                    'disable Save menu
   MENU 4, 0, 0                                    'disable Phone menu
   OPEN "I", #2, filename$
   oldSaveFlag = saveFlag
   saveFlag = false
   sendFlag = true                                 'turn on sending
   startFlag = true
   RETURN

'*
'* Close file and re-enable various competing menus
'*
EndFile:
   CLOSE #2
   ShowCur
   PrintString endMessage$
   MENU 1, 2, 1                                    'enable Receive File option
   MENU 1, 3, 1                                    'enable Send File option
   MENU 1, 1, 1                                    'remove check mark
   MENU 2, 0, 1                                    'enable Options menu
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
   MENU 3, 0, 1                                          'enable Save menu
   MENU 4, 0, 1                                          'enable Phone menu
   saveFlag = oldSaveFlag                                'restore state of Save
   viewFlag = false                                      'turn off viewing
   sendFlag = false
   receiveFlag = false
   startFlag = false
   endFlag = false
   endSendFlag = false
   endViewFlag = false
   RETURN

'*
'* A(4): Return to BASIC.
'*
DoneBas:
   MENU RESET                                            'restore BASIC menu
   CLOSE                                                 'close all open files
   END                                                   'return to BASIC

'*
'* A(5): Return to Macintosh desktop.
'*
DoneDesk:
   CLOSE
   SYSTEM

'*
'* Options Menu was selected.
'*
OptionMenu:
   ON ItemSel GOSUB Pause, Strip, Config
   RETURN

'*
'* B(1): Stop processing of data.
'*
Pause:
   pauseFlag = pauseFlag XOR -1
   MENU 2, 1, -1 * pauseFlag + 1
   IF pauseFlag THEN PRINT #1, xon$; ELSE PRINT #1, xoff$;
   RETURN
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* Set communication parameters.
'*
Config:
   CLOSE 1                                                    'close COM1:
   SaveScreen
   WINDOW 2, , (50, 35) - (450, 185), 2                       'open new window
   GOSUB DisplayDefaults                                      'show parameters
   GOSUB SelectOptions                                        'get selection
   options$ = choice$(1) + "," + choice$(2) + "," + choice$(3) + "," + choice$(4)

   '*
   '* Open communications port.
   '*
   OPEN "COM1:" + options$ AS 1 LEN = buffer
   WINDOW CLOSE 2
   RestoreScreen
   RETURN

DisplayDefaults:
   RESTORE                                    'to read data statements from start
   FOR count = 1 TO 16
      READ x, y, group(count), nam$(count)
      BUTTON count, 1, nam$(count), (x, y) - (x + 130, y + 15), 3
   NEXT count
   BUTTON 17, 1, "OK", (320, 110) - (380, 135)

   '*
   '* Simulate button pushes to highlight defaults.
   '*
   FOR count = 1 TO 4
      ButtonSelect choice(count)
   NEXT
   RETURN

'*
'* Here is x,y coordinate of button, groupNum, title.
'*
   DATA 10, 10, 1, 110 bits per sec
   DATA 10, 30, 1, 300 bits per sec
   DATA 10, 50, 1, 600 bits per sec
   DATA 10, 70, 1, 1200 bits per sec
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
   DATA 10, 90, 1, 2400 bits per sec
   DATA 10, 110, 1, 4800 bits per sec
   DATA 10, 130, 1, 9600 bits per sec

   DATA 292, 10, 2, No parity
   DATA 292, 30, 2, Even parity
   DATA 292, 50, 2, Odd parity

   DATA 162, 10, 3, 5 Data bits
   DATA 162, 30, 3, 6 Data bits
   DATA 162, 50, 3, 7 Data bits
   DATA 162, 70, 3, 8 Data bits

   DATA 162, 110, 4, 1 Stop bits
   DATA 162, 130, 4, 2 Stop bits

SelectOptions:
   eventTrapped = DIALOG(0)                          'dialog event
   IF eventTrapped <> 1 THEN SelectOptions           'watch for button press
   buttonPushed = DIALOG(1)                          'which button

   '*
   '* Get button number if through making selections (clicked OK).
   '*
   IF buttonPushed < 17 THEN ButtonSelect buttonPushed : GOTO SelectOptions
   RETURN

'*
'* Save Menu was selected.
'*
SaveMenu:
   ON ItemSel GOSUB StartSave, ContSave, StopSave
   RETURN

'*
'* C(1): Start saving.
'*
StartSave:
   tim$ = TIME$                                      'get current time for filename
   FOR count = 3 TO 6 STEP 3                         'change : to /
      MID$(tim$, count) = "/"
   NEXT count
   filename$ = "Saved " + tim$                       'assign filename
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
   OPEN "A", #3, filename$                          'open new file
   saveFlag = true                                  'set save flag to true
   MENU 3, 1, 0                                     'disable Start
   MENU 3, 3, 1                                     'enable Stop
   RETURN

'*
'*C(2): Continue saving (if stopped).
'*
ContSave:
   saveFlag = true
   MENU 3, 2, 0                                     'disable Continue
   MENU 3, 3, 1                                     'enable Stop
   RETURN

'*
'*C(3): Stop saving.
'*
StopSave:
   saveFlag = false                                 'set save flag to false
   MENU 3, 2, 1                                     'enable Continue
   MENU 3, 3, 0                                     'disable Stop
   RETURN

'*
'* Phone Menu was selected.
'*
PhoneMenu:
   ON ItemSel GOSUB Directory, EnterNumber, Disconnect, Redial
   RETURN

'*
'* D(1): Call Direct subprogram.
'*
Directory:
   SaveScreen
   dialFlag = false                         'set dialFlag going into subprogram
   Direct                                   'call directory subprogram
   RestoreScreen
   IF NOT dialFlag THEN RETURN              'check dialFlag on return
   PrintString "Dialing " + phoNam$ + cr$   'tell us who we're calling
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
   num$ = Dial$ + phoNum$                          'add modem dial command
   SendToModem num$                                'send dial command and number
   RETURN

'*
'* D(2): Create dialog box.
'*
EnterNumber:
   SaveScreen
   WINDOW 4, , (100, 100) - (370, 185), -4
   MOVETO 20, 27
   PRINT "enter phone number"
   EDIT FIELD 1, phoNum$, (160, 15) - (260, 30)    'create edit field
   BUTTON 1, 1, "Cancel", (20, 45) - (80, 70)
   BUTTON 2, 1, "OK", (190, 45) - (250, 70)

'*
'* Wait for user to finish entering number.
'*
EnterLoop:
   eventTrapped = DIALOG(0)
   IF eventTrapped = 6 THEN GOTO EnterContinue     'Return key pressed
   IF eventTrapped <> 1 THEN GOTO EnterLoop        'button wasn't selected
   buttonSel = DIALOG(1)
   IF buttonSel <> 1 THEN GOTO EnterContinue
   WINDOW CLOSE 4                                  'Cancel button clicked
   RestoreScreen
   RETURN

'*
'* Retrieve number from dialog box.
'*
EnterContinue:
   phoNum$ = EDIT$(1)                              'retrieve number
   phoNam$ = "manually entered number"
   WINDOW CLOSE 4
   RestoreScreen
   num$ = Dial$ + phoNum$
   SendToModem num$                                'dial number
   MENU 4, 4, 1                                    'enable Redial
   RETURN
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* D(3): Tell modem to hang up.
'*
Disconnect:
   SendToModem hangup$
   RETURN

'*
'* D(4): Send number last dialed to modem.
'*
Redial:
   PRINT "Dialing "; phoNam$; ": "                    'tell us who we're calling
   num$ = Dial$ + phoNum$
   SendToModem num$
   RETURN

'*
'* Subprogram Section
'*

'*
'* Subprogram ButtonSelect(buttonPushed) is called from DisplayDefaults and
'* SelectOptions subroutines. User has just pushed button.  Highlight
'* that button and remember selection in choice() and choice$().
'*
SUB ButtonSelect(buttonPushed) STATIC
   SHARED nam$(), group(), choice(), choice$()
   groupNum = group(buttonPushed)                     'group 1, 2, 3, or 4
   BUTTON choice(groupNum), 1                         'reset old selection
   BUTTON buttonPushed, 2                             'set new selection
   choice(groupNum) = buttonPushed                        '1 through 16
   IF groupNum = 2 THEN SetParity                     'from parity group
   choice$(groupNum) = STR$(VAL(nam$(buttonPushed)))  'from other group
   EXIT SUB
SetParity:
   choice$(groupNum) = LEFT$(nam$(buttonPushed), 1)       'get first letter
END SUB
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* Delay.
'*
SUB Delay(count) STATIC
   FOR hold = 1 TO count
   NEXT
END SUB

'*
'* Subprogram Direct called from Directory. Display telephone
'* directory and allow user to edit directory or select number to dial.
'*
SUB Direct STATIC
   SHARED phoNum$, phoNam$, dialFlag%, scrnsave%(), directFile$
   SHARED num$(), who$(), cr$, false, true
   GOSUB DefineVariables
   GOSUB ReadData
   GOSUB ShowWindow

'*
'* Print string of text.
'*
SUB PrintString(text$) STATIC
   SHARED curX, curY
   CALL LINE (5, 0)                                    'erase cursor
   MOVETO curX, curY                                   'move back
   PRINT text$;
   ShowCur                                             'show new cursor
END SUB

'*
'* Remove passed character from passed line.
'*
SUB RemoveChars(lin$, char$) STATIC
   SHARED removeFlag
   removeFlag = false                                  'reset flag
   position = INSTR(lin$, char$)                       'where is first offensive character
   IF position = 0 THEN EXIT SUB                       'there wasn't one
   removeFlag = true                                   'there was one
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
  WHILE position > 0                                    'remove first and check for more
    lin$ = LEFT$(lin$, position - 1) + RIGHT$(lin$, LEN(lin$) - position)
    position = INSTR(lin$, char$)
  WEND
END SUB

'*
'* Place cursor in upper left corner.
'*
SUB RestoreCur STATIC
  MOVETO 0, 10
  ShowCur
END SUB

'*
'* Display previously saved screen area after removing dialog box.
'*
SUB RestoreScreen STATIC
  SHARED scrnsave()
  PUT (39, 7), scrnsave(0), PSET
END SUB

'*
'* Save maximum screen area used by any dialog box, before displaying dialog box.
'*
SUB SaveScreen STATIC
  SHARED scrnsave()
  GET (39, 7) - (455, 187), scrnsave(0)
END SUB

'*
'* Get keystroke from keyboard and send it out COM1:.
'*
SUB SendKey STATIC
  keyTyped$ = INKEY$
  IF keyTyped$ <> "" THEN PRINT #1, keyTyped$;
END SUB

'*
'* View or send a file.
'*
SUB SendLine STATIC
  SHARED viewFlag, endViewFlag
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
    SHARED sendFlag, endSendFlag, true, false
    LINE INPUT #2, lin$                                    'get line from file
    IF sendFlag THEN PRINT #1, lin$ ELSE PRINT lin$        'send it someplace
    IF NOT EOF(2) THEN EXIT SUB
    IF viewFlag THEN endViewFlag = true
    IF sendFlag THEN endSendFlag = true
    viewFlag = false : sendFlag = false
END SUB

'*
'* Send passed string to modem, one character at a time with pause
'* between characters. Used to pass commands and phone numbers to modem.
'*
SUB SendToModem(out$) STATIC
    FOR position = 1 TO LEN(out$)                          'take numbers one by one
        code$ = MID$(out$, position, 1)
        Delay 500                                          'don't send too fast
        IF code$ = "~" THEN Delay 8000 : GOTO SkipCode     'long pause
        PRINT #1, code$;
SkipCode:
    NEXT
    Delay 5000
    PRINT #1, cr$                                          'end with carriage return
END SUB

'*
'* Show cursor at end of current line.
'*
SUB ShowCur STATIC
    SHARED  curX, curY
    curX = WINDOW(4)                          'horizontal location of next character
    curY = WINDOW(5)                                       'vertical location
    CALL LINE (5, 0)                                       'draw cursor
    MOVETO curX, curY                         'put print location back where it was
END SUB

'*
'* Trap button and edit-field actions, and branch to appropriate subroutine.
'*
MainLoop:
    WHILE true
        eventTrapped = DIALOG(0)
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
    IF eventTrapped = 1 THEN GOSUB ButPush                    'button clicked
    IF eventTrapped = 2 THEN GOSUB EdFldClk                   'edit field clicked
    IF eventTrapped = 7 THEN GOSUB EdFldTb                    'Tab key pressed
  WEND

'*
'* Define variables (most are shared with main program).
'*
DefineVariables:
  DEFINT a - z
  yDist = 12                                  'vertical distance between entries
  RETURN

'*
'* Open directory file and assign entries to who$() and num$().
'*
ReadData:
  ON ERROR GOTO NoFileError                              'enable error trapping
  OPEN directFile$ FOR INPUT AS #4
  IF errQuitFlag THEN EXIT SUB
  ON ERROR GOTO 0                                       'disable error trapping
  entry = 0
  WHILE (entry < 11) AND (NOT EOF(4))
    entry = entry + 1                                    'number of entries
    INPUT #4, who$(entry), num$(entry)
  WEND
  last = entry                                                'last entry
  CLOSE #4                                               'we're through with it
  RETURN

'*
'* Open directory display window.
'*
ShowWindow:
  WINDOW 3, "Phone Directory", (50, 35) - (367, 200), -2
  LINE (15, 5) - (305, 140), , b                                'inside box
  TEXTMODE 1                                          'keep selection dot clean
  TEXTSIZE 9                                            'keep dialog box small
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
    '*
    '* List names and numbers.
    '*
    FOR entry = 1 TO last
      yPos = entry * yDist                                    'to shorten following lines

      '*
      '* Create name and number edit fields.
      '*
      EDIT FIELD 2 * entry - 1, who$(entry), (20, 2 + yPos) - (155, 15 + yPos), 3
      EDIT FIELD 2 * entry, num$(entry), (170, 2 + yPos) - (280, 15 + yPos), 3, 3
    NEXT
    edField = 1 : EDIT FIELD edField                          'make first edit field active
    GOSUB PutDot                                              'show which entry is active

    '*
    '* Create command buttons at bottom.
    '*
    BUTTON 1, 1, "Cancel", (15, 145) - (75, 160)
    BUTTON 2, 1, "Add", (91, 145) - (151, 160)
    BUTTON 3, 1, "Done", (169, 145) - (229, 160)
    BUTTON 4, 1, "Dial", (245, 145) - (305, 160)
    RETURN

'*
'* Button was clicked; determine which one and take action.
'*
ButPush:
    buttonPushed = DIALOG(1)
    IF buttonPushed = 1 THEN changeFlag = false : GOTO Fini        'Cancel clicked
    IF buttonPushed = 2 THEN GOSUB CheckEdit : GOSUB AddEntry      'Add clicked
    IF buttonPushed = 3 THEN GOSUB CheckEdit : GOTO Fini           'Done clicked
    IF buttonPushed = 4 THEN GOSUB CheckEdit : GOTO Dial           'Dial clicked
    RETURN

'*
'* Activate new edit field below last one.
'*
AddEntry:
    changeFlag = true                                         'update file when quitting
    last = last + 1
    IF last = 10 THEN BUTTON 2, 0
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
   '*
   '* Create right edit field first, so left one ends up active.
   '*
   EDIT FIELD 2 * last, "number", (170, 2 + last * yDist) - (280, 15 + last * yDist), 3, 3
   EDIT FIELD 2 * last - 1, "name", (20, 2 + last * yDist) - (155, 15 + last * yDist), 3
   edField = 2 * last - 1
   num$(last) = "number"                                    'assign temporary values
   who$(last) = "name"
   GOSUB PutDot
   RETURN

'*
'* Inactive edit field was clicked; determine which one and make it active.
'*
EdFldClk:
   GOSUB CheckEdit                                  'see if current field has changed
   edField = DIALOG(2) : EDIT FIELD edField                  'make new field active
   GOSUB PutDot
   RETURN

'*
'* Tab key was pressed; move to next edit field.
'*
EdFldTb:
   GOSUB CheckEdit
   IF edField < 2 * last THEN edField = edField + 1 ELSE edField = 1
   EDIT FIELD edField                                                'make it active
   GOSUB PutDot
   RETURN

'*
'* Check if any changes have been made to edit field before
'* quitting, dialing, or moving to another edit field.
'*
CheckEdit:
   entry = (edField + 1) \ 2
   IF (edField MOD 2 = 1) THEN CheckNam
   IF num$(entry) <> EDIT$(edField) THEN changeFlag = true
   IF num$(entry) <> EDIT$(edField) THEN num$(entry) = EDIT$(edField)
   RETURN
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
CheckNam:
   IF who$(entry) <> EDIT$(edField) THEN changeFlag = true
   IF who$(entry) <> EDIT$(edField) THEN who$(entry) = EDIT$(edField)
   RETURN

'*
'* Place dot at right end of currently selected entry.
'*
PutDot:
   entry = (edField + 1) \ 2
   LINE (290, 16) - (304, 139), 30, bf                    'white box erases old dot
   MOVETO 290, 12 + entry * yDist                                  'active entry
   PRINT CHR$(165);                                                'print dot
   RETURN

'*
'* Dial button was pressed; get number.
'*
Dial:
   phoNam$ = who$(entry)                                  'who are we going to call
   phoNum$ = num$(entry)                                        'their phone number
   dialFlag = true                                               'tell main program

'*
'* Return to main program.
'*
Fini:
   IF changeFlag <> true THEN WINDOW CLOSE 3 : EXIT SUB                'no changes
   OPEN directFile$ FOR OUTPUT AS #4
   FOR entry = 1 TO last
      IF who$(entry) = "" AND num$(entry) = "" THEN PrintSkip          'compress
      WRITE #4, who$(entry); num$(entry)
PrintSkip:
   NEXT
   CLOSE #4                                                          'close file
   WINDOW CLOSE 3
END SUB
```

Figure 12-63. The complete enhanced communication program *(continued)*

more...

```
'*
'* Attempt made to open directory file that wasn't on startup disk.
'*
NoFileError:

   '*
   '* Crash if error other than File Not Found was trapped.
   '*
   IF ERR <> 53 THEN ON ERROR GOTO 0
   BEEP                                                    'get some attention
   SaveScreen
   WINDOW 3, , (50, 50) - (375, 185), -2
   PRINT "The file containing the telephone directory"
   PRINT "is not on the startup disk"
   BUTTON 1, 1, "Load from another disk", (20, 50) - (200, 70), 2
   BUTTON 2, 1, "Use default settings", (20, 80) - (200, 100), 2
   BUTTON 3, 1, "Cancel", (250, 95) - (310, 120)
   WHILE DIALOG(0) <> 1
   WEND                                                    'wait for some action
   ButPush = DIALOG(1)                                     'which button was clicked
   IF ButPush = 3 THEN WINDOW CLOSE 3 : RestoreScreen
   IF ButPush = 3 THEN errQuitFlag = true : RESUME NEXT
   IF ButPush = 2 THEN GOTO DefaultDirectory
   WINDOW CLOSE 3
   RestoreScreen
   directFile$ = FILES$(1, "TEXT")                         'open file dialog box
   IF directFile$ = "" THEN GOTO NoFileError               'no selection
   OPEN directFile$ FOR INPUT AS #4
   RESUME NEXT

DefaultDirectory:
   OPEN directFile$ FOR OUTPUT AS #4                       'create new file
   WRITE #4, "New Directory", "number"                     'store something in it
   CLOSE #4                                                'close it
   WINDOW CLOSE 3
   RestoreScreen
   RESUME
```

Figure 12-63. The complete enhanced communication program *(continued)*

SECTION IV

Games

IV

Introduction to Games

Chapter 13

All work and no play makes Jack, and the rest of us, dull. I'll admit that work on the Macintosh often *seems* like play, but there comes a time to stop thinking of the computer as a tool and let it entertain you. So let's work at playing for a while.

Practically any computer, from a pocket calculator to a mainframe, can be programmed to play games of some sort, a fact that can be attested to by many a midnight marauder who has used the company computer to defend the solar system against enemy invaders.

The features of the Macintosh that make it particularly enjoyable as a gaming machine are the ease with which it can be programmed to create graphic images, and the speed at which it produces them. We will work our way through two game programs in this section.

The first game, which everyone will recognize, is an honest version of what is described by the *Oxford American Dictionary* as "a swindling sleight-of-hand game using shells." I wrote this program in about 8 hours (it was probably closer to 12, but programmers remember time the same way fishermen recall their catch), and then spent a few days sporadically messing around with it and modifying it.

The second program is a very polished and professional-looking version of backgammon, written by Barry Preppernau of Microsoft Press. Barry tells me he spent 50 hours writing this program, which is about how long it took me just to learn to play the board version of backgammon. (If I had had this game and a Macintosh, the learning process would have been shorter, as the program refuses to allow an illegal move.)

Computer games are based on every imaginable subject, and often serve purposes in addition to mere entertainment. Some "games" are designed to help people become comfortable using computers, and others teach specific skills, such as typing, spelling, or math.

Most computer games fall into one or more of three broad classes: puzzles, adventures, and simulations; sometimes it is difficult to draw the line between them. Regardless of the class they fall into, Linda Gail Christie, author of *The Successful Computer Game* (Computer BookBase, vol. 2, no. 4, 1983) maintains that games that become successful usually share several qualities: They are difficult to master; they reward good performance with bonus points, free games, or impressive sound and graphic displays; they keep track of previous scores and sometimes even promote competition by announcing the current high score among multiple players; they are unstructured, with varying levels of difficulty and speed; and they are unpredictable.

A few years ago, it seemed as though anyone who could write a program that met most of these criteria was assured of a steady income for life. The expectations of players, however, seem to rise at least as fast as the capabilities of computers, so to be profitable, games now require much more hard work and attention to detail than in the past. Of course, your motive for writing game programs may simply be to intrigue or amuse your friends, or instruct your children. If you do decide to publish for profit, these same people should be your first, and hopefully most critical, audience. The games offered as examples in the next two chapters were developed relatively rapidly and tested by only two or three people. If these games were to be offered for sale, this would be just the beginning point for hundreds of hours of testing, by people of every degree of computer and gaming sophistication.

If you do decide to attempt to break into this potentially profitable area of programming, spend a lot of time studying the market. Play the popular games and try to determine the qualities that contribute to their success. Rather than trying to write yet another version of the most popular game, try to transplant its qualities—that is, suspense, excitement, need for fast reflexes, eye-hand coordination, technical knowledge of a subject, dazzling graphics, and so on—to another setting. And above all else, stick to subjects that you know about and that interest you. If you aren't excited about playing your own game, it is doubtful that others will be either.

The Shell Game

Chapter 14

For those who have lead a really sheltered life, I'll start by explaining how to play the shell game. Take a look at Figure 14-1, which is the playing area. The game is started by placing the pea under one of the shells. In the manual version of the game, a person with very quick hands shuffles the shells around and then gives you a chance to guess

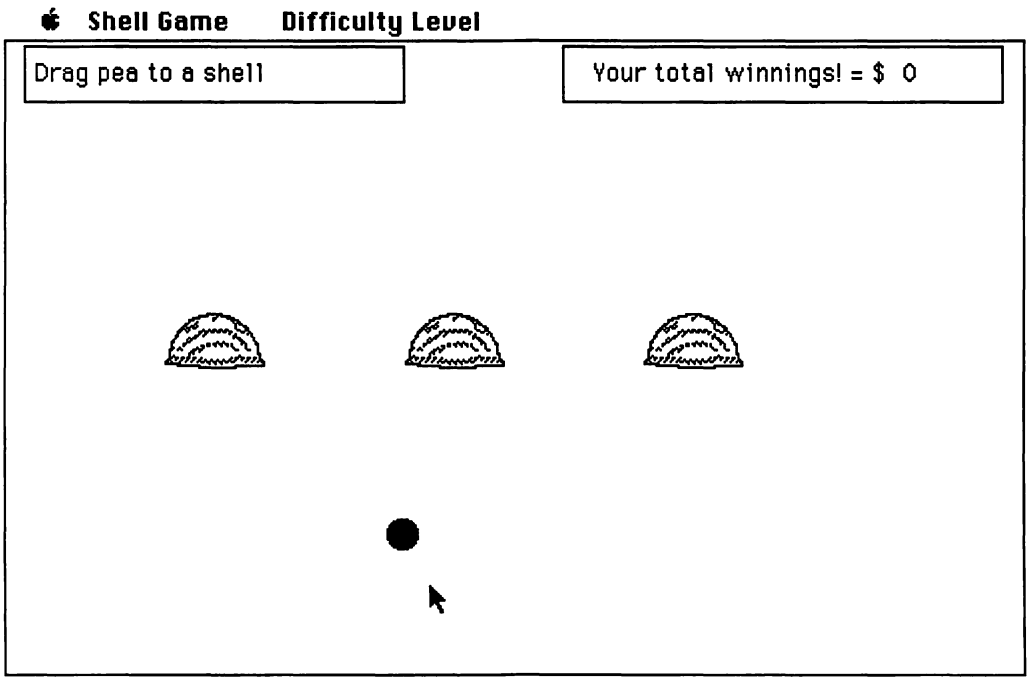


Figure 14-1. The shell-game playing area

which shell is covering the pea. In this version of the game, the computer randomly moves the shells, two at a time, for a preset period. The player chooses the level of difficulty, which changes the speed the shells move.

As we have worked our way through the programs in this book, I have tried to approach the explanations from various directions, in order to give you a better understanding of the logic behind them, the techniques used, and the troubles encountered. You are already familiar with most of the commands and techniques used in the shell-game program, so rather than discuss the finished program in the order in which the lines appear or the order in which they are executed, as we have done until now, let's work our way through a rough draft of the program and look at a few of the problems that cropped up while I was writing it.

The listing in Figure 14-2 is the finished program. If you like games of this sort, you could type it in and play it a few times, to relate the program to what actually happens on the screen.

```
'* The Shell Game
'*

   '*
   '* Initialize for first game.
   '*
   GOSUB InitializeVariables
   GOSUB CreateScreen
   GOSUB InitializeShell

'*
'* Initialization subroutines for all games after first.
'*
NextGame:
   GOSUB InitializeBoard

'*
'* Interaction with player starts at this point.
'*
MovePea:
   DisplayMessage 15, 20, outline, "Drag pea to a shell"
```

Figure 14-2. The complete shell-game program

more...

```
    WHILE MOUSE(0) <> -1
    WEND
    xPtr = MOUSE(3)
    yPtr = MOUSE(4)
    IF ABS(xPtr - xPea) > 20 OR ABS(yPtr - yPea) > 20 GOTO MovePea          'near pea?
    WHILE MOUSE(0) = -1
      endX = MOUSE(5)
      endY = MOUSE(6)
      IF (endX = xPea) AND (endY = yPea) THEN GOTO SkipMove
      PlacePea                                                              'erase pea
      xPea = endX
      yPea = endY
      PlacePea                                                              'redraw pea
SkipMove:
    WEND
    IF ABS(endY - yShel(1)) > 28 GOTO PlacePea                              'near shell?
    FOR shell = 1 TO 3
      IF ABS(endX - xShel(shell)) < 28 THEN hide = shell : GOTO Shuffle
    NEXT
    GOTO MovePea

'*
'* Initialize variables.
'*
InitializeVariables:
    DEFINT a - z                                                    'for speed
    RANDOMIZE TIMER                                 'reseed random-number generator
    DIM shellArray(151)                                             'image of shell
    DIM peaArray(36)                                                'image of pea
    DIM bound(3)                                                    'initial location of pea
    DIM xShel(3)                                    'x coordinate of center of each shell
    DIM yShel(3)                                                    'y coordinate
    DIM pat(3)                                                      'pea pattern
    DIM dif$(4)                                     'description of difficulty level
    PENMODE 10                                                      'invert mode
    dif= 20                                                         'initial level of difficulty
    odds = 2                                                        'initial odds
    winnings! = 0!                                                  'initial winnings
    creditLimit! = winnings! + 5000                                 'initial credit limit
    true = -1                                                       'logical true
    false = 0                                                       'logical false
    outline = true                                                  'used for messages
    noOutline = false                                               'used for messages
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
   '*
   '* Descriptions of the levels of difficulty
   '*
   dif$(1) = "Boring"
   dif$(2) = "Easy"
   dif$(3) = "Challenge"
   dif$(4) = "Dizzy"
   RETURN

'*
'* Open window and create menu.
'*
CreateScreen:
   WINDOW 1, , (0, 20) - (512, 342), 3                    'full screen for background

MenuInit:
   MENU 1, 0, 1, "Shell Game"
   MENU 1, 1, 1,  "Start New Game"
   MENU 1, 2, 1, "Quit to BASIC"
   MENU 1, 3, 1, "Quit to Desktop"
   MENU 3, 0, 1, "Difficulty Level"
   MENU 3, 1, 1, "1-Boring"
   MENU 3, 2, 1, "2-Easy"
   MENU 3, 3, 1, "3-Challenge"
   MENU 3, 4, 1, "4-Dizzy"
   MENU 2, 0, 0, "" :  MENU 4, 0, 0, "" : MENU 5, 0, 0, ""
   MENU 3, dif \ 10, 2                                    'check initial selection
   ON MENU GOSUB MenuHandle
   MENU ON
   RETURN

'*
'* Read data statements describing shell and store in shellArray.
'*
InitializeShell:
   FOR count = 0 TO 151
      READ shellArray(count)
   NEXT
   RETURN
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
'*
'* Prepare board for starting game. Place shells and pea in their initial
'* locations, and display total winnings or losses.
'*
InitializeBoard:
   CLS

   '*
   '* Specify initial location for center of each shell.
   '*
   xShel(1) = 105 : xShel(2) = 225 : xShel(3) = 345
   yShel(1) = 150 : yShel(2) = 150 : yShel(3) = 150

   '*
   '* Vertical and horizontal offsets (from center of each shell)
   '* determine four boundaries of shell (top, bottom, left, right).
   '*
   yOffset = 15 : xOffset = 25                          'vertical and horizontal offsets

   '*
   '* Place each shell at its starting location
   '*
   PlaceShell 1
   PlaceShell 2
   PlaceShell 3

   '*
   '* Draw pea then store its image in peaArray
   '*
   xPea = 200 : yPea = 250                          'initial x/y coordinates pea center
   bound(0) = yPea - 8                                                   'top of pea
   bound(1) = xPea - 8                                                   'left edge of pea
   bound(2) = yPea + 8                                                   'bottom of pea
   bound(3) = xPea + 8                                                   'right edge of pea
   FOR count = 0 TO 3                                    'store black pattern for pea
      pat(count) = -1
   NEXT
   FILLOVAL VARPTR (bound(0)), VARPTR (pat(0))

   '*
   '* Store image of pea.
   '*
   GET (bound(1), bound(0)) - (bound(3), bound(2)), peaArray(0)
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
    '*
    '* Display player's financial condition relative to beginning of game.
    '*
    IF winnings! < 0 THEN message$ = "Your total losses = $" + STR$(ABS(winnings!))
    IF winnings! => 0 THEN message$ = "Your total winnings! = $" + STR$(winnings!)
    DisplayMessage 285, 20, outline, message$
    RETURN

'*
'* Select two shells at random and exchange their position repeatedly.
'*
Shuffle:
    HIDECURSOR

    '*
    '* Blink pea.
    '*
    FOR blink = 1 TO 3
        PlacePea                                              'erase pea
        Pause 2000
        PlacePea                                              'draw pea
        Pause 2000
    NEXT blink
    PlacePea                                                  'final erase

    '*
    '* Set length of time shells will be shuffled.
    '*
    start! = TIMER                                            'note single precision
    finish! = start! + 15
    WHILE finish! > TIMER

        '*
        '* Randomly select first shell to move.
        '*
        aShell = INT(RND(1) * 3 + 1)
SelectB:                                                      'other shell
        bShell = INT(RND(1) * 3 + 1)
        IF bShell = aShell GOTO SelectB                       'can't select shell twice
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
'*
'* Swap shells.
'*
Switch:
    MENU OFF
    xDistance = xShel(bShell) - xShel(aShell)                    'how far apart
    mult = 2 * dif * SGN(xDistance)                              'multiplier--used later
    IF ABS(xDistance) > 200 THEN mult = 1 * dif * SGN(xDistance)
    FOR moveNumber = 1 TO ABS(xDistance) \ dif
      PlaceShell aShell                                          'erase first shell
      xShel(aShell) = xShel(aShell) + dif * SGN(xDistance)   'distance, direction to move
      yShel(aShell) = yShel(aShell) + mult * SIN(6.283 * moveNumber / (xDistance \ dif))
      PlaceShell aShell                                          'redraw first shell
      PlaceShell bShell                                          'erase second shell
      xShel(bShell) = xShel(bShell) - dif * SGN(xDistance)   'distance, direction to move
      yShel(bShell) = yShel(bShell) - mult * SIN(6.283 * moveNumber / (xDistance \ dif))
      PlaceShell bShell                                          'redraw second shell
    NEXT moveNumber

    '* Following five lines are needed only if shells tend to drift
    '* out of horizontal alignment. This will happen if distance between
    '* shells is not evenly divisible by difficulty level.

    'PlaceShell aShell                                           'erase first shell
    'PlaceShell bShell                                          'erase second shell
    'yShel(aShell) = 150 : yShel(bShell) = 150                  'compensate for vertical error
    'PlaceShell aShell                                          'place shells
    'PlaceShell bShell

    MENU ON
  WEND

  '*
  '* Reset number of guesses before starting Bets section
  '*
  numGuess = 0

'*
'* Give suckers chance to part with their money.
'*
Bets:
  DisplayMessage 15, 20, outline, "Care to place a bet?"
  WINDOW 2, , (50, 55) - (350, 140), -4
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
    DisplayMessage 10, 25, noOutline, "The current difficulty level is " + dif$(dif \ 10)
    DisplayMessage 10, 50, noOutline, "Enter your bet"
    EDIT FIELD 1, "", (110, 36) - (170, 51)
    DisplayMessage 20, 75, noOutline, "odds are" + STR$(odds) + " to 1"
    BUTTON 1, 1, "OK", (220, 50) - (280, 70)
    SHOWCURSOR

'*
'* Wait for OK button or Return.
'*
Loop:
    event = DIALOG(0)
    IF event = 1 OR event = 6 THEN GOTO Done
    GOTO Loop

'*
'* Retrieve bet.
'*
Done:
    ON ERROR GOTO Overflow                                    'number too large
    bet! = VAL(EDIT$(1))
    ON ERROR GOTO 0                                           'disable error trapping
    IF bet! < 0 THEN BEEP : WINDOW 1 : GOTO Bets
    IF bet! > creditLimit! THEN GOTO OverLimit
    WINDOW CLOSE 2
    DisplayMessage 15, 20, outline, "Select a shell"

'*
'* Guess which shell pea is under.
'*
Guess:
    DisplayMessage 15, 20, outline, "Which shell? (Click)"
    WHILE MOUSE(0) <> 1 : WEND
    xPtr = MOUSE(3) : yPtr = MOUSE(4)
    IF ABS(yPtr - yShel(1)) > 28 GOTO Guess                   'near shell vertically?
    FOR shell = 1 TO 3                                        'which shell?
      IF ABS(xPtr - xShel(shell)) < 28 THEN guessShell = shell : GOTO Show
    NEXT shell
    GOTO Guess
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
'*
'* Show sucker, uh ... player... what's under shell.
'*
Show:
   numGuess = numGuess + 1                                    'keep track of number of guesses
   PUT (xShel(guessShell) - xOffset, yShel(guessShell) - yOffset), shellArray   'erase shell
   IF hide = guessShell GOTO GotIt                                              'right one?
   winnings! = winnings! - bet!                                                 'amount lost
   creditLimit! = winnings! - bet!                                              'new credit limit
   IF creditLimit! < 2500 THEN creditLimit! = 2500                              'minimum credit limit
   DisplayMessage 295, 20, outline, "You lost $" + STR$(bet!)
   IF numGuess > 1 GOTO Guess ELSE GOTO Bets

'*
'* Correct shell was guessed.
'*
GotIt:
   PUT (xShel(guessShell) - 8, yShel(guessShell) - 8), peaArray
   DisplayMessage 15, 20, outline, "You got it!"
   IF numGuess > 2 GOTO Delay
   winnings! = winnings! + odds * bet!
   creditLimit! = winnings! + 5000
   IF winnings! > 10 ^ 7 - 1 THEN GOTO BrokeBank
   DisplayMessage 295, 20, outline, "You won $" + STR$(odds * bet!)
Delay:
   Pause 5000
   GOTO NextGame

'*
'* Routine branched to when item is selected from menu.
'*
MenuHandle:
   menuBar = MENU(0) : menuItem = MENU(1)
   MENU
   IF menuBar = 3 GOTO SetLevel
   IF menuItem = 1 THEN dif = 20 : winnings! = 0 : GOTO NextGame
   IF menuItem = 2 THEN CLEAR : END
   IF menuItem = 3 THEN SYSTEM
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
'*
'* Set difficulty level.
'*
SetLevel:
   MENU 3, dif \ 10, 1
   dif = 10 * menuItem
   odds = 2 ^ (menuItem - 1)
   MENU 3, menuItem, 2
   RETURN

'*
'* These are error routines
'*

'*
'* Invalid bet was entered.
'*
Overflow:
   WINDOW CLOSE 2
   WINDOW 3, , (50, 55) - (350, 140), -4
   BEEP
   DisplayMessage 15, 20, outline, "This bet is invalid"
   Pause 5000
   BEEP
   WINDOW CLOSE 3
   RESUME Bets

'*
'* Bet exceeds player's credit limit.
'*
OverLimit:
   WINDOW CLOSE 2
   WINDOW 3, , (50, 55) - (350, 140), -4
   BEEP
   message$ = "This bet exceeds your current credit limit"
   DisplayMessage 15, 20, noOutline, message$
   message$ = "which is $" + STR$(creditLimit!)
   DisplayMessage 15, 40, noOutline, message$
   BUTTON 1, 1, "OK", (220, 50) - (280, 70)
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
'*
'* Wait for OK button or Return.
'*
WaitForEvent:
   event = DIALOG(0)
   IF (event <> 1) AND (event <> 6) THEN GOTO WaitForEvent
   WINDOW CLOSE 3
   GOTO Bets

'*
'* Player won enough to force display into scientific notation.
'*
BrokeBank:
   WINDOW 2, , (20, 50) - (490, 250), -4

   '*
   '* Change to San Francisco textfont.
   '*
   TEXTFACE 49
   TEXTSIZE 12
   message$ = "You've won all my money and I can't afford to pay"
   DisplayMessage 20, 40, noOutline, message$
   message$ = "Let me off for 10¢ on the dollar or I'll blow your Mac Up!"
   DisplayMessage 20, 90, noOutline, message$
   TEXTFACE 0
   TEXTSIZE 12
   BUTTON 1, 1, "", (50, 110) - (70, 130)
   BUTTON 2, 1, "", (50, 145) - (70, 165)
   DisplayMessage 90, 127, noOutline, "This is extortion, but I'll do it"
   DisplayMessage 90, 162, noOutline, "I demand my money!"
   WHILE DIALOG(0) <> 1
   WEND
   IF DIALOG(1) = 2 THEN GOTO Crash
   winnings! = winnings! /10
   WINDOW CLOSE 2
   GOTO NextGame
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
Crash:

   '*
   '* Show Arcs, Sparks, and Soundeffects.
   '* Crash system.
   '*
   END

'*
'* These DATA statements contain values that are read into shellArray.
'* They were created with a utility program and merged with this program.
'*
DATA 51, 31, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 1020, 0, 0, 0, 15895
DATA -16384, 0, 1, -8096, -4096, 0, 7, 64
DATA 15360, 0, 4, 16384, 5632, 0, 24, -32768
DATA 2816, 0, 51, 16384, 2240, 0, 109, -32768
DATA 2144, 0, 212, 2413, -31136, 0, 392, 14043
DATA 24976, 0, 256, 9216, 18472, 0, 768, -14336
DATA 6152, 0, 517, -32768, 5636, 0, 1034, 0
DATA 772, 0, 3084, 18, 386, 0, 2096, 877
DATA -24253, 0, 2080, 6948, -24511, 0, 3200, 18432
DATA 4161, 0, 6465, -20480, 3121, 0, 4099, 8192
DATA 1, 0, 4101, 0, 1, -32768, 5124, 0
DATA 512, -32768, 7021, -28672 ,11702, -32768, 12873, 9362
DATA 18724, -16384, 26770, 23405, -20407, 24576, 32640, 292
DATA 0, 8192, 511, -8192, 4095, -8192, 0, 16383
DATA -2048, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0

'*
'* Print message at specified location near top of window.
'*
SUB DisplayMessage(x, y, box, message$) STATIC
   SHARED true, false
   LINE (x - 15, y - 18) - (x + 220, y + 10), 30, bf
   MOVETO x, y
   PRINT message$
```

Figure 14-2. The complete shell-game program *(continued)*

more...

```
    IF NOT box THEN EXIT SUB
    LINE (x - 5, y - 18) - (x + 215, y + 10), , b
END SUB

'*
'* Pause by counting to number passed.
'*
SUB Pause(count) STATIC
    FOR Pause = 1 TO count
    NEXT
END SUB

'*
'* Place image of pea at location specified by values of xPea and yPea.
'*
SUB PlacePea STATIC
    SHARED xPea, yPea, peaArray()
    PUT (xPea - 8, yPea - 8), peaArray
END SUB

'*
'* Place image of shell at location specified by values of xShel() and yShel().
'*
SUB PlaceShell(shellNum) STATIC
    SHARED shellArray(), xShel(), yShel(), yOffset, xOffset
    PUT (xShel(shellNum) - xOffset, yShel(shellNum) - yOffset), shellArray(0)
END SUB
```

Figure 14-2. The complete shell-game program *(continued)*

Writing the program

This is not a particularly complex program; but as much as I would like to claim that I simply sat down at the keyboard and produced it in its finished form, I didn't, so I won't. The program went through the same development stages as most of my programs: Think about it, sketch it out, write some rough code, debug the code so that it works with all the right choices, refine it, fix the problems introduced while refining, and then bulletproof it (try to trap potential errors so the program doesn't crash when the player does something wrong). Depending upon the program being written, some of the stages may have to be repeated several times.

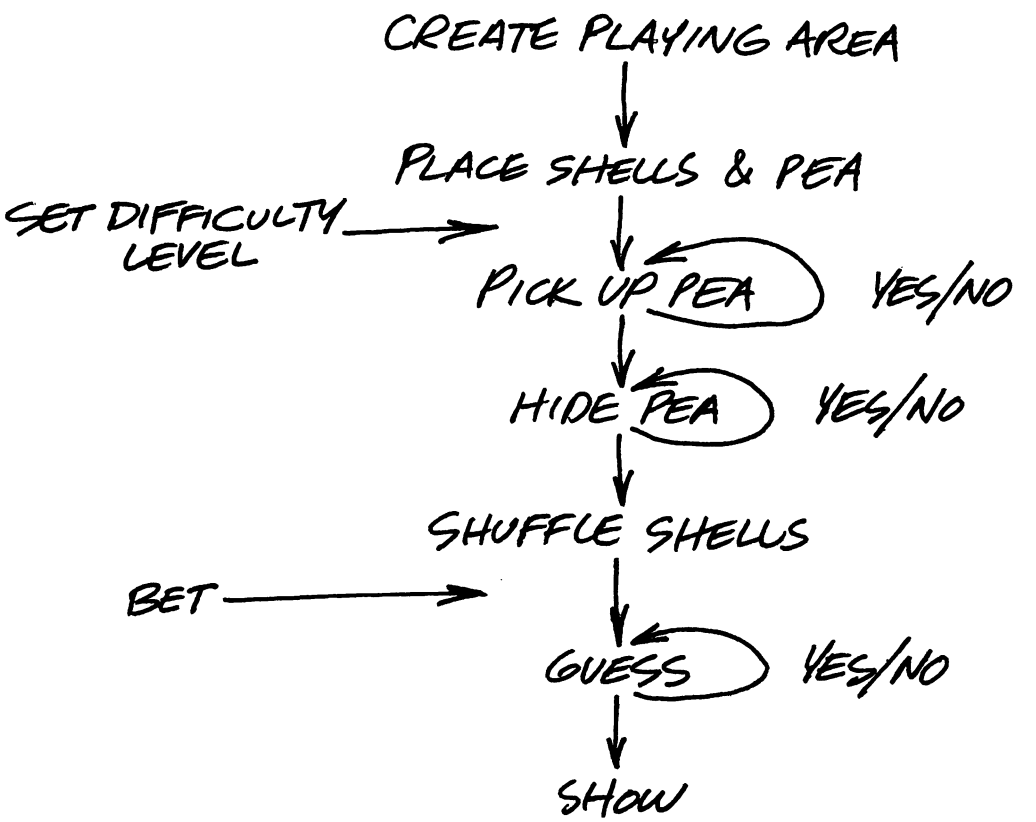


Figure 14-3. A preliminary flow chart for the shell game

Thinking and sketching

Since this program is fashioned after an existing game, I didn't have to give a lot of thought to what it would look like. My initial sketch looked like Figure 14-3, which isn't much of a flow chart, but it does get things moving in the right direction.

The betting and difficulty levels were afterthoughts that didn't get tacked on until I was halfway through the program and thought it was a little dull. The points I considered important in the beginning were to:

- Express all screen locations in terms of predefined variables, since I wasn't sure of the final size, shape, or location of the "shells" (I used circles until the very end).

- Decide whether to partition one window, use multiple windows, or both, in order to communicate with the player for displaying occasional instructions and scores, entering bets, and so forth.
- Include data statements that would create an image of a shell.
- Select the two shells moved in each swap and the direction of movement (clockwise or counterclockwise) randomly.
- Develop a formula that would vary the value of the y coordinate of the shell location to make it move up or down and then return to the same horizontal level.
- Have the program test for whether, when the player dragged the pea to the shell, the mouse was clicked close enough to the pea to "pick it up" and whether, after being dragged, the pea was dropped close enough to a shell to be considered under it; and then have the program determine which shell the pea is under.
- Either vary the distance moved each cycle or insert a variable delay to control the speed of movement and therefore the difficulty, since movement would be simulated by erasing the shells and redrawing them at a slightly different location and each erase/redraw cycle has the same minimum time requirement.

Taken one at a time, these seemed to be fairly simple problems—and since simple problems are the only kind I like, that's the way I took them.

Roughing in the code

There were several approaches open to me at this point. Some people like to see immediate results when they start a program. If that is your frame of mind and you know pretty much what a program you're writing will do, then creating the menus and the menu-handling routine is a good place to start. You can carry this stage as far as branching to an appropriately named subroutine for each menu selection and then placing a RETURN at that point (stubbing out the GOSUBs, as we did in the last section). Personally, I would rather write and modify the menus to follow the capabilities

```
DEFINT a - z
RANDOMIZE TIMER
DIM shellArray(200)
DIM bound(3), pat(3), peaArray(50)
WINDOW 1, , (0, 20) - (512, 342), 3
```

Figure 14-4. Initializing the program

of the program as they develop than restrict my program to the features I remembered to put on the menu.

I began with a couple of things that I knew from the start I was going to need: the usual define-declare-and-dimension section, and some random numbers. My first few lines of code, which made it through to the end almost unscathed, are shown in Figure 14-4. (If you are going to type this program into your computer, use the finished version, not these intermediate steps.)

The only new statement in this group is RANDOMIZE, which is used to reseed the random-number generator. I suppose the word "seed" is used in this context because the number supplied here—the seed number—is the one from which all the random numbers provided later are "grown." The syntax for this statement is:

RANDOMIZE [*expression*]

The optional *expression* after RANDOMIZE can be any integer between −32768 and 32767, or the function TIMER, which returns the number of seconds since midnight (based on the Macintosh clock) and therefore provides an "almost random" seed for the random-number generator. A little quick math and you will discover that there are 86,400 seconds in a day—a fact we will use with the TIMER function a little later.

The dimensions for *shellArray* and *peaArray,* the array variables to hold the images of the shell and the pea, are just estimates that I figured at the time would be at least large enough. During the refining stage, I replaced these with the minimum values required, to conserve memory. The information stored in the *bound* and *pat* arrays is used later for ROM calls that draw and fill shapes.

With this initialization out of the way, the next project was to create the playing surface, the three circles that will later be redrawn as shells, and the pea. Figure 14-5 shows how that started out.

```
InitializeBoard:
  CLS
  xShel(1) =105 : xShel(2) = 225 : xShel(3) = 345
  yShel(1) =150 : yShel(2) = 150 : yShel(3) = 150
  yOffset = 20 : xOffset = 20
  xPea = 200 : yPea = 250
  CIRCLE (xShel(1), yShel(1)), 20
  x = xShel(1) : y = yShel(1)
  GET (x - xOffset, y - yOffset) - (x + xOffset, y + yOffset), shellArray(0)
  PUT (xShel(2) - xOffset, yShel(2) - yOffset), shellArray(0)
  PUT (xShel(3) - xOffset, yShel(3) - yOffset), shellArray(0)
  bound(0) = yPea - 8
  bound(1) = xPea - 8
  bound(2) = yPea + 8
  bound(3) = xPea + 8
  FOR count = 0 TO 3
    pat(count) = -1
  NEXT
  FILLOVAL VARPTR (bound(0)), VARPTR (pat(0))
  GET (bound(1), bound(0)) - (bound(3), bound(2)), peaArray(0)
```

Figure 14-5. Creating the playing area

This section is run at the beginning of each game, to clear the playing area and place the pieces at their starting locations. The CLS statement clears the window before each game (always necessary after the first game). The array variables *xShel* and *yShel* define the centers of the three circles and *xPea* and *yPea* define the center of the pea. (I used the variable names *x* and *y* to shorten the first GET statement so that it would fit within the page format for this book.) Since the GET and PUT statements that move these images reference the upper left corner of the space in which they are drawn, a horizontal and vertical offset is needed when using those statements. It would be just as easy to refer to the pea and shells by their upper left corners instead of their centers, but the offsets would still be necessary, since many of the commands in the program reference the center of the pea or a shell. I used the variables *yOffset* and *xOffset* to hold the vertical and horizontal offsets between the center of the shell (circle) and the upper left corner.

The first circle is drawn with a CIRCLE statement, and then a GET statement is used to store the image in *shellArray.* All shells from this point on are drawn by PUTting *shellArray,* so when I eventually replace the circle with a picture of a shell, the only change I have to make is to store the drawing of the first shell in *shellArray.*

The pea is drawn as a filled oval, and GET is again used to store it in an array. I tried peas of a few different sizes and finally decided to stick with one 16 pixels across, so I didn't use variables to represent the pea offsets as I did for the shell, which will eventually change in size.

Now that I had three circles and a pea, the next problem was picking up the pea and placing it beneath one of the shells. This is done with the *MovePea* routine, shown in Figure 14-6. In this section, the program waits for the player to press and hold the mouse button, and then tests to see if the position of the pointer when the button was pressed was within 20 pixels of the center of the pea. This test is performed by computing the absolute value of the distance between both the x coordinate of the pointer and the center of the pea, and the y coordinate of the same points. If neither distance is greater than 20, the pointer is considered to be close enough.

```
MovePea:
  WHILE MOUSE(0) <> -1
  WEND
  xPtr = MOUSE(3)
  yPtr = MOUSE(4)
  IF ABS(xPtr - xPea) > 20 OR ABS(yPtr - yPea) > 20 GOTO PlacePea
  WHILE MOUSE(0) = -1
    endX = MOUSE(5)
    endY = MOUSE(6)
    PUT (xPea - 8, yPea - 8), peaArray(0)
    xPea = endX
    yPea = endY
    PUT (xPea - 8, yPea - 8), peaArray(0)
  WEND
  IF ABS(endY - yShel(1)) > 28 GOTO MovePea                    'near shell?
  FOR shell = 1 TO 3
    IF ABS(endX - xShel(shell)) < 28 THEN hide = shell : GOTO Shuffle
  NEXT
  GOTO MovePea
```

Figure 14-6. Moving the pea

Dragging the pea is pretty routine—simply a matter of PUTting the pea at the old location (in XOR mode) to erase it, and then PUTting it at the current location of the pointer. Since this is done rapidly, the pea seems to follow the pointer around the screen. The changes made later to this section provided instructions to the player and reduced the flicker when the pea is moved.

When the mouse button is released, the y coordinate is checked to see if it is on the same plane as the shells and, if it is, the x coordinate is compared with the ranges occupied by the shells to see if the pea is close enough to one of them to be considered under it. If either test fails, the program returns to *MovePea* and waits for the player to again click near the pea.

Once the pea is assigned to a shell, the next thing we want to do is shuffle the shells around. The first half of the routine to do this is shown in Figure 14-7, which sets a timer and then randomly selects two shells to move. This section posed a few interesting problems; that is, it posed a few problems that became interesting after they were solved.

```
Shuffle:
  FOR blink = 1 TO 3                                    'blink pea
    PUT (xPea - 8, yPea - 8), peaArray
    FOR Pause = 1 TO 2000 : NEXT
    PUT (xPea - 8, yPea - 8), peaArray
    FOR Pause = 1 TO 2000 : NEXT
  NEXT blink
  PUT (xPea - 8, yPea - 8), peaArray                    'hide pea
  start! = TIMER
  finish! = start! + 15                                 'set length of shuffle
  WHILE finish! > TIMER

    '*
    '* Randomly select first shell to move.
    '*
    aShell = INT(RND(1) * 3 + 1)
SelectB:                                                'other shell
    bShell = INT(RND(1) * 3 + 1)
    IF bShell = aShell GOTO SelectB                     'can't select shell twice
```

Figure 14-7. Moving the shells

The first part of the section, a FOR...NEXT loop, blinks the pea off and on three times, to warn the player that something is about to happen, and the PUT statement following the loop erases the pea a final time.

You will notice that the variables *start!* and *finish!* are single-precision variables, rather than our usual integers. This is because they are both set relative to the value returned by the TIMER function: the number of seconds since midnight, which we calculated earlier to range from 0 to 86399. Now, integer variables can hold positive values only up to 32767, which, in terms of time, works out to about six minutes after 9 o'clock in the morning. So guess who got this program working perfectly well with integer variables at 3 a.m. (when writers and programmers do their best work), and then tried to show it off to a friend in the afternoon. It's not a mistake I'll make again.

Back to our program. The values of *start!* and *finish!* are the current time and 15 seconds into the future, respectively. The couple of dozen lines of code between *WHILE finish! > TIMER* and the following WEND statement, which is in the next section, run repeatedly until TIMER returns a value equal to or greater than *finish!*. Even if you count the number of times the shells are moved, and then think of the number of times each shell is drawn during a move, you probably still won't be able to grasp how fast the Macintosh is whipping through this program.

The first thing that happens each time through the WHILE loop is the selection of the two shells to move. This selection is controlled by the RND function, which returns a random number greater than 0 but less than 1. Its syntax is:

RND[(X)]

The optional *X* in this function determines how the random number is generated, according to this schedule:

X Value	Action
< 0	Restarts the same sequence of numbers for any given X
> 0 or no X	Generates the next random number in the current sequence
= 0	Repeats the last number generated

The two shells to be moved are referred to as *aShell* and *bShell*. The variable *aShell* is first set equal to *INT(RND(1) * 3 + 1)*, which evaluates to either 1, 2, or 3, since INT returns the largest integer produced when the expression within its parentheses

is evaluated, and RND always returns a value less than 1. The variable *bShell* is then set equal to the same formula, and will of course also come up as 1, 2, or 3. Since we don't want *aShell* and *bShell* ever to be equal to the same number, we test for this condition and compute another value for *bShell* if they are.

I originally thought it would be easiest to randomly select just one shell, and then swap the other two. However, it turned out that there is a problem with this approach: With the range of randomly generated numbers limited to just three possible values, the same number often comes up eight or ten times in a row, which then causes the same two shells to be swapped eight or ten times in a row and takes a lot of the challenge out of the game. (As you will see when rolling dice in the backgammon game in Chapter 15, the same number comes up pretty often even when you are selecting one number out of six.)

Writing the part of this routine that actually swaps the shells was my biggest problem. If you typed the finished program and ran it, you noticed that as the shells are swapped, they move in a fairly smooth curve. Of course, this move is actually made up of a series of short moves controlled by GET and PUT statements. Each of these short moves has a horizontal and a vertical component: That is, to move diagonally up and to the left, a new set of coordinates is created that is so many pixels up and so many pixels to the left. The horizontal component wasn't particularly difficult, so I wrote that first. The two selected shells simply slid right and left, crossed over in the middle, and came to rest in each other's old position. The original version of this section is shown in Figure 14-8.

```
Switch:
    xDistance = xShel(bShell) - xShel(aShell)
    FOR j = 1 TO ABS(xDistance)
       PUT (xShel(aShell) - xOffset, yShel(aShell) - yOffset), shellArray
       xShel(aShell) = xShel(aShell) + SGN(xDistance)
       PUT (xShel(aShell) - xOffset, yShel(aShell) - yOffset), shellArray
       PUT (xShel(bShell) - xOffset, yShel(bShell) - yOffset), shellArray
       xShel(bShell) = xShel(bShell) - SGN(xDistance)
       PUT (xShel(bShell) - xOffset, yShel(bShell) - yOffset), shellArray
    NEXT
```

Figure 14-8. The original switch

The top half of the FOR...NEXT loop moves *aShell*; the bottom half moves *bShell*. The distance and direction of the move are determined by adding *SGN(xDistance)* to the value of *xShel(aShell)* and *xShel(bShell)* each time through the loop, before the shell is redrawn. The function:

SGN(X)

returns −1 if X is negative, +1 if X is positive, and 0 if X equals 0. Since *aShell* and *bShell* are in different locations, there is always distance between them, so *SGN(xDistance)* will always be either +1 or −1. By adding this value to the x coordinate of one shell and subtracting it from the x coordinate of the other, we move the shells in opposite directions.

When I tested this segment of the program, the shells moved just as expected, but they moved *incredibly* slowly. The reason for this, I realized, was that I was moving each shell only one pixel each time through the loop. Obviously, if I wanted to increase the shell speed, I had to increase the distance moved each time, which I could do by multiplying *SGN(xDistance)* by some fixed value each time I incremented *xShel(aShell)* and *xShel(bShell)*. Since this would make the shell move a greater distance each time through the loop, I also had to decrease the number of times the program went through the loop by dividing *ABS(xDistance)* by the same value. This seemed like a good place to control the level of difficulty, so I divided and multiplied by a variable I named *dif*, which I defined as being equal to 20. The parts of this section that were changed are highlighted in Figure 14-9.

```
Switch:
    dif = 20
    xDistance = xShel(bShell) - xShel(aShell)
    FOR moveNumber = 1 TO ABS(xDistance) / dif
      PUT (xShel(aShell) - xOffset, yShel(aShell) - yOffset), shellArray
      xShel(aShell) = xShel(aShell) + dif * SGN(xDistance)
      PUT (xShel(aShell) - xOffset, yShel(aShell) - yOffset), shellArray
      PUT (xShel(bShell) - xOffset, yShel(bShell) - yOffset), shellArray
      xShel(bShell) = xShel(bShell) - dif * SGN(xDistance)
      PUT (xShel(bShell) - xOffset, yShel(bShell) - yOffset), shellArray
    NEXT
```

Figure 14-9. The changes to *Switch*

Once the shells were moving smoothly back and forth, I attacked the problem of making them move up and down. This required adding a number to the y coordinate during half the shell's journey from the beginning and to the ending point, and subtracting a number (or adding a negative number) during the other half. I needed a formula that would create a series of numbers that began near zero, increased to some value, returned to zero, increased in a negative direction, and returned to zero. I could then assign this range of numbers to the y coordinate and match it to the range of variation in the x coordinates, so each cycle through the FOR...NEXT loop would move the shell vertically and horizontally. The shell should follow a path similar to half of the sine curve commonly used to represent alternating current. And since SIN(X) is a standard trig function available in BASIC, it seemed that function could somehow be used to produce the numbers I needed here.

It seemed that way, but it has been more years than I care to remember since my last trigonometry class, and I didn't have a book of trig tables handy. So I did what most other programmers probably do in similar situations: I started experimenting. At times such as this the separate List and Command windows of BASIC are very convenient. I typed this command sequence into the Command window:

```
FOR J = 0 TO 15 : PRINT J, SIN(J) : NEXT
```

But before pressing the Return key to execute the command, I dragged through it and copied it to the Clipboard, so that I would not have to retype the entire sequence to try a slight variation on it: I could paste it back into the Command window, make minor changes, and press Return to execute it again. When I pressed Return the first time, the list of numbers in Figure 14-10 (on the next page) appeared in the Output window. This list of numbers starts, predictably, at 0, increases to about 0.9, decreases to about −0.9 (passing, of course, through 0 on the way), then begins increasing again, passing through 0 somewhere between $J = 6$ and $J = 7$.

It was apparent that incrementing the y coordinate by the sine of a series of numbers varying from 0 to a little over 6 would produce the results I was looking for. And about this time, the value "a little over 6" rang a bell, and I remembered that multiples of π were somehow significant. So I tried 6.283, which is about 2π and found that

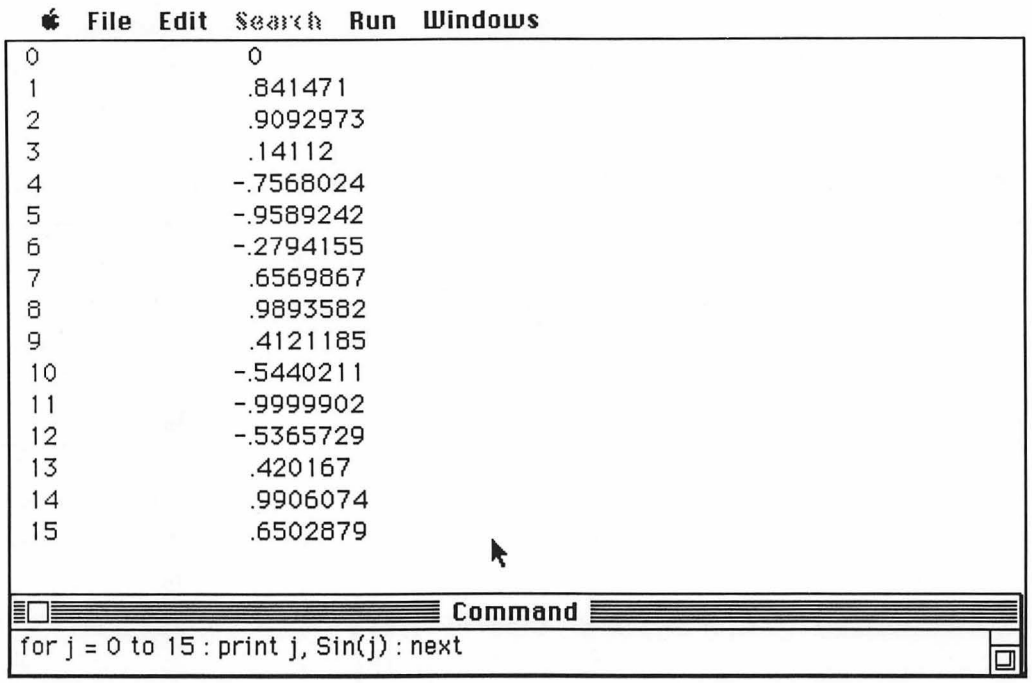


Figure 14-10. The numbers from 0 to 15 with their sines

it worked fine as the upper extreme of the range. I pressed Command-V to paste the previous command sequence back into the Command window and modified it as shown here:

```
CLS : FOR J! = 0 TO 6.5 STEP .5 : PRINT J!, SIN(J!) : NEXT
```

(Note the exclamation marks after the variable *J*, declaring it as a single-precision variable. This was necessary because the *DEFINT a-z* statement used in the program still affects commands entered into the Command window after the program has been run.) When I tested this new sequence, the program produced the numbers shown in Figure 14-11, substantiating my earlier impression.

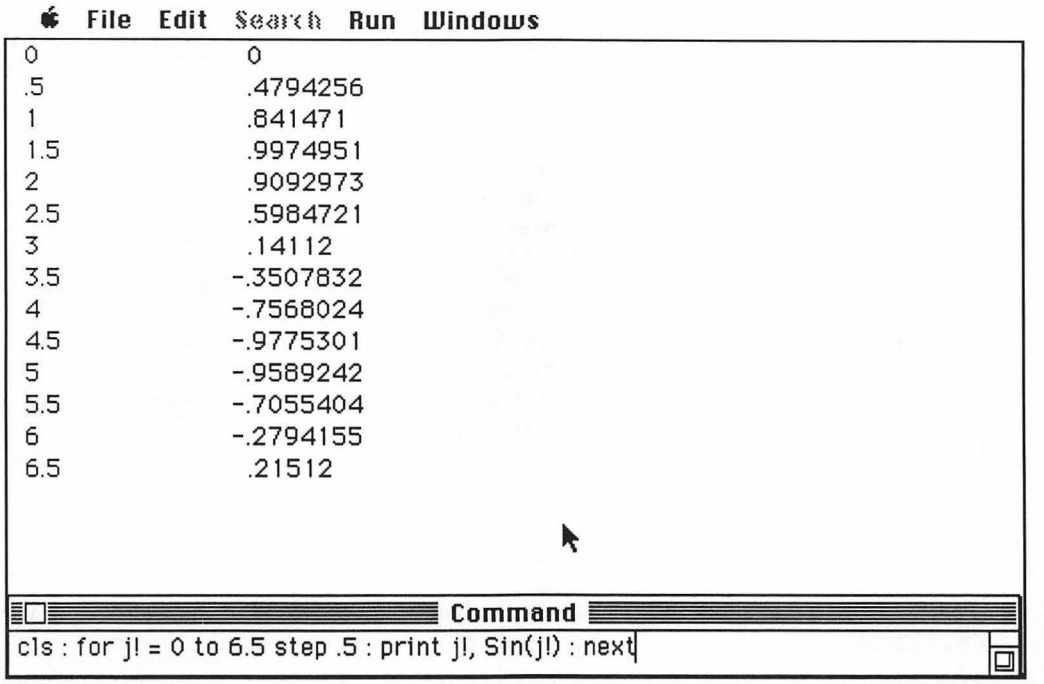


Figure 14-11. The numbers from 0 to 6.5 with their sines

Now I needed a number that would increase smoothly from about 0 to about 1, that I could use to multiply 6.283 by to create a smooth curve. Looking around for such a variable (I didn't want to complicate things by creating a new one), I realized that since *moveNumber,* the counter in the FOR...NEXT loop, varied from 1 to *xDistance \ dif,* then *moveNumber / (xDistance \ dif)* would range from something very small to 1. So after the line that increments *xShel(aShell),* I inserted the line:

```
yShel(aShel) = yShel(aShel) + SIN(6.283 * moveNumber / (xDistance \ dif))
```

and after the line that increments *xShel(bShell),* I inserted:

```
yShel(bShel) = yShel(bShel) - SIN(6.283 * moveNumber / (xDistance \ dif))
```

```
Switch:
    dif = 20
    xDistance = xShel(bShell) - xShel(aShell)
    mult = 2 * dif
    IF ABS(xDistance) > 200 THEN mult = 1 * dif
    FOR moveNumber = 1 TO ABS(xDistance) / dif
      PUT (xShel(aShell) - xOffset, yShel(aShell) - yOffset), shellArray
      xShel(aShell) = xShel(aShell) + dif * SGN(xDistance)
      yShel(aShel) = yShel(aShel) + mult * SIN(6.283 * moveNumber / (xDistance \ dif))
      PUT (xShel(aShell) - xOffset, yShel(aShell) - yOffset), shellArray
      PUT (xShel(bShell) - xOffset, yShel(bShell) - yOffset), shellArray
      xShel(bShell) = xShel(bShell) - dif * SGN(xDistance)
      yShel(aShel) = yShel(aShel) - mult * SIN(6.283 * moveNumber / (xDistance \ dif))
      PUT (xShel(bShell) - xOffset, yShel(bShell) - yOffset), shellArray
    NEXT
```

Figure 14-12. Adding the y coordinate multiplier to *Switch*

When I ran the program, I could see, by watching closely, that the shells did move up and down, but the value returned by *SIN(6.283 * moveNumber / (xDistance \ dif))* was so small that the total movement was barely noticeable. So now *this* value had to be multiplied by some number. By trial and error, I found that about 2 times the value assigned to *dif* seemed to work well when adjacent shells were selected, but made the arc too high when the two end shells were selected. So I added the variable *mult* to

```
Guess:
  LINE (10, 2) - (200, 30), , b
  MOVETO 15, 20
  PRINT "Which shell? (Click)"
  WHILE MOUSE(0) <> 1 : WEND
  xPtr = MOUSE(3) : yPtr = MOUSE(4)
  IF ABS(yPtr - yShel(1)) > 28 GOTO Guess                    'near shell vertically?
  FOR shell = 1 TO 3                                         'which shell?
    IF ABS(xPtr - xShel(shell)) < 28 THEN guessShell = shell : GOTO Show
  NEXT shell
  GOTO Guess
```

Figure 14-13. Guessing where the pea is now

```
Show:
  PUT (xShel(guessShell) - xOffset, yShel(guessShell) - yOffset), shellArray
  IF hide = guessShell GOTO GotIt
  GOTO Guess
```

Figure 14-14. Checking the guess

each of the lines that increments a y coordinate, and gave *mult* two definitions, depending upon the value of *xDistance.* Figure 14-12 shows the finished routine.

This routine seemed to move the shells around in a reasonable manner, so I could go on to check the player's guess. This section, shown in Figure 14-13, took no time to write, since I could use essentially the same routine used to place the pea.

When the click is close enough to a shell, the program goes to the *Show* routine (Figure 14-14), which erases the selected shell and checks to see if the pea was hidden under it. If the correct shell was selected, the program goes to *GotIt,* shown in Figure 14-15, otherwise it returns to *Guess* to give the player another try.

Getting the program running

The description I just gave was a very streamlined version of the process by which I wrote this program. I left out a lot of the trial-and-error experimentation used to find the correct value for a variable or the ideal position for an image on the screen. There were a few times that the program simply didn't do what I thought it should, and I could not see why. At these times, the TRON statement, which traces the flow of the program, proved its usefulness.

```
GotIt:
  PUT (xShel(guessShell) - 8, yShel(guessShell) - 8), peaArray
  FOR pause = 1 TO 5000 : NEXT
  GOTO InitializeBoard
```

Figure 14-15. The correct-guess routine

You can turn Trace on and off from the Run menu, from the Command window, or from within a program. If you know about where the program is going haywire, insert this line just before that point:

```
LIST : TRON
```

Before you run the program with Trace turned on, arrange the List window so that it is as large as possible without covering anything you will have to see or click in the output window. When the *LIST : TRON* statement is encountered in the program, the List window will be displayed and each command outlined as it is executed. (For even more refined troubleshooting, choose the step option.) When a program "isn't running right," it usually *is* running precisely "right"—it simply isn't running the way you expected it to. The computer, unfortunately, doesn't know what you expect it to do, only what you tell it to do. Trace will point out where the program takes off on a tangent from your expectations.

Adding refinements

The most obvious refinement to this program was the addition of menus. You have been through menus enough times that these won't require much explanation. As you saw in the complete program listing shown at the beginning of the chapter, I placed all the statements creating the menus in a routine called *MenuInit,* just before *InitializeBoard.* I used menus 1 and 3, just to space them out a bit, and turned off BASIC's menus 2, 4, and 5 by creating new menus in their slots with zero in the *state* position. Then I used the ON MENU GOSUB statement to tell the program where to go when a selection is made from a menu, and activated menu-event trapping with the MENU ON statement.

I put the subroutine that handles trapped menu events, *MenuHandle,* at the end of the program. Figure 14-16 shows the *MenuInit* routine and all the subroutines used to handle menu selections. Some of these features were on the menus from the beginning, but others were added later, as they occurred to me.

```
MenuInit:
   MENU 1, 0, 1, "Shell Game"
   MENU 1, 1, 1,  "Start New Game"
   MENU 1, 2, 1, "Quit to BASIC"
   MENU 1, 3, 1, "Quit to Desktop"
   MENU 3, 0, 1, "Difficulty Level"
   MENU 3, 1, 1, "1-Boring"
   MENU 3, 2, 1, "2-Easy"
   MENU 3, 3, 1, "3-Challenge"
   MENU 3, 4, 1, "4-Dizzy"
   MENU 2, 0, 0, "" :  MENU 4, 0, 0, "" : MENU 5, 0, 0, ""
   MENU 3, dif \ 10, 2                                    'check initial selection
   ON MENU GOSUB MenuHandle
   MENU ON
   RETURN

MenuHandle:
   menuBar = MENU(0) : menuItem = MENU(1)
   MENU
   IF menuBar = 3 GOTO SetLevel
   IF menuItem = 1 THEN dif = 20 : winnings! =  0 : GOTO NextGame
   IF menuItem = 2 THEN CLEAR : END
   IF menuItem = 3 THEN SYSTEM

'*
'* Set difficulty level.
'*
SetLevel:
   MENU 3, dif \ 10, 1
   dif = 10 * menuItem
   odds = 2 ^ (menuItem - 1)
   MENU 3, menuItem, 2
   RETURN
```

Figure 14-16. Handling the menus

The *Difficulty Level* menu required this statement at the end of the *MenuInit* section, to place a check mark in front of the current difficulty level.

```
MENU 3, dif \ 10, 2
```

The menu selection is defined by the expression *dif* \ *10*. Since *dif* was originally defined as 20, menu 3 will initially have a check mark in front of the second item. If the player selects another difficulty level from the menu, *MenuHandle* diverts the program to *SetLevel*, which removes the check mark from the current level, redefines *dif* as 10 times the number of the item selected, and places the check mark in front of the new selection.

Communicating with the player

Because I wrote this program, its operation seems totally obvious to me. But it's important never to assume that the person running your program knows what to do next. Sometimes people who are new to computing are so busy being intimidated by the computer that they don't give themselves time to think about what is actually happening with the program. For this reason I wanted to be able to post a few notes to guide the player through the program—and of course, to announce the score.

Originally, rather than create separate windows, I used the LINE statement with the *b* option to draw a box, and the PRINT statement to display my comments. When it was necessary to remove the box or erase its contents, I used the LINE statement again, this time with the *bf* option, to create a box outlined and filled with white. If the white box was the same size and at the same location as the first box with its black outline, both the outline and its contents disappeared; if the white box was a pixel smaller all the way around, just the contents disappeared.

Somewhere around version four of the program, I decided to add a subprogram to display a message with or without a border (Figure 14-17). I could pass it the location to start printing the message, whether or not to put a border around it, and the contents of the message: The subprogram did the rest. I then used a series of statements similar to this:

```
DisplayMessage 15, 20, outline, "Drag pea to a shell"
```

to display messages appropriate for the different events in the program.

To indicate whether or not the message should be framed, I created the variables *outline* and *noOutline* and added them to the variable initialization section, setting them equal to *true* and *false* (which I also added).

```
'*
'* Print message at specified location near top of window.
'*
SUB DisplayMessage(x, y, box, message$) STATIC
   SHARED true, false
   LINE (x - 15, y - 18) - (x + 220, y + 10), 30, bf
   MOVETO x, y
   PRINT message$
   IF NOT box THEN EXIT SUB
   LINE (x - 5, y - 18) - (x + 215, y + 10), , b
END SUB
```

Figure 14-17. Displaying messages

Placing bets

The routines shown on the following page in Figure 14-18—*Bets, Loop,* and *Done*—were inserted in the program just before the *Guess* routine and are all used in placing a bet. At first, I made placing a bet a menu selection, allowing the player to select it if desired, but after playing the game a few times I decided I would rather have the option offered automatically before the first and second guesses, which is closer to the way the fellows with the fast fingers do it in the park. By placing the sequence of betting routines between the sequence that shuffles the shells and the *Guess* routine, the only changes that I had to make to the existing program were to add a routine to count the number of guesses and to redirect the program back to *Bets* after the first incorrect guess.

The *Bets* routine creates a second window, with an edit field and an OK button, and announces the current difficulty level and the odds. In order to present the difficulty level as a descriptive word (Easy, Challenge, and so forth), I went back to the beginning of the program, dimensioned another array—*dif$(4)*—and defined four elements of the array as the words describing the four difficulty levels. The current odds are determined by the difficulty, so I again returned to the beginning of the program to set an initial value and then added a line to the *SetLevel* routine to compute new odds each time the difficulty level is changed. A final change required by the addition of betting was initialization of the variable *winnings!* to zero at the start of the program and addition of a statement resetting it to zero when the player starts a new game by choosing that option from the menu.

```
'*
'* Give suckers chance to part with their money.
'*
Bets:
   DisplayMessage 15, 20, outline, "Care to place a bet?"
   WINDOW 2, , (50, 55) - (350, 140), -4
   DisplayMessage 10, 25, noOutline, "The current difficulty level is " + dif$(dif \ 10)
   DisplayMessage 10, 50, noOutline, "Enter your bet"
   EDIT FIELD 1, "", (110, 36) - (170, 51)
   DisplayMessage 20, 75, noOutline, "odds are" + STR$(odds) + " to 1"
   BUTTON 1, 1, "OK", (220, 50) - (280, 70)
   SHOWCURSOR

'*
'* Wait for OK button or Return.
'*
Loop:
   event = DIALOG(0)
   IF event = 1 OR event = 6 THEN GOTO Done
   GOTO Loop

'*
'* Retrieve bet.
'*
Done:
   bet! = VAL(EDIT$(1))
   IF bet! < 0 THEN BEEP : WINDOW 1 : GOTO Bets
   IF bet! > creditLimit! THEN GOTO OverLimit
   WINDOW CLOSE 2
   DisplayMessage 15, 20, outline, "Select a shell"
```

Figure 14-18. Placing a bet

Drawing the shells

One of the last changes I made to the program was to get rid of the circles I was using for shells, replacing them with something that looks a little more like a walnut shell. To do this, I drew the shell in MacPaint and copied it to the Scrapbook. I then wrote a short program to bring the image into BASIC and create a set of DATA statements that describe it. These DATA statements were saved as a file and I used the MERGE statement to add them to my program. All I had to do then was add a short routine (*ShellInit*) to read the DATA statements and store them in *shellArray.* Finally, I

deleted the line that created the circles, changed the offsets a bit, and was back in business. (The utility program I wrote to create the DATA statements is short and simple, and is explained in Appendix B, which presents a few short utility programs.

A possible correction

One of the standard tradeoffs in programming is speed against accuracy. I have consistently used integer variables where possible, since they require less memory space and are manipulated more rapidly by the computer. We usually think of accuracy as important primarily when dealing with money or with scientific calculations, but this game demonstrates another situation where the error introduced by rounding calculations off to the nearest integer can cause a problem. The number of moves made by the shells when they are exchanging positions is determined by the value *xDistance* \ *dif*. Now, both *xDistance* and *dif* are integers, but depending upon the value of each, the quotient may not be. All combinations of the present values of *xDistance* and *dif* in this program do produce an integer for *xDistance* \ *dif*, but if you decide to place the shells differently or to use different difficulty levels, you may find that the shells miss being lined up by a few pixels after each swap. The error is cumulative, and after 15 or 20 swaps, can become pretty large. The easiest solution to this problem is to realign the shells at the end of each swap, when the error is still so small that it is barely noticeable. Adding the five lines in Figure 14-19 after the FOR...NEXT loop that performs the swap will do the trick. (Since the present values of *xDistance* and *dif* don't require this correction, you may not want to include them in the final version of the program.)

```
'* Following five lines are needed only if shells tend to drift
'* out of horizontal alignment. This will happen if distance between
'* shells is not evenly divisible by difficulty level.

'PlaceShell aShell                                    'erase first shell
'PlaceShell bShell                                   'erase second shell
'yShel(aShell) = 150 : yShel(bShell) = 150     'compensate for vertical error
'PlaceShell aShell                                    'place shells
'PlaceShell bShell
```

Figure 14-19. Aligning the shells after each swap

Bulletproofing

The most common places for a program to go astray are those points where the user enters information. Either through poor instructions on the part of the programmer or lack of attention on the part of the user, incompatible information is entered. You ask the user for a number from 1 to 4, and you get 27, or maybe the word *three.* Or the classics—lowercase l's for 1's and uppercase O's for 0's. Using menus that let the user make choices with the mouse, rather than requiring a typed entry, helps prevent errors, but there are still places (for instance when a bet is entered) where things can go astray. So here are some techniques to help bulletproof your interactive programs.

The size of the edit field used in this program to enter the bet is four digits. You can, of course, make the edit field any size you want, but entering one number more than will fit will always cause an overflow error, and there will always be a high-roller who wants to bet a billion dollars. The solution is to trap the error right when it happens, and take some appropriate action. To avoid a potential crash, add the line:

```
ON ERROR GOTO Overflow
```

to the *Done* routine, just before the information in the edit field is retrieved, and then tack the routine in Figure 14-20 onto the program.

```
'*
'* Invalid bet was entered.
'*
Overflow:
   WINDOW CLOSE 2
   WINDOW 3, , (50, 55) - (350, 140), -4
   BEEP
   DisplayMessage 15, 20, outline, "This bet is invalid"
   Pause 5000
   BEEP
   WINDOW CLOSE 3
   RESUME Bets
```

Figure 14-20. The *Overflow* routine

If you expect a variety of errors, you could use the ERR function to return the number of the error that diverted the program to this routine. That isn't necessary in this program but you *will* want to add the line:

```
ON ERROR GOTO 0
```

to the *Done* routine to turn off error trapping just after the information is retrieved, since the *Overflow* routine is not appropriate for errors that might happen in other parts of the program.

It also seemed like a good idea to add a sliding credit limit, based on the amount of money won or lost. The credit limit is set and varied in other parts of the program; the *Done* routine then tests each bet against this credit limit and routes the program to *OverLimit* if necessary:

```
IF bet! > creditLimit! THEN GOTO OverLimit
```

The *OverLimit* routine displays the current credit limit and then returns to *Bets,* to let the player enter a new bet.

Another possible error is the entry of a negative number. This would not cause the program to crash, but it would allow the player to win money by guessing *incorrectly.* This is solved by testing the retrieved information and not accepting it if it is less than zero.

The final problem I trapped (I'm sure I missed a few) is the attempt to display too large a number. Any single-precision number over seven digits is expressed in scientific notation when displayed by the Binary version of BASIC (eight digits for the Decimal version). This is tested for in the *GotIt* routine. If the total winnings go over $9,999,999, the program branches to the *BrokeBank* routine (Figure 14-21).

That is about all there is to this program. It is fairly simple, but has the potential for expansion. You could add sound effects and sarcastic comments, or goad the player to bet more after losing. The odds and difficulty levels could also be manipulated more realistically. At the moment, the Crash routine simply ends the program; you

```
'*
'* Player won enough to force display into scientific notation.
'*
BrokeBank:
   WINDOW 2, , (20, 50) - (490, 250), -4

   '*
   '* Change to San Francisco textfont.
   '*
   TEXTFACE 49
   TEXTSIZE 12
   message$ = "You've won all my money and I can't afford to pay"
   DisplayMessage 20, 40, noOutline, message$
   message$ = "Let me off for 10¢ on the dollar or I'll blow your Mac Up!"
   DisplayMessage 20, 90, noOutline, message$
   TEXTFACE 0
   TEXTSIZE 12
   BUTTON 1, 1, "", (50, 110) - (70, 130)
   BUTTON 2, 1, "", (50, 145) - (70, 165)
   DisplayMessage 90, 127, noOutline, "This is extortion, but I'll do it"
   DisplayMessage 90, 162, noOutline, "I demand my money!"
   WHILE DIALOG(0) <> 1
   WEND
   IF DIALOG(1) = 2 THEN GOTO Crash
   winnings! = winnings! /10
   WINDOW CLOSE 2
   GOTO NextGame

Crash:

   '*
   '* Show Arcs, Sparks, and Soundeffects.
   '* Crash system.
   '*
   END
```

Figure 14-21. The *Broke Bank* and *Crash* routines

might want to add pyrotechnics and sound effects to make it live up to its accompanying comments.

If you prefer games you can get a little more involved in, the next one should keep you happy for a while.

The Backgammon Game

Chapter 15

Backgammon, which has a less tarnished reputation than the shell game, is a substantially more complex game to play. It is a game of strategy, tactics, and intimidation—all the things that contribute to a good war. Our program to produce this game is over 300 lines long. If you glance through it, however, you will find only one or two BASIC instructions with which you are not already familiar. This is a good example of the fact that every BASIC program, no matter how complex its function, is composed of a bunch of relatively simple commands. The challenge to a programmer's ingenuity is to use these simple commands in a clear, concise, unambiguous manner. The prerequisite for meeting this challenge is an understanding of both the commands and the problem you want to solve with them. Barry Preppernau was able to write this program in a relatively short time because, in addition to understanding BASIC, he understood the rules and techniques of backgammon.

How the backgammon program works

Although this is a fairly sophisticated program, the individual steps involved in making it run are no more difficult than in any other program. Two-thirds of the program deals with setting up the playing surface; the remainder takes care of making the moves, checking their validity, and keeping score.

The main differences between playing backgammon on a flat board with pieces you pick up and move, and playing Barry's version on the Macintosh, is that on the Macintosh it is much more difficult to make an illegal move and there is never an argument over where a piece was before you picked it up; if you have any doubts, the Undo button will quickly return the board to its condition before the move.

Barry's comments, which appear to the right of program lines, set off by an apostrophe, do a good job of explaining the function of individual lines. I have added remark statements, preceded by REM, to trace the order in which the sections would typically be executed. I have also listed, before each subroutine, the numbers of the sections that call that subroutine, to remind you where you should move back to when you get to the RETURN statement. And, as usual, you will find the complete program listing at the end of the chapter (Figure 15-20), and we'll work through it here a section at a time.

Beginning the game

The *BeginGame* section (Figure 15-1) is simply a group of statements that send the program first to the subroutines used to create the playing surface, and then to the routine that actually controls the game. This section was included in order to group these major segments together at the beginning of the program, rather than placing each where it would naturally appear in the flow of the program (typically in the sequence 2, 3, 15, and 16). By carefully arranging the sections in their natural order, you could eliminate this section and the RETURN statements at the end of *VariableInit, BoardInit,* and *MenuInit.* However, there would be no practical purpose in doing this, since it would make the logic of the program harder to follow, and because the activities of these sections take place before the game starts, the savings in time and memory would be insignificant.

```
REM Section 1. Call three subroutines that set up game.
BeginGame:
  CLEAR
  GOSUB VariableInit
  GOSUB BoardInit
  GOSUB MenuInit
  GOTO StartGame
```

Figure 15-1. Section 1: *BeginGame*

Initializing the variables

The *VariableInit* segment, shown in Figure 15-2 on the following page, defines and dimensions the arrays used in the program and also sets up a number of constants. Here is what these arrays represent:

Variable	Purpose
pat(3)	Holds values used to define pattern for point
rect(3)	Holds values of boundaries of rectangle to be used in ROM calls: *rect(0)* = top, *rect(1)* = left, *rect(2)* = bottom, and *rect(3)* = right
poly(10)	Holds values used by ROM call that creates polygon used as point shape (this new ROM call will be explained later)
board(28,3)	Serves three purposes: *board(1,1)* through *board(28,1)* hold x coordinate of upper left corner of 24 points, two areas in OFF, and two in BAR; *board(1,2)* through *board(28,2)* hold y coordinate; *board(1,3)* through *board(28,3)* hold number of pieces in areas 1 through 28 (negative for black pieces; positive for white)
oldBoard(28)	Holds location of all pieces before current move; used to restore board if Undo button clicked
tempBoard(28)	Holds board locations in middle of turn to allow for moving one piece multiple times
pointArray(1820)	Holds images of four points (light and dark, up and down); stored by GET statements and used by PUT statements as needed to draw original board and to redraw single points, so that when piece is moved, entire board does not have to be redrawn
circleArray(64)	Holds outline of playing piece; used when player is dragging piece to new location
diceImage(384)	Holds six faces of die; each face requires 64 bytes, so die of face *n* (*n* being 1 through 6) is specified by offsetting (n − 1) * 64 from *diceImage(0)*
grey(3), *black(3)*, and *white(3)*	Hold integer values of three main patterns; used in ROM calls to fill rectangles and ovals

```
REM Section 2--called by Section 1. Define variables to be used.
VariableInit:
  DEFINT a-z
  RANDOMIZE TIMER
  DIM rect(3), poly(10), board(28,3), oldBoard(28), tempBoard(28)
  DIM pointArray(1820), circleArray(64), diceImage(384)
  DIM grey(3), black(3), white(3)
  FOR x = 0 TO 3                    'integer values for grey, black, and white patterns
    grey(x) = -21931
    black(x) = -1
    white(x) = 0
  NEXT x
  dark = -30686                                              'pattern for dark points
  light = -8841                                              'pattern for light points
  pWidth = 34                                                      'width of points
  piWidth = 30                                                     'width of pieces
  wLeft = 0                                                      'window left side
  wRight = 511                                                  'window right side
  wTop = 20                                                          'window top
  wBot = 340                                                     'window bottom
  boardRight = 409                                              'board right side
  tUp = 0                                                     'top of upper points
  bUp = 150                                                'bottom of upper points
  tLow = 170                                                  'top of lower points
  lghtLeft = 425                                     'left side of light BAR and OFF
  dkLeft = 465                                       'left side of dark BAR and OFF
  offTop = 185                                                        'top of OFF
  barTop = 275                                                        'top of BAR
  FOR pnt= 1 TO 12          'X,Y for points on the board; board(pnt,1)=X, board(pnt,2)=Y
    board(pnt, 1) = 1 + (pnt - 1) * pWidth : board(pnt, 2) = tUp
    board(pnt + 12, 1) = 1 + (12 - pnt) * pWidth : board(pnt + 12, 2) = tLow
  NEXT pnt
  board(1, 3) = -2 : board(6, 3) = 5 : board(8, 3) = 3                'initial locations
  board(12, 3) = -5 : board(13,3) = 5 : board(17,3) =  -3              '<0 black, >0 white
  board(19,3) = -5 : board(24,3) = 2
  board(25, 1) = lghtLeft : board(25, 2) = offTop : board(26, 1) = dkLeft
  board(26, 2) = offTop : board(27, 1) = lghtLeft : board(27, 2) = barTop
  board(28, 1) = dkLeft : board(28, 2) = barTop
  'board(25, 3) = 10 : board(1, 3) = 5                          'end game piece locations
  'board(26, 3) = -10 : board(24, 3) = -5
  RETURN
```

Figure 15-2. Section 2: *VariableInit*

This section goes on to define a few integer constants that will be used in several places throughout the program. Most of these define the boundaries of the various areas on the screen (the window, board, upper points, lower points, BAR and OFF). However, the first two, *dark* and *light,* are used to define a dark and light pattern for alternating points. These constants will be assigned to the *pat* array when initializing the points.

The remainder of this section specifies the (x,y) coordinates for the 24 points, and the numbers and colors of the pieces that will be put on each point at the start of the game. The board shown in Figure 15-3 is the starting board, with the points numbered and the coordinates shown at the edge.

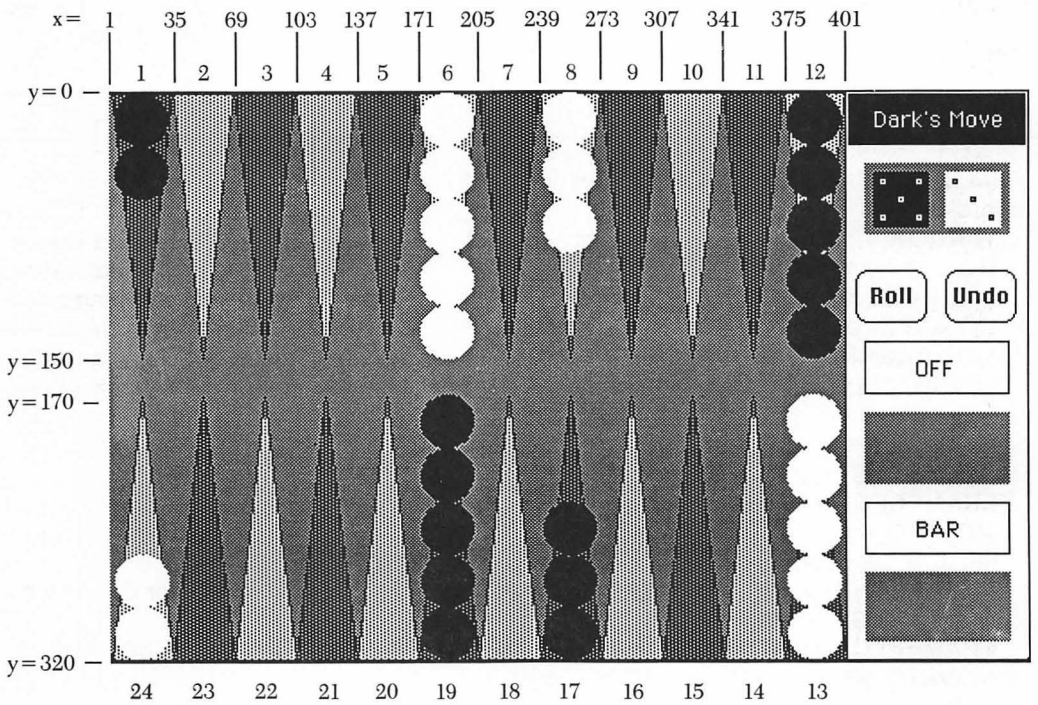


Figure 15-3. The starting board

Initializing the board

The *BoardInit* subroutine (Figure 15-4) orchestrates the creation of the playing surface. It opens a window and then calls the subroutines that create and store images of the points, the playing-piece outline, and the dice. It also sets up the mode for writing text, creates the Roll and Undo buttons, and draws the OFF and BAR sections of the screen. Finally, it calls additional subroutines to actually draw the points and pieces, and cast the dice for the first time.

Initializing the pointer

The *PointInit* subroutine, shown in Figure 15-5, uses the MOD operator to return the integer remainder from the division of two integers. If the dividend and divisor are not integers, BASIC rounds them to integers before performing the division

```
REM Section 3--called by Section 1. Set up board.
BoardInit:
  WINDOW 1, , (wLeft, wTop) - (wRight, wBot), 3          'use whole screen
  GOSUB PointInit : CLS                                  'initialize images in pointArray
  GOSUB CircleInit : CLS                                 'circle outline to use to drag piece
  GOSUB DiceInit : CLS                                   'initialize images in dice array
  LINE (boardRight, 0) - (boardRight, 340)               'separate board from buttons and dice
  TEXTMODE 2                                             'XOR all text
  BUTTON 1, 1, "Roll", (415, 100) - (455, 130)
  BUTTON 2, 1, "Undo", (465, 100) - (505, 130)
  rect(0) = 140 : rect(1) = 420 : rect(2) = 170 : rect(3) = 501
  FRAMERECT VARPTR(rect(0))                              'OFF title box
  MOVETO 448, 160 : PRINT "OFF"
  rect(0) = 180 : rect(2) = 220
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))              'backgnd box for OFF pieces
  rect(0) = 230 : rect(2) = 260
  FRAMERECT VARPTR(rect(0))                              'BAR title box
  MOVETO 448,250 : PRINT "BAR"
  rect(0) = 270 : rect(2) = 310
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))              'backgnd box for BAR pieces
  GOSUB BuildBoard                                       'put all points for first time
  GOSUB FirstRoll                                        'roll one die each and decide whose turn
  RETURN
```

Figure 15-4. Section 3: *BoardInit*

```
REM Section 4--called by Section 3. Define variables used to create four kinds of points:
REM light up and down, and dark up and down.
PointInit:
  FOR points = 1 TO 4                                          'four kinds of points
    IF points MOD 2 = 1 THEN pat = light                       'odd points light; up or down
    IF points MOD 2 = 0 THEN pat = dark                        'even points dark; up or down
    rect(0) = 50 : rect(1) = 33 : rect(2) = 200 : rect(3) = 67 'area to draw points
    poly(0) = 22 : poly(1) = 50 : poly(2) = 33 : poly(3) = 200 'set poly for up points
    poly(4) = 67 : poly(5) = 50 : poly(6) = 50 : poly(7) = 200
    poly(8) = 33 : poly(9) = 200 : poly(10) = 67
    IF points > 2 THEN poly(5) = 200 : poly(6) = 50 : poly(7) = 50   'down points
    IF points > 2 THEN poly(8) = 33 : poly(9) = 50 : poly(10) = 67
    GOSUB BuildPoint                                           'draw that point
    GET (33, 50) - (66, 199), pointArray((points - 1) * 455)   'fill array with all points
  NEXT points
  RETURN
```

Figure 15-5. Section 4: *PointInit*

operation that returns the MOD value. For example, *15.3 MOD 4.3* returns *3*, the remainder when 15 is divided by 4. This section of the program uses MOD to determine whether the number assigned to the varible *points* is odd or even: If a number is divided by 2 and has a remainder of 1, it is odd; if it has a remainder of 0, it is even.

The rest of the section defines the points that will be drawn as polygons by the *BuildPoint* subroutine. The technique for drawing a polygon is similar to the familiar technique for drawing a rectangle or oval. The main difference is in the number of items that have to be stored in the integer array before the ROM call. Rectangles and ovals, you will recall, require you to store the upper, left, lower, and right boundaries of the area to be filled with the shape. A polygon requires those four elements *plus* an (x,y) coordinate for each corner of the polygon. And, since the number of corners is itself a variable factor, one additional element is required to specify the total number of bytes (two per element) used to define the polygon. This sounds a little complex, but it isn't too bad after you have done it once.

The triangle used for a point on the playing board requires 11 elements in its array: six for the three sets of corner coordinates; four for the upper, left, lower, and right boundaries of the surrounding rectangle; and one to give the information that

22 bytes are used in all. In an array with the elements numbered from 0 to 10, these items would appear in this order:

Element	Contents
poly(0)	Number of bytes
poly(1)	Upper boundary
poly(2)	Left boundary
poly(3)	Lower boundary
poly(4)	Right boundary
poly(5)	y coordinate of first corner
poly(6)	x coordinate of first corner
poly(7)	y coordinate of second corner
poly(8)	x coordinate of second corner
poly(9)	y coordinate of third corner
poly(10)	x coordinate of third corner

Note the reversal of the normal (x,y) order of expressing coordinates. The polygon shown in Figure 15-6, which is actually drawn in the next section, displays the array variables assigned in this section and required to draw it.

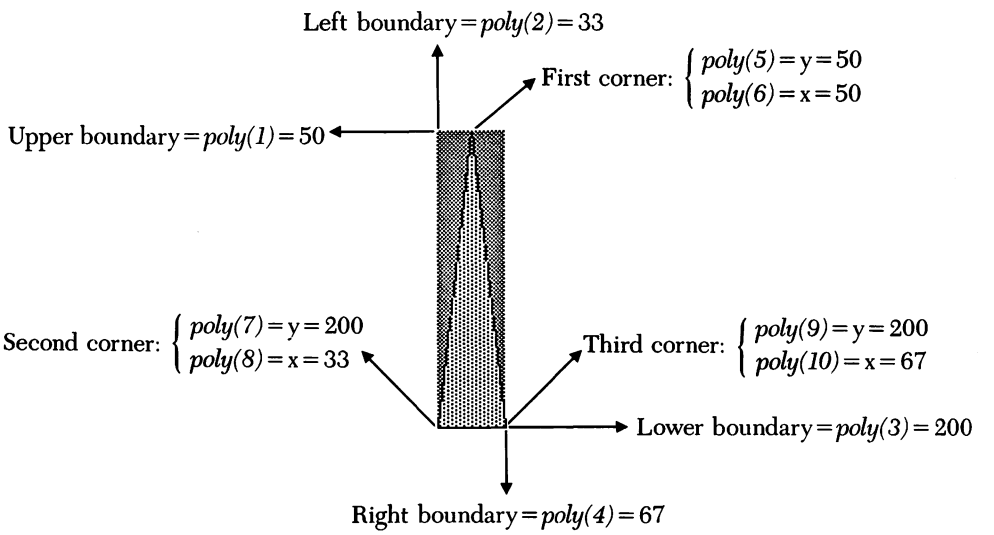


Figure 15-6. The array elements required to draw a polygon

```
REM Section 5--called by Section 4. Draw points defined in Section 4.
BuildPoint:
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))          'backgnd for all points
  FOR x= 0 TO 3 : pat(x) = pat : NEXT x              'set up pattern for actual point
  FILLPOLY VARPTR(poly(0)), VARPTR(pat(0))           'draw point
  LINE(poly(6), poly(5)) - (poly(8), poly(7))        'outline point
  LINE - (poly(10), poly(9)) : LINE - (poly(6), poly(5))
  RETURN
```

Figure 15-7. Section 5: *BuildPoint*

Having defined the points, the *PointInit* subroutine calls the *BuildPoint* subroutine (shown in Figure 15-7) four times—once each time through the loop— to draw one set of light and dark points that point up and another set that point down. It draws each point on a grey background with *FILLPOLY* and then outlines it. When the points are all put on the screen, the grey rectangles around them will form a solid grey background for the playing board.

Drawing the pieces and dice

Figure 15-8 on the following page shows the two subroutines that create the pieces and dice. The *CircleInit* routine simply draws a circle and stores its image in an array. The *DiceInit* routine defines the basic black die and then draws and captures it six times, with from one to six white dots arranged on it.

A FOR...NEXT loop is used to control which die is being created. The *dice* counter is examined to determine which dots belong on this particular image of the die. Again, we use MOD to determine if the counter is odd; if it is, we know that a dot has to go in the center. If the counter is greater than 1, then a dot has to go in the upper left and lower right corners. If the counter is greater than 3, then the lower left and upper right corners get a dot. Finally, if the counter is equal to 6, two dots are drawn in the middle left and right. Notice the manner of offsetting each face in the *diceImage* array so that all faces can be stored and recovered from one array. Only the six versions of the black die are stored; to produce a white die, the black one is simply inverted.

```
REM Section 6--called by Section 3. Draw and store circle used as playing piece.
CircleInit:
  CIRCLE (50, 50), 15                                    'pieces have diameter of 30
  GET (35, 35) - (65, 65), circleArray                   'use array to drag outline of piece
  RETURN

REM Section 7--called by Section 3. Draw and store view of each surface of die.
DiceInit:
  rect(0) = 35 : rect(1) = 35 : rect(2) = 66 : rect(3) = 66          '30x30 square dice
  FOR dice = 1 TO 6
    FILLRECT VARPTR(rect(0)), VARPTR(black(0))  'dice are black, invert for white dice
    IF dice MOD 2 = 1 THEN CIRCLE (50, 50), 1, 30          'dot in center for odd dice
    IF dice > 1 THEN CIRCLE (40, 40), 1, 30 : CIRCLE (60, 60), 1, 30   'up left, low right
    IF dice > 3 THEN CIRCLE (60, 40), 1, 30 : CIRCLE (40, 60), 1, 30   'up right, low left
    IF dice = 6 THEN CIRCLE (40, 50), 1, 30 : CIRCLE (60, 50), 1, 30  'mid left, mid right
    GET (35, 35) - (65, 65), diceImage((dice - 1) * 64)     'store all images in one array
  NEXT dice
  RETURN
```

Figure 15-8. Sections 6 and 7: *CircleInit* and *DiceInit*

Creating the board

The *BuildBoard* subroutine shown in Figure 15-9 is called by *BoardInit* to draw the basic playing board. PENMODE 9 (the overlay mode) is used to OR the new pattern with the existing screen. The two FILLRECT statements fill in areas that aren't covered by the point rectangles: a one-pixel rectangle on the far left side, and the area between the bottom of the upper points and the top of the lower points. *BuildPoint* then uses the *FOR INT=1 TO 28* loop to call the *DrawPoint* subroutine 28 times to produce the points, the pieces, and the BAR and OFF areas.

Each time through the loop, new values are assigned to *ptX* and *ptY* (the x and y coordinates of the point) and to *pieces* (the number of pieces on the point). *DrawPoint* is called at the beginning of the game to draw each point, and again later in the game to redraw points that have had pieces moved on or off them. If the number of the point to be drawn is greater than 24, a GOTO directs the program to the *BarAndOff* routine. The two lines that are executed *IF pnt > 12* draw the upward-pointing points

```
REM Section 8--called by Section 3. Draw board and points.
BuildBoard:
  PENMODE 9
  rect(0) = bUp : rect(1) = wLeft : rect(2) = tLow : rect(3) = boardRight
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))   'fill in between upper and lower points
  rect(0) = wTop : rect(1) = wLeft : rect(2) = wBot : rect(3) = 1
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))                'fill in left side of screen
  FOR pnt = 1 TO 28                    'draw all points; (ptX,ptY) = top left corner of point
    ptX = board(pnt, 1) : ptY = board(pnt,2) : pieces = board(pnt, 3)
    GOSUB DrawPoint
    tempBoard(pnt) = board(pnt, 3)     'use tempBoard to move one piece multiple times
  NEXT pnt
  RETURN

REM Section 9--called by Sections 8, 21, 24, and 27. Draw and refresh points.
DrawPoint:
  IF pnt > 24 GOTO BarAndOff    '25=lightOFF, 26=darkOFF, 27=lightBAR, 28=darkBAR
  IF pnt > 12 AND pnt MOD 2 = 0 THEN points = 0          'set up index into pointArray
  IF pnt > 12 AND pnt MOD 2 = 1 THEN points = 1
  IF pnt < 13 AND pnt MOD 2 = 0 THEN points = 2
  IF pnt < 13 AND pnt MOD 2 = 1 THEN points = 3
  PUT (ptX, ptY), pointArray(points * 455), PSET                'overlay point on board
  GOSUB DrawPiece                                               'draw pieces on this point
  RETURN
```

Figure 15-9. Sections 8 and 9: *BuildBoard* and *DrawPoint*

on the lower row; the two lines that are executed *IF pnt < 13* draw the downward-pointing points on the upper row. For each row of points, *IF pnt MOD 2 = 0,* the point is dark; *IF pnt MOD 2 = 1,* it is light.

The pieces

After each point is drawn, the *DrawPiece* subroutine, shown in Figure 15-10 on the next page, is called by *DrawPoint* to draw the specified pieces on the point. (The number of pieces on the point, stored in the *board(point,3)* array, was assigned to the variable *pieces* in the *BuildBoard* subroutine.)

The majority of this section defines the location of the rectangle where the piece will be drawn. Notice that we have not saved an image of the pieces in an array; instead we call FILLOVAL every time. Since the ROM calls are so fast, this saves us

```
REM Section 10--called by Section 9. Draw pieces on points.
DrawPiece:
  IF pieces = 0 THEN RETURN                          'don't need to draw zero pieces
  IF ABS(pieces) > 5 THEN show = 5 ELSE show = ABS(pieces) 'only show first 5 pieces
  FOR piece = 1 TO show                                    'draw up to 5 pieces
    IF pnt < 13 THEN rect(0) = (piece - 1) * piWidth       'set up rect for current piece
    IF pnt < 13 THEN rect(2) = (piece - 1) * piWidth + piWidth
    IF pnt < 13 THEN rect(1) = (pnt - 1) * pWidth + 3
    IF pnt < 13 THEN rect(3) = (pnt - 1) * pWidth + 33
    IF pnt > 12 THEN rect(0) = 320 - (piece - 1) * piWidth - piWidth
    IF pnt > 12 THEN rect(2) = 320 - (piece - 1) * piWidth
    IF pnt > 12 THEN rect(1) = (24 - pnt) * pWidth + 3
    IF pnt > 12 THEN rect(3) = (24 - pnt) * pWidth + 33
    IF pieces < 0 THEN FILLOVAL VARPTR(rect(0)), VARPTR(black(0))    'draw black
    IF pieces > 0 THEN FILLOVAL VARPTR(rect(0)), VARPTR(white(0))    'draw white
  NEXT piece
  IF ABS(pieces) <= 5 THEN RETURN                  'did we draw all pieces on this point?
  TEXTSIZE 10                                      'write number of pieces stacked on top of:
  IF pnt < 13 THEN CALL MOVETO ((pnt - 1) * pWidth + 6, 20)        'top piece for upper
  IF pnt > 12 THEN CALL MOVETO((24 - pnt) * pWidth + 6, 307)  'bottom piece for lower
  PRINT ABS(pieces) - 5 : TEXTSIZE 12
  RETURN
```

Figure 15-10. Section 10: *DrawPiece*

memory with no loss in speed. If you have typed in this program, run it once with Trace turned on to slow it down, so that you can watch the order in which all these things are drawn.

The BAR and OFF areas

The BAR and OFF areas are where pieces are stored when they have either been bumped from the board by an opponent or finished their circuit and been removed. The *BarAndOff* subroutine (shown in Figure 15-11) first draws the areas and then adds any pieces that belong on them. The *pnt* = 25 and *pnt* = 26 values refer to light and dark pieces, respectively, on the OFF area; *pnt* = 27 and *pnt* = 28 are light and dark pieces in the BAR area.

```
REM Section 11--called by Section 9. Draw BAR and OFF areas and any pieces on them.
BarAndOff:
  IF pnt = 25 OR pnt = 26 THEN rect(0) = 180 : rect(2) = 220          'backgnd rect
  IF pnt = 27 OR pnt = 28 THEN rect(0) = 270 : rect(2) = 310
  IF pnt = 25 OR pnt = 27 THEN rect(1) = 420 : rect(3) = 460
  IF pnt = 26 OR pnt = 28 THEN rect(1) = 460 : rect(3) = 501
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))
  IF pieces = 0 THEN RETURN                          'don't need to draw zero pieces
  rect(0) = ptY : rect(1) = ptX : rect(2) = ptY + piWidth : rect(3) = ptX + piWidth
  IF pnt = 25 OR pnt = 27 THEN FILLOVAL VARPTR(rect(0)), VARPTR(white(0))
  IF pnt = 26 OR pnt = 28 THEN FILLOVAL VARPTR(rect(0)), VARPTR(black(0))
  IF pieces < 10 THEN CALL MOVETO(ptX + 3, ptY + 20)
  IF pieces > 9 THEN CALL MOVETO(ptX - 1, ptY + 20)
  PRINT ABS(pieces)                          'write number of pieces OFF or on BAR
  RETURN
```

Figure 15-11. Section 11: *BarAndOff*

Starting the play

Several routines are involved in actually setting up the board and getting down to the serious business of playing backgammon. These routines are discussed briefly in the next few paragraphs, but once again, you will want to rely on the comments in the program to help you follow the details.

The *FirstRoll* routine (Figure 15-12 on the next page) casts the initial roll of one die for each player, which determines the player (Dark or Light) who will go first. The RND(1) function is used in a short formula to assign a number between 1 and 6 to the variables *dice(1)* and *dice(2)*. Notice that the value assigned to *dice(1)* is multiplied here by −1, to make it negative and thereby keep the Dark player's die black. The die represented by *dice(2)* has a positive value, so it is inverted to make the Light player's die white. Notice also the use of the *oldDice* and *oldBoard* arrays to record conditions before a move, in case the player pushes the Undo button.

Next, the *WhoseTurn* subroutine (Figure 15-13 on the next page) tests the value of the variable *turn*, assigned initially in the *FirstRoll* subroutine, and later in the *Roll* subroutine. If *turn* is greater than zero, it is Light's turn to play, and the program goes to *LightTurn* (also Figure 15-13). If *turn* is less than zero, the program stays in the *WhoseTurn* subroutine and announces that it is Dark's move.

```
REM Section 12--called by Section 3. Determine who starts.
FirstRoll:
  dice(1) = INT(RND(1) * 6 + 1) * -1 : dice(2) = INT(RND(1) * 6 + 1)  '(1)= black, (2)=white
  IF ABS(dice(1)) = dice(2) GOTO FirstRoll                    'can't have doubles on first roll
  IF ABS(dice(1)) > dice(2) THEN turn = -1 ELSE turn = 1      'turn<0 for dark, >0 for light
  oldDice(1) = dice(1) : oldDice(2) = dice(2)                 'use oldDice for Undo
  GOSUB WhoseTurn                                             'draw turn title box
  GOSUB DrawDice
  FOR pnt = 1 TO 28                                  'record current board set up for Undo
    oldBoard(pnt) = board(pnt, 3) : tempBoard(pnt) = board(pnt, 3)
  NEXT pnt
  RETURN
```

Figure 15-12. Section 12: *FirstRoll*

DrawDice and *NextDice* (Figure 15-14) draw the dice after each roll, using the numbers generated in *FirstRoll* or *Roll* to offset the starting point in the *diceImage* array so that the correct dice are drawn. Each die is drawn in black, and then inverted if the value of *dice(1)* or *dice(2)* is not less than zero; that is, if it is Light's move.

The *MenuInit* routine shown in Figure 15-15 produces the backgammon menu, blanks out the BASIC menus, and specifies where to go (*MenuHandle*) if someone selects an item from the menu.

```
REM Section 13(a)--called by Sections 12 and 23. Determine and announce turn.
WhoseTurn:
  rect(0) = 1 : rect(1) = 411 : rect(2) = 30 : rect(3) = 510      'rect for turn title box
  IF turn > 0 GOTO LightTurn
  FILLRECT VARPTR(rect(0)), VARPTR(black(0))                      'draw black box
  MOVETO 425, 20 : PRINT "Dark's Move"
  RETURN

REM Section 13(b)--called by Section 13(a).
LightTurn:
  FILLRECT VARPTR(rect(0)), VARPTR(white(0))          'draw white box to clear rect
  FRAMERECT VARPTR(rect(0))                                       'frame title box
  MOVETO 420, 20 : PRINT "Light's Move"
  RETURN
```

Figure 15-13. Sections 13(a) and 13(b): *WhoseTurn* and *LightTurn*

```
REM Section 14(a)--called by Sections 12 and 23. Draw black dice and invert for white.
DrawDice:
   rect(0) = 40 : rect(1) = 420 : rect(2) = 80 : rect(3) = 501          'backgnd for dice
   FILLRECT VARPTR(rect(0)), VARPTR(grey(0))
   FOR dice = 1 TO 2                                                    'draw both dice
      IF dice = 1 THEN left = lghtLeft ELSE left = dkLeft               'left edge of dice
      PUT (left, 45), diceImage((ABS(dice(dice)) - 1) * 64), PSET 'show correct dice image
      IF dice(dice) < 0 GOTO NextDice        'dice is supposed to be black, draw next one
      rect(0) = 45 : rect(1) = left : rect(2) = 76 : rect(3) = left + 31
      INVERTRECT VARPTR(rect(0))                                        'have to invert for white dice

REM Section 14(b)--called by Section 14(a).
NextDice:
   NEXT dice
   RETURN
```

Figure 15-14. Sections 14(a) and 14(b): *DrawDice* and *NextDice*

The next sections control the actual flow of the game. At the start of the game, the next step is for the player who won the first roll of the dice to make a move. After a valid move has been made, the next step of the game is to roll the dice for the next

```
REM Section 15--called by Section 1. Create menu.
MenuInit:
   MENU 1, 0, 1, "Backgammon"
   MENU 1, 1, 1, "Start New Game"
   MENU 1, 3, 1, "Quit to BASIC"
   MENU 1, 4, 1, "Quit to Desktop"
   MENU 2, 0, 0, "" : MENU 3, 0, 0, "" : MENU 4, 0, 0, "" : MENU 5, 0, 0, ""
   ON MENU GOSUB MenuHandle : MENU ON
   RETURN

REM Section 29--called by menu-event trap.
MenuHandle:
   DIALOG STOP : MOUSE STOP
   menuBar = MENU(0) : menuItem = MENU(1)
   IF menuBar <> 1 THEN DIALOG ON : MOUSE ON : RETURN
   IF menuItem = 1 THEN MOUSE OFF : MENU OFF : DIALOG OFF : GOTO BeginGame
   IF menuItem = 3 THEN END
   IF menuItem = 4 THEN SYSTEM
```

Figure 15-15. Sections 15 and 29: *MenuInit* and *MenuHandle*

```
REM Section 16--called by Section 1.
StartGame:
  ON DIALOG GOSUB DialogHandle : DIALOG ON

REM Section 22--called by dialog-event trap.
DialogHandle:
  MENU STOP : MOUSE STOP
  event = DIALOG(0)
  IF event <>1 THEN MENU ON : MOUSE ON : RETURN  'don't care if no button pushed
  buttonNumber = DIALOG(1)
  IF buttonNumber = 1 THEN GOSUB Roll
  IF buttonNumber = 2 THEN GOSUB Undo
  MENU ON : MOUSE ON
  RETURN

REM Section 23--called by Section 22.
Roll:
  turn = turn * -1                                              'change turns
  dice(3) = 0 : dice(4) = 0                                     'dice(3) and (4) used for doubles
  dice(1) = INT(RND(1) * 6 + 1) * turn : dice(2) = INT(RND(1) * 6 + 1) * turn
  IF dice(1) = dice(2) THEN dice(3) = dice(1) : dice(4) = dice(1)   'oh boy, doubles!!!
  FOR x = 1 TO 4 : oldDice(x) = dice(x) : NEXT x                'oldDice used for Undo
  GOSUB WhoseTurn                                               'draw turn title box
  GOSUB DrawDice
  FOR pnt = 1 TO 28                                             'record current board set up for Undo
    oldBoard(pnt) = board(pnt, 3) : tempBoard(pnt) = board(pnt, 3)
  NEXT pnt
  RETURN

REM Section 24--called by Section 22.
Undo:
  FOR pnt = 1 TO 28                                             'reset points that have changed in this turn
    tempBoard(pnt) = board(pnt, 3)                              'record current number of pieces
    board(pnt, 3) = oldBoard(pnt)                               'reset to start of turn
    ptX = board(pnt, 1) : ptY = board(pnt, 2) : pieces = board(pnt, 3)
    IF pieces <> tempBoard(pnt) THEN GOSUB DrawPoint    'if any change redraw point
    tempBoard(pnt) = board(pnt, 3)          'tempBoard used for multiple moves of 1 piece
  NEXT pnt
  FOR x = 1 TO 4 : dice(x) = oldDice(x) : NEXT x                'reset dice
  RETURN
```

Figure 15-16. Sections 16, 22, 23, and 24: *StartGame, DialogHandle, Roll,* and *Undo*

move, or to undo the previous move to try something else. The *StartGame* routine (shown in Figure 15-16) simply tells the program where it should go if a dialog event is trapped. The only dialog event we are concerned with in this program is the pushing of one of the two buttons: Roll or Undo. *DialogHandle* then decides which button was clicked and branches to the appropriate subroutine. Notice that menu- and mouse-event trapping are turned off at the beginning of *DialogHandle,* and turned back on at the end, to prevent the player from interrupting the process in the middle.

If the player chooses to roll the dice, the *Roll* subroutine uses the RND function to assign values to each die. Unlike the *FirstRoll* subroutine, *Roll* allows doubles (a player who rolls doubles in backgammon gets to move four times, rather than the usual two). If the two dice match, the value of *dice(1)* is assigned to the array variables *dice(3)* and *dice(4),* which are normally set to zero. The *LegalMove* subroutine will later check these values to see whether to allow the extra moves.

The Undo button lets a player with second thoughts try a different move—a practice not always approved of by the opponent. Clicking this button calls the *Undo* subroutine, which restores the board and dice to their condition before the last move.

Making a move

The *MouseCheck* routine (Figure 15-17) is the main loop of the program. This section waits for the player to drag or click the mouse and then checks to see if the particular action is one of interest and, if so, branches to the appropriate subroutine.

```
REM Section 17--flows from Section 16. Wait for mouse action (main loop).
MouseCheck:
  WHILE MOUSE(0) = 0 : WEND
  FOR x =1 TO 600 : NEXT x                          'wait to check for double click
  mousePush = MOUSE(0)
  IF mousePush = 1 GOTO MouseCheck                  'don't care about single clicks
  IF mousePush < -1 GOTO MouseCheck                 'or double or triple clicks/drags
  IF mousePush = -1 THEN GOSUB LegalMove    'trying to drag piece, check for legality
  IF mousePush > 1 THEN GOSUB DoubleClick 'trying to take piece off, check for legality
  GOTO MouseCheck                                   'loop until someone wins
```

Figure 15-17. Section 17: *MouseCheck*

If the player is trying to move a piece by dragging the mouse, *LegalMove* and the three other sections shown in Figure 15-18—*GoodStart, MovePiece,* and *LegalEnd*—test each attempted move to ensure that it is valid. The drag must start from a point where the player has a piece, or from the BAR, if the player has a piece there. It must stop on a valid point: that is, a point that is the correct number of points away from the starting point and that doesn't have more than one of the opponent's pieces already on it. Until a move is checked, the only thing that actually changes on the board is the movement of an outline of the piece, used to show where the player is dragging. If, at

```
REM Section 18--called by Section 17. Check if move is legal.
LegalMove:
  stX = MOUSE(3) : stY = MOUSE(4) : stPnt = 0              'starting X,Y position of drag
  FOR pnt = 1 TO 24                                  'quick check for starting from legal point
    ptX = board(pnt, 1) : ptY = board(pnt, 2)
    IF stX < ptX OR stX > ptX + pWidth THEN NextPnt1
    IF stY >= ptY AND stY <= ptY + bUp THEN stPnt = pnt : pnt = 24
NextPnt1:
  NEXT pnt
  IF stPnt <> 0 GOTO GoodStart                       'started drag on board point 1-24
  FOR pnt = 27 TO 28                                   'check for dragging from BAR
    ptX = board(pnt, 1) :  ptY = board(pnt, 2)
    IF stX < ptX OR stX > ptX + piWidth THEN NextPnt2
    IF stY >= ptY AND stY <= ptY + piWidth THEN stPnt = pnt : pnt = 28
NextPnt2:
  NEXT pnt
  IF stPnt = 0 THEN RETURN                  'can't drag from anyplace but 1-24 or BAR

REM Section 19--called by Section 18. Check for valid starting place.
GoodStart:
  IF stPnt < 25 AND turn = -1 AND board(28, 3) <> 0 THEN RETURN          'still on BAR
  IF stPnt < 25 AND turn = 1 AND board(27, 3) <> 0 THEN RETURN
  startPiece = board(stPnt, 3)
  IF startPiece = 0 THEN RETURN                  'can't drag from point with no pieces
  IF turn < 0 AND startPiece > 0 THEN RETURN           'can't drag opponent's piece
  IF turn > 0 AND startPiece < 0 THEN RETURN
  PUT (stX - 15, stY - 15), circleArray          'OK to drag, put circle outline under pointer
```

Figure 15-18. Sections 18, 19, 20, and 21: *LegalMove, GoodStart, MovePiece,* and *LegalEnd*

more...

```
REM Section 20--flows from Section 19.
MovePiece:
   mousePush = MOUSE(0) : endX = MOUSE(5) : endY = MOUSE(6)
   IF mousePush <> -1 GOTO LegalEnd          'stopped dragging, check for legality
   PUT (stX - 15, stY - 15), circleArray          'erase outline at old location
   PUT (endX - 15, endY - 15), circleArray        'draw outline at new location
   stX = endX : stY = endY                   'use stX, stY to erase next time
   GOTO MovePiece                                 'loop until stop dragging

REM Section 21--called by Section 20.
LegalEnd:
   PUT (stX - 15, stY - 15), circleArray          'erase at last location
   endPnt = 0
   FOR pnt = 1 TO 24                         'quick check for legal ending point
      ptX = board(pnt, 1) : ptY = board(pnt, 2)
      IF endX < ptX OR endX > ptX + pWidth THEN NextPnt3
      IF endY >= ptY AND endY <= ptY + bUp THEN endPnt = pnt : pnt = 24
NextPnt3:
   NEXT pnt
   IF endPnt = 0 THEN RETURN                      'have to end on point 1-24
   endPiece = board(endPnt, 3)
   IF turn < 0 AND endPiece > 1 THEN RETURN   'can't end on point with >1 opponent
   IF turn > 0 AND endPiece < -1 THEN RETURN
   gdEnd = 0
   FOR dTry = 1 TO 4                     'check for moving correct number of points
      dValue = ABS(dice(dTry))
      IF stPnt = 27 AND endPnt = 25 - dValue THEN gdEnd = 1 : dice(dTry) = 0 : dTry = 4
      IF stPnt = 28 AND endPnt = dValue THEN gdEnd = 1 : dice(dTry) = 0 : dTry = 4
      IF stPnt + dValue * turn * - 1 = endPnt THEN gdEnd = 1 : dice(dTry) = 0 : dTry = 4
   NEXT dTry
   IF gdEnd = 0 THEN RETURN                       'sorry Charlie
   IF turn < 0 AND board(endPnt, 3) = 1 THEN board(27, 3) = board(27, 3) + 1
   IF turn < 0 AND board(endPnt, 3) = 1 THEN board(endPnt, 3) = 0
   IF turn > 0 AND board(endPnt, 3) = - 1 THEN board(28, 3) = board(28, 3) - 1
   IF turn > 0 AND board(endPnt, 3) = - 1 THEN board(endPnt, 3) = 0
   board(endPnt, 3) = board(endPnt, 3) + turn          'add piece to end point
   board(stPnt, 3) = board(stPnt, 3) - turn       'subtract piece from start point
   FOR pnt = 1 TO 28                              'update points that changed
      ptX = board(pnt, 1) : ptY = board(pnt, 2) : pieces = board(pnt, 3)
      IF tempBoard(pnt) <> pieces THEN GOSUB DrawPoint 'redraw this point; it changed
      tempBoard(pnt) = board(pnt, 3)          'set up to move same piece again
   NEXT pnt
   RETURN
```

Figure 15-18. Sections 18, 19, 20, and 21: *LegalMove, GoodStart, MovePiece,* and *LegalEnd (continued)*

any point in the process, a test comes up invalid, a RETURN statement sends the program back to *MouseCheck* to wait for the next move. *LegalEnd* also takes care of bumping the opponent to the BAR if the player lands on a point holding only one of the opponent's pieces.

Finally, after the *LegalMove* and *Roll* subroutines have been repeated many times, the *DoubleClick* subroutine, shown in Figure 15-19, starts coming into play. If all conditions are correct, double clicking a piece removes it from a point and places it on the BAR. *DoubleClick* and *GoodDouble* check that the double click was in the proper area and that the piece is ready for removal. Then *TakeOff* removes the piece to the BAR and checks to see if 15 pieces belonging to one player are on the BAR, indicating a win. If so, the program branches to *Winner,* which is used once per game to announce which player won. The *InvertScreen* routine then flashes the screen until someone makes a menu selection to either start another game or quit.

```
REM Section 25--called by Section 17.
DoubleClick:
  stX = MOUSE(5) : stY = MOUSE(6)                    'X,Y location of double click
  IF stX >= board(7, 1) THEN RETURN                  'quick check for double click on left side
  IF turn > 0 AND stY >= board(24, 2) THEN RETURN    'double click in correct quad?
  IF turn < 0 AND stY < board(24, 2) THEN RETURN
  dbPnt = 0
  FOR pnt = 1 TO 6                                   'verify double click on point 1-6 or 19-24
    IF turn > 0 THEN ptX = board(pnt, 1) : ptY = board(pnt, 2)
    IF turn < 0 THEN ptX = board(pnt + 18, 1) : ptY = board(pnt + 18, 2)
    IF stX < ptX OR stX > ptX + pWidth THEN NextPnt4
    IF stY >= ptY AND stY <= ptY +bUp THEN dbPnt = pnt : pnt = 6
NextPnt4:
  NEXT pnt
  IF dbPnt = 0 THEN RETURN                           'didn't click on right point
  IF turn < 0 THEN dbPnt = dbPnt + 18                'adjust dbPnt for dark

REM Section 26--flows from Section 25.
GoodDouble:
  gdDble = 0
  IF turn > 0 THEN firstPnt = 7 ELSE firstPnt = 1
```

more...

Figure 15-19. Sections 25 through 28: *DoubleClick, GoodDouble, TakeOff, Winner,* and *InvertScreen*

```
  FOR pnt = firstPnt TO firstPnt + 17          'check for no pieces on point 1-18 or 7-24
    IF turn > 0 AND board(pnt, 3) > 0 THEN gdDble = 1 : pnt = firstPnt + 17
    IF turn < 0 AND board(pnt, 3) < 0 THEN gdDble = 1 : pnt = firstPnt + 17
  NEXT pnt
  IF gdDble = 1 THEN RETURN                    'all pieces must be in last quad before take off
  FOR dTry = 1 TO 4                            'check for start point to OFF on exact dice roll
    dValue = ABS(dice(dTry))
    IF turn > 0 AND dbPnt = dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
    IF turn < 0 AND 25 - dbPnt = dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
  NEXT dTry
  IF gdDble = 1 GOTO TakeOff                   'mother may I? Yes, you may take off 1 piece
  IF turn > 0 AND dbPnt = 6 THEN RETURN        'can only take off these pieces with 6
  IF turn < 0 AND dbPnt = 19 THEN RETURN
  IF turn > 0 THEN firstPnt = dbPnt + 1 : lastPnt = 6
  IF turn < 0 THEN firstPnt = 19 : lastPnt = dbPnt - 1
  FOR pnt = firstPnt TO lastPnt                'check for no pieces above double click
    IF turn > 0 AND board(pnt, 3) > 0 THEN gdDble = 1 : pnt = lastPnt
    IF turn < 0 AND board(pnt, 3) < 0 THEN gdDble = 1 : pnt = lastPnt
  NEXT pnt
  IF gdDble = 1 THEN RETURN                    'can't take this piece off yet
  FOR dTry = 1 TO 4                            'check to see if point to OFF < dice value
    dValue = ABS(dice(dTry))
    IF turn > 0 AND dbPnt < dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
    IF turn < 0 AND 25 - dbPnt < dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
  NEXT dTry
  IF gdDble = 1 GOTO TakeOff                   'go ahead and take it off
  RETURN

REM Section 27--called by Section 26.
TakeOff:
  board(dbPnt, 3) = board(dbPnt, 3) - turn     'subtract piece from double click point
  IF turn < 0 THEN board(26, 3) = board(26, 3) + turn    'add piece to OFF
  IF turn > 0 THEN board(25, 3)= board(25, 3) + turn
  FOR pnt = 1 TO 28                            'update board
    ptX = board(pnt, 1) : ptY = board(pnt, 2) : pieces = board(pnt, 3)
    IF tempBoard(pnt) <> pieces THEN GOSUB DrawPoint
    tempBoard(pnt) = board(pnt, 3)
  NEXT pnt
  IF turn > 0 AND board(25, 3) = 15 GOTO Winner    'we have a winner
  IF turn < 0 AND board(26, 3) = -15 GOTO Winner
  RETURN
```

Figure 15-19. Sections 25 through 28: *DoubleClick, GoodDouble, TakeOff, Winner,* and *InvertScreen (continued)*

more...

```
REM Section 28(a)--called by Section 27.
Winner:
   DIALOG OFF                                    'turn off dialog trapping but leave menu on
   CLS : BUTTON CLOSE 1 : BUTTON CLOSE 2         'completely clear screen
   rect(0) = wTop : rect(1) = wLeft : rect(2) = wBot : rect(3) = wRight    'use to invert screen
   TEXTSIZE 72                                   'BIG LETTERS
   IF turn > 0 THEN CALL MOVETO(150, 150) : PRINT "LIGHT"
   IF turn < 0 THEN CALL MOVETO(150, 150) : PRINT "DARK"
   MOVETO 150, 220 : PRINT "WINS"
   TEXTSIZE 12                                   'reset textsize

REM Section 28(b)--flows from Section 28(a).
InvertScreen:
   INVERTRECT VARPTR(rect(0))                    'invert whole screen, very fast
   GOTO InvertScreen                             'get out of loop with menu
```

Figure 15-19. Sections 25 through 28: *DoubleClick, GoodDouble, TakeOff, Winner,* and *InvertScreen (continued)*

If all this fun is too frivolous for your nature, after studying the complete backgammon listing in Figure 15-20 you can move on to the next section, which puts your Macintosh to work monitoring and controlling the world around you, and should be more to your liking.

```
REM Backgammon, written by Barry Preppernau
REM Section 1. Call three subroutines that set up game.
BeginGame:
   CLEAR
   GOSUB VariableInit
   GOSUB BoardInit
   GOSUB MenuInit
   GOTO StartGame

REM Section 2--called by Section 1. Define variables to be used.
VariableInit:
   DEFINT a-z
   RANDOMIZE TIMER
   DIM rect(3), poly(10), board(28,3), oldBoard(28), tempBoard(28)
```

Figure 15-20. The complete backgammon program

more...

```
DIM pointArray(1820), circleArray(64), diceImage(384)
DIM grey(3), black(3), white(3)
FOR x = 0 TO 3                          'integer values for grey, black, and white patterns
  grey(x) = -21931
  black(x) = -1
  white(x) = 0
NEXT x
dark = -30686                                              'pattern for dark points
light = -8841                                              'pattern for light points
pWidth = 34                                                'width of points
piWidth = 30                                               'width of pieces
wLeft = 0                                                  'window left side
wRight = 511                                               'window right side
wTop = 20                                                  'window top
wBot = 340                                                 'window bottom
boardRight = 409                                           'board right side
tUp = 0                                                    'top of upper points
bUp = 150                                                  'bottom of upper points
tLow = 170                                                 'top of lower points
lghtLeft = 425                                             'left side of light BAR and OFF
dkLeft = 465                                               'left side of dark BAR and OFF
offTop = 185                                               'top of OFF
barTop = 275                                               'top of BAR
FOR pnt= 1 TO 12          'X,Y for points on the board; board(pnt,1)=X, board(pnt,2)=Y
  board(pnt, 1) = 1 + (pnt - 1) * pWidth : board(pnt, 2) = tUp
  board(pnt + 12, 1) = 1 + (12 - pnt) * pWidth : board(pnt + 12, 2) = tLow
NEXT pnt
board(1, 3) = -2 : board(6, 3) = 5 : board(8, 3) = 3                    'initial locations
board(12, 3) = -5 : board(13,3) = 5 : board(17,3) =  -3                  '<0 black, >0 white
board(19,3) = -5 : board(24,3) = 2
board(25, 1) = lghtLeft : board(25, 2) = offTop : board(26, 1) = dkLeft
board(26, 2) = offTop : board(27, 1) = lghtLeft : board(27, 2) = barTop
board(28, 1) = dkLeft : board(28, 2) = barTop
'board(25, 3) = 10 : board(1, 3) = 5                                    'end game piece locations
'board(26, 3) = -10 : board(24, 3) = -5
RETURN

REM Section 3--called by Section 1. Set up board.
BoardInit:
  WINDOW 1, , (wLeft, wTop) - (wRight, wBot), 3                         'use whole screen
  GOSUB PointInit : CLS                                    'initialize images in pointArray
  GOSUB CircleInit : CLS                                  'circle outline to use to drag piece
  GOSUB DiceInit : CLS                                     'initialize images in dice array
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
   LINE (boardRight, 0) - (boardRight, 340)          'separate board from buttons and dice
   TEXTMODE 2                                                          'XOR all text
   BUTTON 1, 1, "Roll", (415, 100) - (455, 130)
   BUTTON 2, 1, "Undo", (465, 100) - (505, 130)
   rect(0) = 140 : rect(1) = 420 : rect(2) = 170 : rect(3) = 501
   FRAMERECT VARPTR(rect(0))                                           'OFF title box
   MOVETO 448, 160 : PRINT "OFF"
   rect(0) = 180 : rect(2) = 220
   FILLRECT VARPTR(rect(0)), VARPTR(grey(0))          'backgnd box for OFF pieces
   rect(0) = 230 : rect(2) = 260
   FRAMERECT VARPTR(rect(0))                                           'BAR title box
   MOVETO 448,250 : PRINT "BAR"
   rect(0) = 270 : rect(2) = 310
   FILLRECT VARPTR(rect(0)), VARPTR(grey(0))          'backgnd box for BAR pieces
   GOSUB BuildBoard                                      'put all points for first time
   GOSUB FirstRoll                          'roll one die each and decide whose turn
   RETURN

REM Section 15--called by Section 1. Create menu.
MenuInit:
   MENU 1, 0, 1, "Backgammon"
   MENU 1, 1, 1, "Start New Game"
   MENU 1, 3, 1, "Quit to BASIC"
   MENU 1, 4, 1, "Quit to Desktop"
   MENU 2, 0, 0, "" : MENU 3, 0, 0, "" : MENU 4, 0, 0, "" : MENU 5, 0, 0, ""
   ON MENU GOSUB MenuHandle : MENU ON
   RETURN

REM Section 29--called by menu-event trap.
MenuHandle:
   DIALOG STOP : MOUSE STOP
   menuBar = MENU(0) : menuItem = MENU(1)
   IF menuBar <> 1 THEN DIALOG ON : MOUSE ON : RETURN
   IF menuItem = 1 THEN MOUSE OFF : MENU OFF : DIALOG OFF : GOTO BeginGame
   IF menuItem = 3 THEN END
   IF menuItem = 4 THEN SYSTEM

REM Section 4--called by Section 3. Define variables used to create four kinds of points:
REM light up and down, and dark up and down.
PointInit:
   FOR points = 1 TO 4                                                 'four kinds of points
      IF points MOD 2 = 1 THEN pat = light                    'odd points light; up or down
      IF points MOD 2 = 0 THEN pat = dark                    'even points dark; up or down
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
    rect(0) = 50 : rect(1) = 33 : rect(2) = 200 : rect(3) = 67          'area to draw points
    poly(0) = 22 : poly(1) = 50 : poly(2) = 33 : poly(3) = 200          'set poly for up points
    poly(4) = 67 : poly(5) = 50 : poly(6) = 50 : poly(7) = 200
    poly(8) = 33 : poly(9) = 200 : poly(10) = 67
    IF points > 2 THEN poly(5) = 200 : poly(6) = 50 : poly(7) = 50      'down points
    IF points > 2 THEN poly(8) = 33 : poly(9) = 50 : poly(10) = 67
    GOSUB BuildPoint                                                    'draw that point
    GET (33, 50) - (66, 199), pointArray((points - 1) * 455)            'fill array with all points
  NEXT points
  RETURN

REM Section 5--called by Section 4. Draw points defined in Section 4.
BuildPoint:
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))                             'backgnd for all points
  FOR x= 0 TO 3 : pat(x) = pat : NEXT x                                 'set up pattern for actual point
  FILLPOLY VARPTR(poly(0)), VARPTR(pat(0))                              'draw point
  LINE(poly(6), poly(5)) - (poly(8), poly(7))                           'outline point
  LINE - (poly(10), poly(9)) : LINE - (poly(6), poly(5))
  RETURN

REM Section 6--called by Section 3. Draw and store circle used as playing piece.
CircleInit:
  CIRCLE (50, 50), 15                                                   'pieces have diameter of 30
  GET (35, 35) - (65, 65), circleArray                                  'use array to drag outline of piece
  RETURN

REM Section 7--called by Section 3. Draw and store view of each surface of die.
DiceInit:
  rect(0) = 35 : rect(1) = 35 : rect(2) = 66 : rect(3) = 66             '30x30 square dice
  FOR dice = 1 TO 6
    FILLRECT VARPTR(rect(0)), VARPTR(black(0))  'dice are black, invert for white dice
    IF dice MOD 2 = 1 THEN CIRCLE (50, 50), 1, 30                       'dot in center for odd dice
    IF dice > 1 THEN CIRCLE (40, 40), 1, 30 : CIRCLE (60, 60), 1, 30    'up left, low right
    IF dice > 3 THEN CIRCLE (60, 40), 1, 30 : CIRCLE (40, 60), 1, 30    'up right, low left
    IF dice = 6 THEN CIRCLE (40, 50), 1, 30 : CIRCLE (60, 50), 1, 30    'mid left, mid right
    GET (35, 35) - (65, 65), diceImage((dice - 1) * 64)                 'store all images in one array
  NEXT dice
  RETURN

REM Section 8--called by Section 3. Draw board and points.
BuildBoard:
  PENMODE 9
  rect(0) = bUp : rect(1) = wLeft : rect(2) = tLow : rect(3) = boardRight
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
    FILLRECT VARPTR(rect(0)), VARPTR(grey(0))   'fill in between upper and lower points
    rect(0) = wTop : rect(1) = wLeft : rect(2) = wBot : rect(3) = 1
    FILLRECT VARPTR(rect(0)), VARPTR(grey(0))                  'fill in left side of screen
    FOR pnt = 1 TO 28                    'draw all points; (ptX,ptY) = top left corner of point
      ptX = board(pnt, 1) : ptY = board(pnt,2) : pieces = board(pnt, 3)
      GOSUB DrawPoint
      tempBoard(pnt) = board(pnt, 3)     'use tempBoard to move one piece multiple times
    NEXT pnt
    RETURN

REM Section 9--called by Sections 8, 21, 24, and 27. Draw and refresh points.
DrawPoint:
    IF pnt > 24 GOTO BarAndOff    '25=lightOFF, 26=darkOFF, 27=lightBAR, 28=darkBAR
    IF pnt > 12 AND pnt MOD 2 = 0 THEN points = 0          'set up index into pointArray
    IF pnt > 12 AND pnt MOD 2 = 1 THEN points = 1
    IF pnt < 13 AND pnt MOD 2 = 0 THEN points = 2
    IF pnt < 13 AND pnt MOD 2 = 1 THEN points = 3
    PUT (ptX, ptY), pointArray(points * 455), PSET                'overlay point on board
    GOSUB DrawPiece                                              'draw pieces on this point
    RETURN

REM Section 10--called by Section 9. Draw pieces on points.
DrawPiece:
    IF pieces = 0 THEN RETURN                                  'don't need to draw zero pieces
    IF ABS(pieces) > 5 THEN show = 5 ELSE show = ABS(pieces) 'only show first 5 pieces
    FOR piece = 1 TO show                                                'draw up to 5 pieces
      IF pnt < 13 THEN rect(0) = (piece - 1) * piWidth          'set up rect for current piece
      IF pnt < 13 THEN rect(2) = (piece - 1) * piWidth + piWidth
      IF pnt < 13 THEN rect(1) = (pnt - 1) * pWidth + 3
      IF pnt < 13 THEN rect(3) = (pnt - 1) * pWidth + 33
      IF pnt > 12 THEN rect(0) = 320 - (piece - 1) * piWidth - piWidth
      IF pnt > 12 THEN rect(2) = 320 - (piece - 1) * piWidth
      IF pnt > 12 THEN rect(1) = (24 - pnt) * pWidth + 3
      IF pnt > 12 THEN rect(3) = (24 - pnt) * pWidth + 33
      IF pieces < 0 THEN FILLOVAL VARPTR(rect(0)), VARPTR(black(0))      'draw black
      IF pieces > 0 THEN FILLOVAL VARPTR(rect(0)), VARPTR(white(0))      'draw white
    NEXT piece
    IF ABS(pieces) <= 5 THEN RETURN                'did we draw all pieces on this point?
    TEXTSIZE 10                                  'write number of pieces stacked on top of:
    IF pnt < 13 THEN CALL MOVETO ((pnt - 1) * pWidth + 6, 20)          'top piece for upper
    IF pnt > 12 THEN CALL MOVETO((24 - pnt) * pWidth + 6, 307)  'bottom piece for lower
    PRINT ABS(pieces) - 5 : TEXTSIZE 12
    RETURN
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
REM Section 11--called by Section 9. Draw BAR and OFF areas and any pieces on them.
BarAndOff:
   IF pnt = 25 OR pnt = 26 THEN rect(0) = 180 : rect(2) = 220                'backgnd rect
   IF pnt = 27 OR pnt = 28 THEN rect(0) = 270 : rect(2) = 310
   IF pnt = 25 OR pnt = 27 THEN rect(1) = 420 : rect(3) = 460
   IF pnt = 26 OR pnt = 28 THEN rect(1) = 460 : rect(3) = 501
   FILLRECT VARPTR(rect(0)), VARPTR(grey(0))
   IF pieces = 0 THEN RETURN                          'don't need to draw zero pieces
   rect(0) = ptY : rect(1) = ptX : rect(2) = ptY + piWidth : rect(3) = ptX + piWidth
   IF pnt = 25 OR pnt = 27 THEN FILLOVAL VARPTR(rect(0)), VARPTR(white(0))
   IF pnt = 26 OR pnt = 28 THEN FILLOVAL VARPTR(rect(0)), VARPTR(black(0))
   IF pieces < 10 THEN CALL MOVETO(ptX + 3, ptY + 20)
   IF pieces > 9 THEN CALL MOVETO(ptX - 1, ptY + 20)
   PRINT ABS(pieces)                          'write number of pieces OFF or on BAR
   RETURN

REM Section 12--called by Section 3. Determine who starts.
FirstRoll:
   dice(1) = INT(RND(1) * 6 + 1) * -1 : dice(2) = INT(RND(1) * 6 + 1)  '(1)= black, (2)=white
   IF ABS(dice(1)) = dice(2) GOTO FirstRoll                'can't have doubles on first roll
   IF ABS(dice(1)) > dice(2) THEN turn = -1 ELSE turn = 1     'turn<0 for dark, >0 for light
   oldDice(1) = dice(1) : oldDice(2) = dice(2)                      'use oldDice for Undo
   GOSUB WhoseTurn                                                  'draw turn title box
   GOSUB DrawDice
   FOR pnt = 1 TO 28                                'record current board set up for Undo
      oldBoard(pnt) = board(pnt, 3) : tempBoard(pnt) = board(pnt, 3)
   NEXT pnt
   RETURN

REM Section 13(a)--called by Sections 12 and 23. Determine and announce turn.
WhoseTurn:
   rect(0) = 1 : rect(1) = 411 : rect(2) = 30 : rect(3) = 510              'rect for turn title box
   IF turn > 0 GOTO LightTurn
   FILLRECT VARPTR(rect(0)), VARPTR(black(0))                               'draw black box
   MOVETO 425, 20 : PRINT "Dark's Move"
   RETURN

REM Section 13(b)--called by Section 13(a).
LightTurn:
   FILLRECT VARPTR(rect(0)), VARPTR(white(0))          'draw white box to clear rect
   FRAMERECT VARPTR(rect(0))                                          'frame title box
   MOVETO 420, 20 : PRINT "Light's Move"
   RETURN
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
REM Section 14(a)--called by Sections 12 and 23. Draw black dice and invert for white.
DrawDice:
  rect(0) = 40 : rect(1) = 420 : rect(2) = 80 : rect(3) = 501          'backgnd for dice
  FILLRECT VARPTR(rect(0)), VARPTR(grey(0))
  FOR dice = 1 TO 2                                                    'draw both dice
    IF dice = 1 THEN left = lghtLeft ELSE left = dkLeft                'left edge of dice
    PUT (left, 45), diceImage((ABS(dice(dice)) - 1) * 64), PSET 'show correct dice image
    IF dice(dice) < 0 GOTO NextDice          'dice is supposed to be black, draw next one
    rect(0) = 45 : rect(1) = left : rect(2) = 76 : rect(3) = left + 31
    INVERTRECT VARPTR(rect(0))                          'have to invert for white dice

REM Section 14(b)--called by Section 14(a).
NextDice:
  NEXT dice
  RETURN

REM Section 16--called by Section 1.
StartGame:
  ON DIALOG GOSUB DialogHandle : DIALOG ON

REM Section 17--flows from Section 16. Wait for mouse action (main loop).
MouseCheck:
  WHILE MOUSE(0) = 0 : WEND
  FOR x =1 TO 600 : NEXT x                              'wait to check for double click
  mousePush = MOUSE(0)
  IF mousePush = 1 GOTO MouseCheck                      'don't care about single clicks
  IF mousePush < -1 GOTO MouseCheck                     'or double or triple clicks/drags
  IF mousePush = -1 THEN GOSUB LegalMove    'trying to drag piece, check for legality
  IF mousePush > 1 THEN GOSUB DoubleClick 'trying to take piece off, check for legality
  GOTO MouseCheck                                       'loop until someone wins

REM Section 22--called by dialog-event trap.
DialogHandle:
  MENU STOP : MOUSE STOP
  event = DIALOG(0)
  IF event <>1 THEN MENU ON : MOUSE ON : RETURN  'don't care if no button pushed
  buttonNumber = DIALOG(1)
  IF buttonNumber = 1 THEN GOSUB Roll
  IF buttonNumber = 2 THEN GOSUB Undo
  MENU ON : MOUSE ON
  RETURN
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
REM Section 23--called by Section 22.
Roll:
  turn = turn * -1                                                    'change turns
  dice(3) = 0 : dice(4) = 0                                  'dice(3) and (4) used for doubles
  dice(1) = INT(RND(1) * 6 + 1) * turn : dice(2) = INT(RND(1) * 6 + 1) * turn
  IF dice(1) = dice(2) THEN dice(3) = dice(1) : dice(4) = dice(1)      'oh boy, doubles!!!
  FOR x = 1 TO 4 : oldDice(x) = dice(x) : NEXT x                    'oldDice used for Undo
  GOSUB WhoseTurn                                                     'draw turn title box
  GOSUB DrawDice
  FOR pnt = 1 TO 28                                  'record current board set up for Undo
    oldBoard(pnt) = board(pnt, 3) : tempBoard(pnt) = board(pnt, 3)
  NEXT pnt
  RETURN

REM Section 24--called by Section 22.
Undo:
  FOR pnt = 1 TO 28                              'reset points that have changed in this turn
    tempBoard(pnt) = board(pnt, 3)                        'record current number of pieces
    board(pnt, 3) = oldBoard(pnt)                                  'reset to start of turn
    ptX = board(pnt, 1) : ptY = board(pnt, 2) : pieces = board(pnt, 3)
    IF pieces <> tempBoard(pnt) THEN GOSUB DrawPoint   'if any change redraw point
    tempBoard(pnt) = board(pnt, 3)        'tempBoard used for multiple moves of 1 piece
  NEXT pnt
  FOR x = 1 TO 4 : dice(x) = oldDice(x) : NEXT x                          'reset dice
  RETURN

REM Section 18--called by Section 17. Check if move is legal.
LegalMove:
  stX = MOUSE(3) : stY = MOUSE(4) : stPnt = 0              'starting X,Y position of drag
  FOR pnt = 1 TO 24                                'quick check for starting from legal point
    ptX = board(pnt, 1) : ptY = board(pnt, 2)
    IF stX < ptX OR stX > ptX + pWidth THEN NextPnt1
    IF stY >= ptY AND stY <= ptY + bUp THEN stPnt = pnt : pnt = 24
NextPnt1:
  NEXT pnt
  IF stPnt <> 0 GOTO GoodStart                          'started drag on board point 1-24
  FOR pnt = 27 TO 28                                      'check for dragging from BAR
    ptX = board(pnt, 1) :  ptY = board(pnt, 2)
    IF stX < ptX OR stX > ptX + piWidth THEN NextPnt2
    IF stY >= ptY AND stY <= ptY + piWidth THEN stPnt = pnt : pnt = 28
NextPnt2:
  NEXT pnt
  IF stPnt = 0 THEN RETURN                  'can't drag from anyplace but 1-24 or BAR
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
REM Section 19--called by Section 18. Check for valid starting place.
GoodStart:
  IF stPnt < 25 AND turn = -1 AND board(28, 3) <> 0 THEN RETURN          'still on BAR
  IF stPnt < 25 AND turn = 1 AND board(27, 3) <> 0 THEN RETURN
  startPiece = board(stPnt, 3)
  IF startPiece = 0 THEN RETURN                    'can't drag from point with no pieces
  IF turn < 0 AND startPiece > 0 THEN RETURN              'can't drag opponent's piece
  IF turn > 0 AND startPiece < 0 THEN RETURN
  PUT (stX - 15, stY - 15), circleArray          'OK to drag, put circle outline under pointer

REM Section 20--flows from Section 19.
MovePiece:
  mousePush = MOUSE(0) : endX = MOUSE(5) : endY = MOUSE(6)
  IF mousePush <> -1 GOTO LegalEnd                'stopped dragging, check for legality
  PUT (stX - 15, stY - 15), circleArray                  'erase outline at old location
  PUT (endX - 15, endY - 15), circleArray                'draw outline at new location
  stX = endX : stY = endY                              'use stX, stY to erase next time
  GOTO MovePiece                                          'loop until stop dragging

REM Section 21--called by Section 20.
LegalEnd:
  PUT (stX - 15, stY - 15), circleArray                          'erase at last location
  endPnt = 0
  FOR pnt = 1 TO 24                                  'quick check for legal ending point
    ptX = board(pnt, 1) : ptY = board(pnt, 2)
    IF endX < ptX OR endX > ptX + pWidth THEN NextPnt3
    IF endY >= ptY AND endY <= ptY + bUp THEN endPnt = pnt : pnt = 24
NextPnt3:
  NEXT pnt
  IF endPnt = 0 THEN RETURN                                  'have to end on point 1-24
  endPiece = board(endPnt, 3)
  IF turn < 0 AND endPiece > 1 THEN RETURN      'can't end on point with >1 opponent
  IF turn > 0 AND endPiece < -1 THEN RETURN
  gdEnd = 0
  FOR dTry = 1 TO 4                          'check for moving correct number of points
    dValue = ABS(dice(dTry))
    IF stPnt = 27 AND endPnt = 25 - dValue THEN gdEnd = 1 : dice(dTry) = 0 : dTry = 4
    IF stPnt = 28 AND endPnt = dValue THEN gdEnd = 1 : dice(dTry) = 0 : dTry = 4
    IF stPnt + dValue * turn * - 1 = endPnt THEN gdEnd = 1 : dice(dTry) = 0 : dTry = 4
  NEXT dTry
  IF gdEnd = 0 THEN RETURN                                              'sorry Charlie
  IF turn < 0 AND board(endPnt, 3) = 1 THEN board(27, 3) = board(27, 3) + 1
  IF turn < 0 AND board(endPnt, 3) = 1 THEN board(endPnt, 3) = 0
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
   IF turn > 0 AND board(endPnt, 3) = - 1 THEN board(28, 3) = board(28, 3) - 1
   IF turn > 0 AND board(endPnt, 3) = - 1 THEN board(endPnt, 3) = 0
   board(endPnt, 3) = board(endPnt, 3) + turn                    'add piece to end point
   board(stPnt, 3) = board(stPnt, 3) - turn                      'subtract piece from start point
   FOR pnt = 1 TO 28                                             'update points that changed
      ptX = board(pnt, 1) : ptY = board(pnt, 2) : pieces = board(pnt, 3)
      IF tempBoard(pnt) <> pieces THEN GOSUB DrawPoint  'redraw this point; it changed
      tempBoard(pnt) = board(pnt, 3)                             'set up to move same piece again
   NEXT pnt
   RETURN

REM Section 25--called by Section 17.
DoubleClick:
   stX = MOUSE(5) : stY = MOUSE(6)                               'X,Y location of double click
   IF stX >= board(7, 1) THEN RETURN                  'quick check for double click on left side
   IF turn > 0 AND stY >= board(24, 2) THEN RETURN       'double click in correct quad?
   IF turn < 0 AND stY < board(24, 2) THEN RETURN
   dbPnt = 0
   FOR pnt = 1 TO 6                                   'verify double click on point 1-6 or 19-24
      IF turn > 0 THEN ptX = board(pnt, 1) : ptY = board(pnt, 2)
      IF turn < 0 THEN ptX = board(pnt + 18, 1) : ptY = board(pnt + 18, 2)
      IF stX < ptX OR stX > ptX + pWidth THEN NextPnt4
      IF stY >= ptY AND stY <= ptY +bUp THEN dbPnt = pnt : pnt = 6
NextPnt4:
   NEXT pnt
   IF dbPnt = 0 THEN RETURN                                      'didn't click on right point
   IF turn < 0 THEN dbPnt = dbPnt + 18                           'adjust dbPnt for dark

REM Section 26--flows from Section 25.
GoodDouble:
   gdDble = 0
   IF turn > 0 THEN firstPnt = 7 ELSE firstPnt = 1
   FOR pnt = firstPnt TO firstPnt + 17            'check for no pieces on point 1-18 or 7-24
      IF turn > 0 AND board(pnt, 3) > 0 THEN gdDble = 1 : pnt = firstPnt + 17
      IF turn < 0 AND board(pnt, 3) < 0 THEN gdDble = 1 : pnt = firstPnt + 17
   NEXT pnt
   IF gdDble = 1 THEN RETURN          'all pieces must be in last quad before take off
   FOR dTry = 1 TO 4                  'check for start point to OFF on exact dice roll
      dValue = ABS(dice(dTry))
      IF turn > 0 AND dbPnt = dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
      IF turn < 0 AND 25 - dbPnt = dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
   NEXT dTry
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
  IF gdDble = 1 GOTO TakeOff              'mother may I? Yes, you may take off 1 piece
  IF turn > 0 AND dbPnt = 6 THEN RETURN   'can only take off these pieces with 6
  IF turn < 0 AND dbPnt = 19 THEN RETURN
  IF turn > 0 THEN firstPnt = dbPnt + 1 : lastPnt = 6
  IF turn < 0 THEN firstPnt = 19 : lastPnt = dbPnt - 1
  FOR pnt = firstPnt TO lastPnt           'check for no pieces above double click
    IF turn > 0 AND board(pnt, 3) > 0 THEN gdDble = 1 : pnt = lastPnt
    IF turn < 0 AND board(pnt, 3) < 0 THEN gdDble = 1 : pnt = lastPnt
  NEXT pnt
  IF gdDble = 1 THEN RETURN                'can't take this piece off yet
  FOR dTry = 1 TO 4                        'check to see if point to OFF < dice value
    dValue = ABS(dice(dTry))
    IF turn > 0 AND dbPnt < dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
    IF turn < 0 AND 25 - dbPnt < dValue THEN gdDble = 1 : dice(dTry) = 0 : dTry = 4
  NEXT dTry
  IF gdDble = 1 GOTO TakeOff               'go ahead and take it off
  RETURN

REM Section 27--called by Section 26.
TakeOff:
  board(dbPnt, 3) = board(dbPnt, 3) - turn  'subtract piece from double click point
  IF turn < 0 THEN board(26, 3) = board(26, 3) + turn   'add piece to OFF
  IF turn > 0 THEN board(25, 3)= board(25, 3) + turn
  FOR pnt = 1 TO 28                                     'update board
    ptX = board(pnt, 1) : ptY = board(pnt, 2) : pieces = board(pnt, 3)
    IF tempBoard(pnt) <> pieces THEN GOSUB DrawPoint
    tempBoard(pnt) = board(pnt, 3)
  NEXT pnt
  IF turn > 0 AND board(25, 3) = 15 GOTO Winner         'we have a winner
  IF turn < 0 AND board(26, 3) = -15 GOTO Winner
  RETURN

REM Section 28(a)--called by Section 27.
Winner:
  DIALOG OFF                          'turn off dialog trapping but leave menu on
  CLS : BUTTON CLOSE 1 : BUTTON CLOSE 2   'completely clear screen
  rect(0) = wTop : rect(1) = wLeft : rect(2) = wBot : rect(3) = wRight   'use to invert screen
  TEXTSIZE 72                                           'BIG LETTERS
  IF turn > 0 THEN CALL MOVETO(150, 150) : PRINT "LIGHT"
  IF turn < 0 THEN CALL MOVETO(150, 150) : PRINT "DARK"
```

Figure 15-20. The complete backgammon program *(continued)*

more...

```
    MOVETO 150, 220 : PRINT "WINS"
    TEXTSIZE 12                                          'reset textsize

REM Section 28(b)--flows from Section 28(a).
InvertScreen:
    INVERTRECT VARPTR(rect(0))                    'invert whole screen, very fast
    GOTO InvertScreen                               'get out of loop with menu
```

Figure 15-20. The complete backgammon program *(continued)*

SECTION V

Data Acquisition and Control

Introduction to Data Acquisition and Control

Chapter 16

In Chapter 15, you worked with a communication program that allowed you to connect your Macintosh to another computer, either directly or over the telephone lines, and exchange information. It is a general-purpose communication program: You can use it to call Dow Jones News/Retrieval Service, DIALOG, CompuServe, or a friend down the street. With a few refinements, it can automatically log you on to a database service, taking care of such routine matters as entering your identification number and password.

The programs we will develop in this section are specialized communication programs. They use the same BASIC commands to route information in and out the modem port, but rather than communicate with friends or a variety of database services, they communicate with only one specialized device—an analog-to-digital (A/D) converter—and perform a special function: data acquisition and control.

The phrase "data acquisition and control" is somewhat mysterious and intimidating. It sounds like something that is done by people who wear white coats and keep rats in cages—and it is. But it's also done by each and every one of us every day of our lives. You are driving down the street and the traffic light turns red: You put your foot on the brake. You are cold: You turn up the heat. It starts to rain: You put up an umbrella.

These are all commonplace activities that you do without a moment's thought, but in each case you acquire some data, evaluate it, make a decision, and take action. The "input devices" used to acquire this data are your senses: sight, hearing, smell, taste, and touch. Your brain evaluates the data and makes a decision based on the memory of a previous experience or on information provided by someone else, and then controls your muscles to convert thoughts into actions.

Your Macintosh is probably not going to drive your car or hoist your umbrella; but, with the proper interface to the "real world," it can enhance your senses and extend them and your muscles into places and times where it isn't convenient for you personally to take action. There are many readily available, low-cost devices that can represent some aspect of their environment as a small electrical signal. And there are interface devices, called analog-to-digital converters, that can transform these electrical signals into a digital format that your Macintosh can understand.

All A/D converters perform essentially the same task, but they do it with varying degrees of speed and accuracy, and provide various options. The primary purpose, as their name implies, is to convert variations in natural phenomena, such as temperature, pressure, strain, movement, light, and moisture, to a number that can be understood by a computer. Amounts or changes in these phenomena are typically measured as analog quantities—numbers that vary smoothly in infinitely small increments through a continuous range from a low point to a high point, or vice versa. For example, at the end of the day the level of light fades gradually after sunset, passing almost imperceptibly through dusk to dark. The computer, on the other hand, can handle only digital numbers at precise intervals over a clearly specified range. Let's look at a hypothetical situation in which your Macintosh is connected to an A/D converter that is monitoring wind speed. You are an apple grower, and the speed of the wind is more than a matter of mild curiosity for you. At certain stages in its development, your fruit is susceptible to severe damage due to low temperatures, and the effect of temperature is related to wind speed. You have your orchard wired to monitor temperature and wind speed at various locations, and your Macintosh has been programmed to decide when the critical combination is being approached and to take preventive action to save your apples. (Strangely, preventive action, at least in the state of Washington, involves spraying the trees with a fine mist of water, which freezes, forming a protective coat over the budding fruit and maintaining its temperature at 32 degrees, several degrees above the point at which damage occurs.)

The process of converting wind speed to information on the screen of your Mac follows the flow chart shown in Figure 16-1. The speed of the wind is monitored by a transducer, which is a device that converts the magnitude of a measured phenomenon to a voltage, current, resistance, or number of pulses within a certain time period that is proportional to the magnitude. The output of the transducer is fed into a signal-conditioning circuit (generally an amplifier), where it is converted to a voltage within

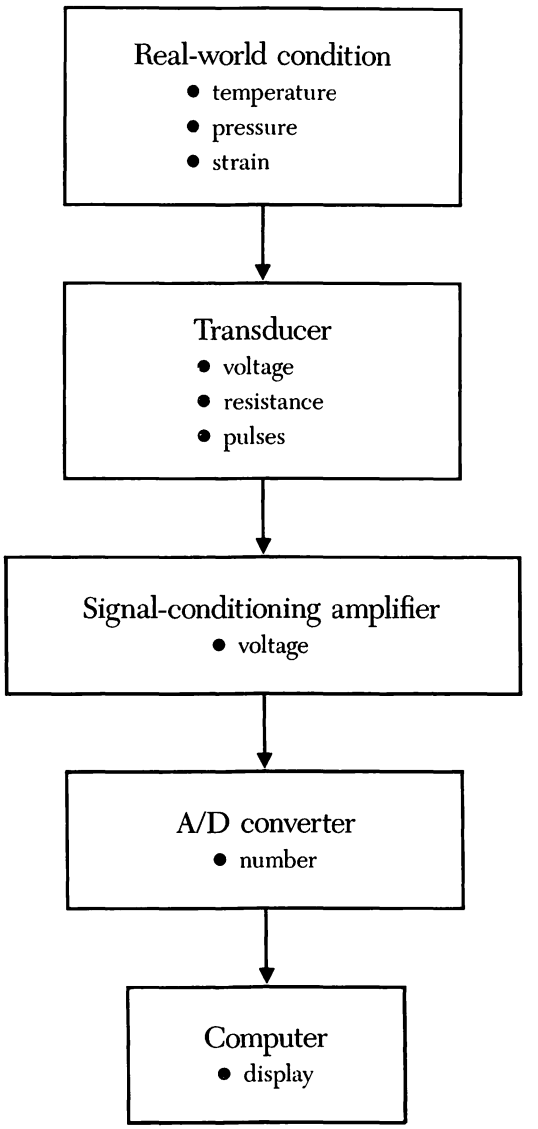


Figure 16-1. Flow chart of analog-to-digital conversion process

a specified range. This voltage is in turn fed into the A/D converter, which produces an integer output proportional to the point in the allowable input range where the present voltage falls.

The transducer used to monitor wind speed is called an anemometer. You have undoubtedly seen these in home weather stations and at the top of boat masts—three little cups spinning in the breeze. One brand produces 2.94 millivolts for each mile per hour of wind speed. With this model, if you measure the wind speeds from 0 to 100 mph, you will expect the anemometer to produce from 0 to 294 millivolts. If the signal-conditioning amplifier transforms the transducer's output to a range of 0 to 5 volts (that is, 0 volts from the transducer equals 0 volts going into the A/D converter; 294 millivolts from the transducer equals 5 volts into the converter) and the wind is blowing at 50 mph, then the transducer produces 147 millivolts and the A/D converter receives 2.5 volts. If the integer output of the A/D converter ranges from 0 to 255, then the output with a 50 mph wind will be 127. With your Macintosh connected to the converter, you can have a BASIC program ask the converter for the output of the channel monitoring the anemometer, and the response will be 127. The program would presumably include a formula that multiplies the response from the converter by 0.3921568 (100 mph divided by 255 steps) in order to compute the speed of the wind. Once the multiplication is done, the Mac will print on its screen that the wind speed is 49.8039 miles per hour. The slight inaccuracy in this conversion is due to the maximum resolution possible with this particular converter.

Most A/D converters currently on the market offer resolution in the range of 8 to 16 bits. This number is the number of data bits used to send the value of the converted voltage to your computer, and determines its highest possible degree of accuracy. Figure 16-2 shows you how the data is passed, and the precision provided by different numbers of bits. You can see that if the converter can pass 8 bits of information, it is capable of passing a number between 0 and 255. With the signal-conditioning amplifier properly balanced to the full range of values you want to measure, the lowest value you want to measure should correspond to a 0, and the highest to a 255.

Remembering that the output of the converter is always a whole number (an integer), each step in the range between the highest and lowest numbers returned by the transducer/signal-conditioning amplifier/converter combination is determined according to the following formula:

$$\text{One measurement step} = \frac{(\text{highest value} - \text{lowest value})}{(2 \text{ ^ number of bits of resolution})}$$

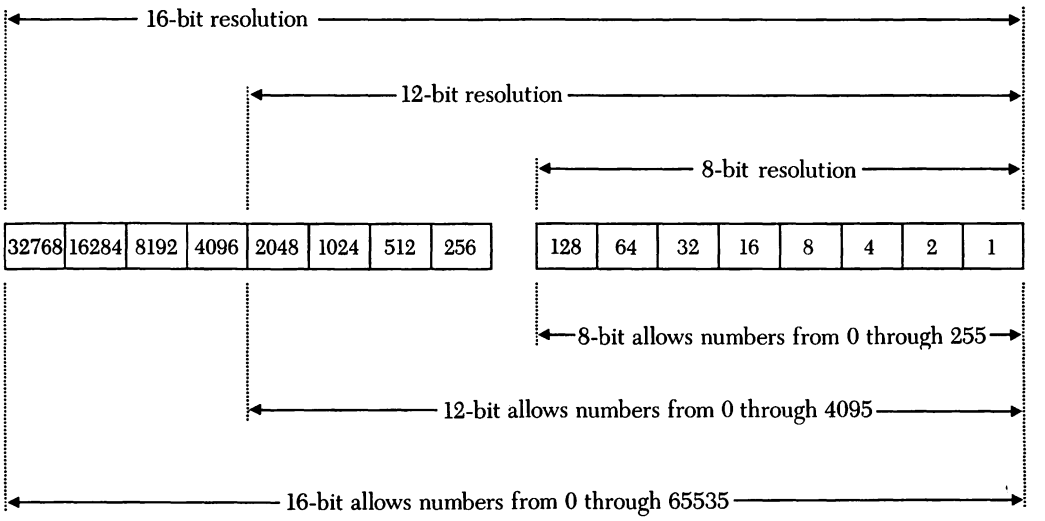


Figure 16-2. Bits passed

As an example, if you want to measure temperatures that range from 32 to 160 degrees—a convenient range of 128 degrees—the minimum temperature variation an 8-bit converter is capable of distinguishing is:

$$\text{One measurement step} = \frac{(160-32)}{256} = 0.5 \text{ degrees per step}$$

In other words, each degree measured by the transducer is equivalent to a change of two steps in the number returned by the converter. If you were to switch to a 12-bit converter, the formula would be:

$$\text{One measurement step} = \frac{(160-32)}{4096} = 0.03125 \text{ degrees per step}$$

So each degree measured by the transducer is equivalent to a change of 32 steps in the number returned by the converter.

In the wind-speed example, if we were measuring a wind speed of 50 mph (corresponding to a level of 2.5 volts input to the A/D converter and output of half the

range of the converter), the following table shows the increase in accuracy that accompanies greater resolution:

Resolution (bits)	Minimum measurement (mph per step)	A/D converter output	Resulting measurement (mph)
8	0.392	127	48.8039
12	0.0244	2047	49.9756
16	0.001525	32767	49.9984

Of course, in all such measurements it is important to remember that you can't expect the answer to be any more accurate than the least accurate element in the chain of measurements and computations that leads to it. The weakest element in measurement systems is usually the transducer: Unless you are willing to spend quite a bit of money, when the readout on your Mac says the wind is blowing at 49.9756 mph, you had better settle in your mind for "about 50."

Another factor that can affect the accuracy of the answer is the sampling rate of the A/D converter. The maximum sampling rate is determined by several things—a major one being the time required by the A/D converter to actually convert the input voltage to an output number, which ranges from about 10 microseconds up to about 1 second for converters I have investigated. The baud rate of the communication link between the converter and computer, and the speed and efficiency of the computer program also affect overall system performance. A high sampling rate is obviously important when measuring something that fluctuates fairly rapidly in value, such as the human voice, than when measuring something relatively stable, such as temperature. Another advantage of a high sampling rate is that your Macintosh can compute the average value during each 1-second time period, based on several hundred samples, rather than having to rely on the value at a precise instant in that time period, which is what you get with one sample.

There are many devices that can be hooked to the Macintosh to allow it to monitor or control external events. As discussed in Section III, practically anything designed to be connected to a serial port that complies with the RS-232C standard can also be connected to one of the Macintosh serial ports, which follow the RS-422 standard. Beyond the actual connection, there is very little standardization, either in what

the devices do or how they do it. Some have built-in software that determines the type of event they can monitor; others are more general in nature, simply converting a signal from the sensor and passing it along to the Macintosh. In the latter case, a program must be loaded into the Macintosh to match the input from the sensor to an appropriate formula to compute temperature, wind speed, widgets per minute on the assembly line, or whatever the particular sensor is monitoring.

Many A/D converters have been built up into systems that offer more than simply analog-to-digital conversion. They often monitor digital inputs, which allows them to inform you when the water or temperature exceeds a certain level, or when a burglar breaks through your back door. Many allow your computer to close and open relay contacts built into the converter, in order to turn on and off electrical devices, such as lights, motors, heaters, and sirens.

Chapter 17 describes a specific commercially available converter, the ADC-1 from Remote Measurement Systems, and develops a BASIC program to communicate with it, evaluate the information received, and produce an output. This particular program monitors temperature, but with minor changes it could just as easily monitor the security of your house, the energy consumption of your company, the soil conditions in your garden, or the processes that go into making a product. And monitoring is only one side of the coin: A few more lines of BASIC program and your Macintosh could turn on an alarm, shut off the air conditioning, water your garden, or waylay a widget that doesn't meet specifications.

Chapter 18 describes the HBC-1, a low-cost A/D converter that can be built by anyone with a little experience assembling electronic kits—or a sense of adventure—and develops a program to communicate with it. This program uses the HBC-1 to measure a voltage, and displays the result on the Macintosh screen as a digital readout, a bar chart, and a conventional voltmeter dial.

If the description of the HBC-1 catches your interest, you can turn to Appendix C for technical details, schematics, and assembly instructions. You will find additional information about data acquisition and control in the books and magazines listed in the following bibliography.

Bibliography

Analog Devices, Inc. *Analog-Digital Conversion Handbook.* 1976.

Carr, J.J. *Microcomputer Interfacing Handbook: A/D and D/A.* Tab Books, Inc., 1980.

Ciarcia, S. "Analog Interfacing in the Real World." *BYTE,* February 1985.

Ciarcia, S. "Control the World! (Or at Least a Few Analog Points)." *BYTE,* September 1977.

Englemann, B. and M. Abraham. "Personal Computer Signal Processing." *BYTE,* April 1984.

Garrett, P. *Analog I/O Design.* Reston Publishing, 1981.

Genet, R.M., L.J. Boyd, and D.J. Sauer. "Interfacing for Real-Time Control." *BYTE,* April 1984.

Hallgren, R. "Putting the Apple II Work." *BYTE,* April 1984.

Hnatek, E. *A User's Handbook of D/A and A/D Converters.* Wiley-Interscience, 1976.

Hybrid Systems Corporation. *Digital-to-Analog Converter Handbook.* 1970.

Sheingol, D. *Transducer Interfacing Handbook: A Guide to Analog Signal Conditioning.* Analog Devices, Inc. 1980.

VandenHeede, T.M. "A/D Conversion Brings the Real World to the Personal Computer." *Personal Computing,* June 1982.

Wyss, C. "Planning a Computerized Measurement System." *BYTE,* April 1984.

The ADC-1

Chapter 17

Since the purpose of this chapter is to give a few examples of what a BASIC program can do with the information provided by a monitoring device, rather than to teach you about all such devices, I have picked one very flexible, general-purpose device on which to base my program. Later in this chapter I will describe a few other devices and tell you how they differ from this one. The device I have chosen is the ADC-1, manufactured by Remote Measurement Systems, 2633 Eastlake Avenue East, Seattle, WA 98102.

Converter features

Besides the fact that the manufacturer was willing to loan me one and help me understand it, I chose the ADC-1 because it has the following features:

- Sixteen analog inputs capable of measuring varying voltages.
- Analog-signal resolution of 12 bits.
- Optional instrumentation amplifier.
- Four digital inputs capable of sensing an on/off condition.
- On/off control of six outputs to operate external devices.
- Line carrier control.
- A computer link that is compatible with RS-232 devices.
- Ease of control by a BASIC program.
- Relatively low cost.

On the chance that some of these features are as meaningless to you as they were to me the first time I read about them, I will explain them one by one.

Analog inputs

Each analog input to the ADC-1 can convert the varying voltage produced by a transducer monitoring some real-world condition, such as temperature or amount of electricity consumed, to an integer number the Macintosh can understand. This allows your computer to track changes in the value being monitored, and compute the rate of change or some other value, such as a projected electrical bill based on the current rate of consumption. Conditions typically monitored by this section of the ADC-1 are speed, temperature, distance, light level, sound, electrical current, pressure, and radioactivity.

Although the ADC-1 has sixteen channels, it monitors only one at a time—and waits to do that until you tell it, via the computer program, which channel you want checked. This is a form of multiplexing that is common to most multi-channel A/D converters.

Resolution

The concept of resolution was fairly well covered in the last chapter. The 12-bit resolution of the ADC-1 puts it about mid-range in accuracy.

Instrumentation amplifier

The instrumentation amplifier is an optional add-on used to amplify extremely low-level signals from devices such as thermocouples, which typically produce 40 microvolts per degree measured.

Digital inputs

As discussed in other chapters, the Macintosh performs its internal manipulations of letters and numbers by first converting them to binary digits (bits). Bits are always in one of two states, usually represented by such mutually exclusive terms as on or off, 1 or 0, true or false, yes or no, and plus or minus. The conditions reported by the digital-input section of the ADC-1 are also mutually exclusive: It tells you that a switch is in one of two states—either open or closed. Sensors typically connected to

the digital inputs are thermostatic switches, photoelectric switches, magnetic contacts, relays, pressure switches, mechanical counters, vibration sensors, and water-level switches. When a sensor detects the absence or presence of the condition it is monitoring, it notifies the ADC-1 by opening or closing the connection between itself and the ADC-1.

The events that can be monitored by the four digital inputs are determined more by your reason for monitoring than by the event being monitored. If you must know a precise value at every moment, then you have to use an analog input, but if you really only have to know if a certain value has been passed, the digital inputs will work fine. Typical uses for digital monitoring are turning the heater on when the temperature drops below a set level, turning the lights on when the light level drops to a certain point, sounding an alarm if a window or door is opened, calling you at home if the electricity goes off at your cabin. In each of these cases, a yes/no status is adequate. You don't care how far above or below the critical point the temperature or light is; it is when the critical point is passed that the necessary action must be taken. Likewise, you don't care if a burglar opens your door a foot or a yard, or leaves it open for a second or all day—you just want to know if the door is open when it shouldn't be.

Controlled outputs

The six controlled outputs are switches that your Macintosh can be programmed to turn on or off by sending a signal to the ADC-1. These switches can be used, either directly or through a relay, to control other devices, such as a motor, lights, an alarm, or the heater for your hot tub.

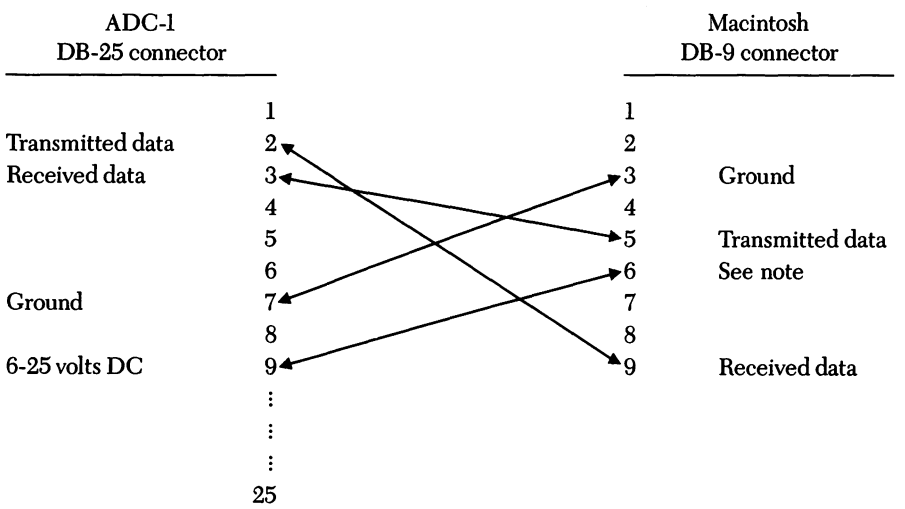
Line carrier control

Line carrier control is an interesting and easy method of controlling electrical devices around a building without having to run control wiring to them. This is accomplished by placing a coded high-frequency control signal on the AC power line that supplies voltage to the electrical outlets in the building. This signal has no effect on the equipment you normally plug into these outlets (including your computer), but line-carrier receivers, which you can plug or wire into the house wiring, can decode the signal and react to specific instructions to open or close a switch. Line carrier receivers are marketed under a variety of names—the most common is BSR—and come in configurations designed to control almost anything you can imagine.

This feature lends itself nicely to home-control projects, such as automatically turning lights or appliances on and off at certain times or in response to certain situations (as when someone walks into the room).

The communication link

The communication connection on the ADC-1, though not strictly in keeping with RS-232 standards, has the essentials where expected on pins 2, 3, and 7 (transmitted data, received data, and ground) of its DB-25 connector. Communication between the Mac and the ADC-1 is just like communication between the Macintosh and a modem or another computer. Figure 17-1 shows the wiring diagram for a cable between the Macintosh communication port and the ADC-1.



NOTE: Pin 6 on the Mac supplies 12 volts DC and seems to power the ADC-1 nicely. However, there is a limit to the amount of power available here to drive relays and other devices that might be connected to the ADC-1, so you may want to power it from a battery or an external power supply. This also avoids the possibility of damaging the Mac by connecting the power line incorrectly.

Figure 17-1. Wiring for the ADC-1 to Mac cable

BASIC control

One of the nicest features of the ADC-1 is that it is not devoted to any particular task; it is a general-purpose box, under the control of your computer. You determine what it monitors by connecting transducers, and how the retrieved information will be interpreted by writing a program—in our case in BASIC, though almost any other language would also serve the purpose. The controlling program can send commands out the communication port to the ADC-1, which acts on them. A response is sent by the ADC-1, received at the Macintosh modem port, and dealt with by the program, usually by either displaying it on the screen or entering it into a formula.

As you will see in the sample program we develop in this chapter, the portions of the program that actually communicate with the ADC-1 consist of a few short subroutines. The remainder of each program is devoted to displaying or manipulating the information provided by the ADC-1.

The price

I realize that "price" is not a technical term that is unfamiliar to you, but it is a pertinent factor in determining the feasibility of using your Macintosh to water your lawn or feed your cat. The ADC-1 sells for about $500. The transducers that connect to it range from a few dollars to a few hundred dollars each. This puts it out of the range of the average person who just wants to type in the BASIC program we'll discuss in a moment to see if it actually runs. However, if you have a need for a monitoring device, the ADC-1 is relatively low-priced for its usefulness.

If you would like to compare the features of the ADC-1 to those of other A/D converters, the list of manufacturers at the end of this chapter will give you a starting point (and, of course, you will want to read Chapter 18, which describes an A/D converter you can build). Most converters that can be connected to a computer will work with the Macintosh, and can be controlled by a simple BASIC program such as the following one, which plots the temperature.

A program to plot temperature

Both this program and the program in the next chapter were originally written in Microsoft BASIC 1.0 by Keith Ronnholm of Remote Measurement Systems and

Rob Spencer, a biochemist in Toronto. I converted them to BASIC 2.0 and added a simulation routine—a low-cost method used to see the display produced by the converter without having to actually buy or build one. This first program, which plots the temperature and displays it as shown in Figure 17-2, is capable of communicating with the ADC-1, but has an extra subroutine to fake a response if you don't have the control unit.

The temperature line is plotted as if there were a pen at the right edge of the plot area. Each new reading is plotted at that point, and then the plot area scrolls to the left. If the reading is above or below the specified plot-range, the plot area scrolls down or up to keep the points plotted within the area, and the maximum and minimum temperature labels are adjusted accordingly. The scroll command is new, so I'll discuss it in detail when we come to it. The complete program is listed in Figure 17-15, but as usual I'll take it a chunk at a time.

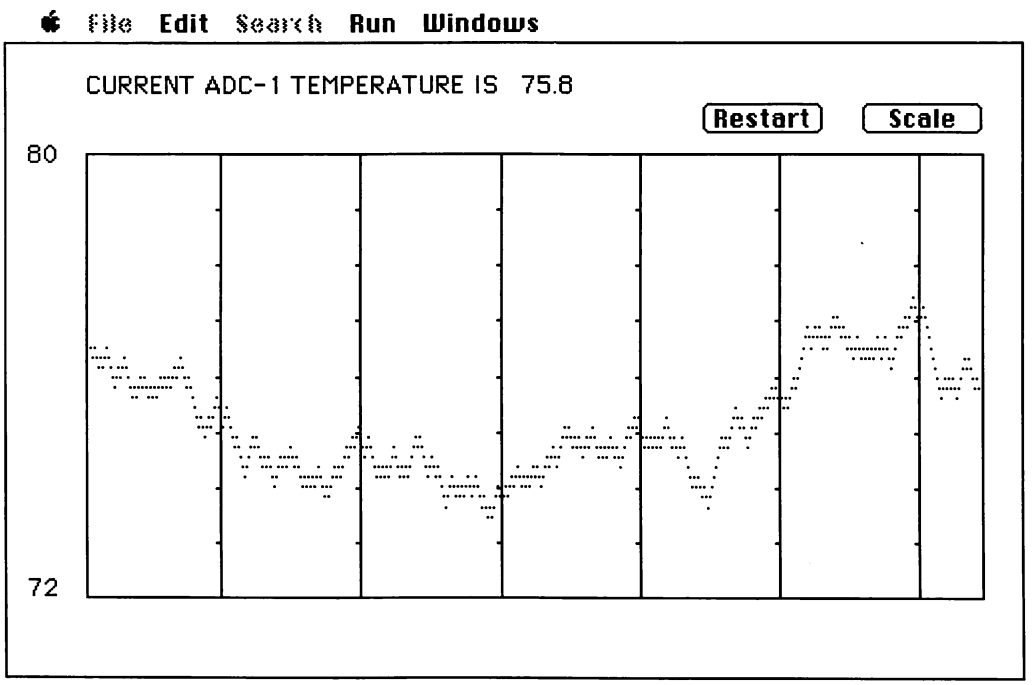


Figure 17-2. The temperature-plot display

```
'*
'* Initialize variables and open communication port.
'*
   CLEAR , 9000, 3000
InitializeVariables:
   false = 0 : true = NOT false
   simFlag = true
   timeStep = 15
   baudRate$ = "9600"                                    'for communication port
   maxTemp = 80 : minTemp = 72                           'max and min temperatures
   top = 55 : left = 40 : bottom = 280 : right = 490     'borders of plot area
   xValue = right - 1                                    'horizontal point to plot
   response = 2980                                       'initial number for simulation routine
   RANDOMIZE TIMER                                       'reseed random # generator
   chnl = 24                                             'ADC-1 internal temp sensor
   IF NOT simFlag THEN GOSUB OpenCom                     'open port if not simulating ADC-1
```

Figure 17-3. The *InitializeVariables* routine

Initializing the variables

The first section of this program, shown in Figure 17-3, initializes most of the variables used in the program and, if necessary, opens the communication port.

Notice the variable *simFlag*, near the top of this section. The true/false state of this variable determines whether the program opens the communication port and expects to get information from the ADC-1, or simulates this information with the *Siml* subroutine. The initial minimum and maximum temperatures and the dimensions of the plot area are also listed here, so someone running the program can edit a few items and totally change the appearance of the display. You will notice as we get into the program that everything displayed on the screen is placed relative to the plot area, so as the plot area is changed, so are the labels, buttons, and the plotted points.

Opening the communication port

If you are running this program with the ADC-1 connected, and therefore have *simFlag* set to *false*, the *OpenCom* subroutine in Figure 17-4 (on the following page) is used to open the communication port and flush the input buffer.

```
'*
'* Communication port is opened only if not simulating ADC-1.
'*
OpenCom:
   OPEN "com1:" + baudRate$ + ", n, 8, 2" AS #1
   garbage$ = INPUT$(LOC(1), 1)                                    'flush buffer
   RETURN
```

Figure 17-4. The *OpenCom* subroutine

The ADC-1 will communicate with the Macintosh at baud rates up to 19200, but each baud rate requires a different switch setting on the ADC-1.

Initializing the display

The next section, shown in Figure 17-5, opens a window and then draws the outline of the plot area and prints the labels around it.

The LINE statement at the bottom of this section creates the frame around the plot area by drawing a box from the top left corner to the bottom right corner, using

```
'*
'* Open window and create labels, buttons, and plot area.
'*
InitializeDisplay:
   WINDOW 1, , (1, 20) - (512, 342), 2

   '*
   '* Position buttons and labels relative to plot area.
   '*
   BUTTON 1, 1, "Restart", (right - 140, top - 25) - (right - 80, top - 10), 1
   BUTTON 2, 1, "Scale", (right - 60, top - 25) - (right, top - 10), 1
   MOVETO left, top - 30
   PRINT "CURRENT ADC-1 TEMPERATURE IS ";
   GOSUB PrintTemps
   LINE (left, top) - (right, bottom), 33, b                       'frame plot area
```

Figure 17-5. The *InitializeDisplay* routine

```
'*
'* Program returns here if Restart or Scale buttons clicked.
'*
StartNewPlot:
  degPerPixel = (maxTemp - minTemp) / (bottom - top)          'degrees per pixel
  LINE (left + 1, top + 1) - (right - 1, bottom - 1), 30, bf  'clear plot area
  mark = TIMER                                                'set start time
  OBSCURECURSOR                                               'hide cursor
```

Figure 17-6. The *StartNewPlot* section

the values of *top, left, bottom,* and *right* provided in the initialization section. Again, notice that the placement of everything else is relative to this plot area.

The three lines in the *StartNewPlot* section, shown in Figure 17-6, are really the tail end of the *InitializeDisplay* section; the extra label is added to provide a place to break into the initialization routine when plotting is restarted in the middle of a plot.

First, a new variable, *degPerPixel,* is defined to express the relationship between the temperature range and the height of the plot area. Then the LINE statement creates a filled box in white. Since the dimensions of the box are one pixel less than the plot area all the way around, this box effectively erases the plot area. The program places a time-mark on the plotted line every 15 seconds (or whatever other time period you specify), so setting *mark* equal to TIMER in the third line defines the starting time for the current plot. If you look through this section, you will find that the only thing you haven't previously bumped into is the OBSCURECURSOR ROM call in the last line. This call makes the cursor disappear until the mouse is moved. (Another ROM call that gets rid of the cursor is HIDECURSOR; the difference is that when you use HIDECURSOR, the cursor stays invisible until the SHOWCURSOR call is used.)

The next section, shown in Figure 17-7 on the following page, feeds the raw data returned by the ADC-1 (or the simulation subroutine) through a formula to convert to the units being plotted—in this case degrees Fahrenheit—and then checks the result against the range. If the point is outside the range, the plot area is scrolled up or down and the range adjusted. The point is then plotted and the plot area is scrolled to the left one pixel, in preparation for plotting the next point.

The formulas in this section are based on an ADC-1 channel with 12-bit resolution. The first line of this section jumps either to a subroutine that reads the internal

```
'*
'* Convert raw data returned to temperatures, and display.
'*
CrunchData:
  IF simFlag = true THEN GOSUB Sim1 ELSE GOSUB SndChnl
  degKelvin = response / 10                              'degrees Kelvin
  degFahr = (1.8 * (degKelvin - 273.16)) + 32            'degrees Fahrenheit
  MOVETO left + 210, top - 30
  PRINT USING "###.#"; degFahr;                          'print temp
  IF degFahr <= minTemp THEN dir = -1 : GOSUB ScrollVert 'adjust range
  IF degFahr >= maxTemp THEN dir = 1 : GOSUB ScrollVert  'adjust range
  yValue = bottom - (degFahr - minTemp) / degPerPixel    'compute Y-value
  PSET(xValue, yValue)                                   'plot it
  SCROLL (left + 1, top + 1) - (right - 1, bottom - 1), -1, 0
  IF TIMER = mark + timeStep THEN GOSUB TimeMark         'check time
  event = DIALOG(0)                                      'see if button has been clicked
  IF event <> 1 THEN GOTO CrunchData                     'continue plotting if no click
  butClick = DIALOG(1)
  IF butClick = 1 THEN GOTO StartNewPlot                 'restart
```

Figure 17-7. The *CrunchData* routine

temperature sensor in the ADC-1 and returns a value or to one that randomly generates a typical value. Either way, the value returned (the variable *response*) is an integer between 0 and 4095. We will look at the *SndChnl* and *Sim1* subroutines after we work our way through the rest of the program.

Let's assume, for the moment, that the temperature we want to plot is within the proper range, so the program is not diverted to the *ScrollVert* subroutine (I will come back to this subroutine after we plot a point and scroll to the left.) The statement that actually plots the point is:

```
PSET(xValue, yValue)                                     'plot it
```

which plots a point at the (x,y) coordinates represented by *xValue* and *yValue*. The full generic syntax for this statement is:

PSET [STEP] (*x,y*)[, *color*]

If the STEP option is used, the values of the x and y coordinates are considered to be relative to the current pen position; otherwise they are absolute values in the current window. If the *color* option is omitted, the point is plotted in black (color 33). This is the only difference between PSET and PRESET, which plots in white if the *color* option is not included.

In this program, the x coordinate is always one pixel to the left of the right edge of the plot area, and the y coordinate is determined by the formula:

```
yValue = bottom - (degFahr - minTemp) / degPerPixel                    'compute Y-value
```

After the point is plotted, the SCROLL statement is used to move the line to the left one pixel. The syntax for the SCROLL statement is:

SCROLL *rectangle, delta-x, delta-y*

The variable *rectangle* defines an area in the usual (x1,y1) - (x2,y2) format, and *delta-x* and *delta-y* are distances, in pixels, to scroll. Positive values of x and y scroll to the right and down; negative values scroll to the left and up.

I had a little trouble understanding the concept behind the SCROLL statement the first time I used it. For some reason I assumed that the entire rectangle would scroll *delta-x* or *delta-y* pixels. This is not the case. The *rectangle* variable defines the area on the screen affected by the scroll statement, but that area stays exactly where it is. Only the image within *rectangle* is moved. The portion of the image that moves out of *rectangle* disappears and the area of *rectangle* uncovered by moving the image is filled with the background pattern.

After the plot area is scrolled to the left, TIMER is checked to see if 15 seconds have elapsed since the last time the variable *mark* was set. If this is the case, then the *TimeMark* subroutine, shown in Figure 17-8 on the following page, is called to place a scale on the plot line. *TimeMark* resets the value of *mark* to the current time and then draws a vertical line across the plot line, with a tick mark at every degree step.

Let's go back to *CrunchData* and have a look at what happens if the monitored temperature moves outside the range defined by *maxTemp* and *minTemp*. After each reading is converted to degrees Fahrenheit, it is compared to the minimum and maximum temperature the plot area will display. If the temperature is less than or equal to

```
'*
'* Draw time-mark and scale line.
'*
TimeMark:
   mark = TIMER
   MOVETO xValue, top + 1 : LINETO xValue, bottom - 1
   degreeStep = (bottom - top) / (maxTemp - minTemp)
   FOR yMark = top TO bottom STEP degreeStep
      MOVETO xValue, yMark
      LINETO xValue - 2, yMark
   NEXT
   RETURN
```

Figure 17-8. The *TimeMark* subroutine

the minimum temperature, then the variable *dir* is set to −1 to indicate that we need to scroll up. If the temperature is greater than or equal to the maximum temperature, then *dir* is set to 1. Either way, we then call the *ScrollVert* subroutine, which is shown in Figure 17-9.

This routine computes how much the temperature exceeds the minimum or maximum, by subtracting *degFahr* from either *minTemp* or *maxTemp* depending on the value of *dir*. The difference is increased by one to make sure there is at least a little

```
'*
'* Move screen and adjust range if range exceeded.
'*
ScrollVert:
   IF dir = 1 THEN xTemp = maxTemp ELSE xTemp = minTemp
   excess = INT(ABS(xTemp - degFahr) + 1)
   maxTemp = maxTemp + excess * dir
   minTemp = minTemp + excess * dir
   pixToScrol = excess / degPerPixel
   SCROLL (left + 1, top + 1) - (right - 1, bottom - 1), 0, pixToScrol * dir
   GOSUB PrintTemps
   RETURN
```

Figure 17-9. The *ScrollVert* subroutine

scrolling on small over-ranges. The INT function is used in computing excess to avoid fractional values for *maxTemp* and *minTemp*.

The number of pixels to scroll is determined by dividing the number of degrees of excess temperature by the number of degrees represented by each pixel:

```
pixToScrol = excess / degPerPixel
```

The SCROLL statement then uses minus *pixToScrol* to move the plot area down, and positive *pixToScrol* to move it up.

Other than the subroutine that provides the raw data (*SndChnl* or *Sim1*) we have covered everything going on in the *CrunchData* section. You may have noticed that only one point is plotted during the pass through this section. This is obviously part of a loop that has to be repeated for each pixel plotted. The condition that returns the program to the top of the loop is tested for in the last four lines of *CrunchData:*

```
event = DIALOG(0)                          'see if button has been clicked
IF event <> 1 THEN GOTO CrunchData          'continue plotting if no click
butClick = DIALOG(1)
IF butClick = 1 THEN GOTO StartNewPlot                          'restart
```

Checking DIALOG(0) each time through the *CrunchData* loop is an alternative to formal dialog-event trapping with an ON DIALOG GOSUB... statement. If a button has been clicked since the last time DIALOG(0) was checked, the program falls through to the next section of code; otherwise it returns to the top of *CrunchData*.

You have been through the process of creating edit fields and retrieving the entries enough times that there should be no surprises in the next section, *CreateBox*, shown in Figure 17-10 on the following page. After retrieving the information in the edit fields, this section sends the program back to *StartNewPlot*, which is the re-entry point in the display initialization routine.

```
'*
'* Create dialog box with two edit fields and OK button.
'*
CreateBox:
  WINDOW 2, , (200, 180) - (475, 280), -4
  MOVETO 8,12 : PRINT "Edit minimum and maximum temperatures"
  MOVETO 30, 40 : PRINT "Maximum"
  MOVETO 30, 70 : PRINT "Minimum"
  BUTTON 3, 1, "OK", (220, 80) - (260, 95), 1
  EDIT FIELD 1, STR$(maxTemp), (100, 30) - (180, 45), 1, 2
  EDIT FIELD 2, STR$(minTemp), (100, 60) - (180, 75), 1, 2
  edFld = 2
  event = 0
  WHILE event <> 1 AND event <> 6                          'wait for new values
    event = DIALOG(0)
    IF event = 2 THEN edFld = DIALOG(2) : EDIT FIELD edFld
    IF event = 7 THEN edFld = (edFld MOD 2) + 1 : EDIT FIELD edFld
  WEND
  maxTemp = VAL(EDIT$(1))                                  'new maximum temperature
  minTemp = VAL(EDIT$(2))                                  'new minimum temperature
  WINDOW CLOSE 2                                           'close window after editing
  GOSUB PrintTemps
  GOTO StartNewPlot                                        'start monitoring again
```

Figure 17-10. The *CreateBox* routine

Communicating with the ADC-1

Most multiple-channel A/D converters, the ADC-1 included, do not monitor and convert all their channels all the time. They wait until a request for information about a specific channel is received and then they check that channel and get the retrieved information ready to send to the computer. Checking the channel and converting the measured analog information to digital information takes a little time (very little—usually measured in milli- or microseconds). When the converted data is ready, it is stored until the computer makes another request, and then it is sent to the computer. The specifics of this communication exchange vary from converter to converter; a typical exchange goes like this:

- Converter waits for request.
- Computer sends channel request.

- Converter responds with whatever garbage is in its output buffer.
- Computer reads its input buffer to get rid of garbage reply.
- Converter converts measured information to digital and waits for next request.
- Computer sends request for value.
- Converter responds with value.
- Computer retrieves value from its input buffer.

With slight variations, this general scheme is common to many converters. The only variation on the part of the ADC-1 is that, since it offers 12-bit resolution but passes information to the computer eight bits at a time, it requires two value requests to get the full value. The first request produces the high eight bits, and the second the low eight.

Figure 17-11 shows the routine used to send a channel number to the ADC-1, and the *Reply* routine that is used to get the response to the channel request and the high- and low-byte requests.

Communicating with the converter really doesn't differ from communicating with a modem. We do have to get a little tricky with the high and low bytes retrieved.

```
'*
'* Send channel request.
'*
SndChnl:
   PRINT #1, CHR$(chnl);                    'tell ADC-1 which channel to check
   GOSUB Reply                                          'garbage response

'*
'* Get reply.
'*
Reply:
   WHILE LOC(1) = 0 : WEND                              'wait for response
   returnedData = ASC(INPUT$(1, 1))                     'convert to ASCII
   RETURN
```

Figure 17-11. The *SndChnl* and *Reply* subroutines

```
'*
'* Request high byte.
'*
ReqHiByte:
   PRINT #1, CHR$(161);
   GOSUB Reply                                              'get it
   returnByte = returnedData
   IF (returnByte AND 128) <> 0 THEN ReqHiByte              'bit 7 has to be 0
   hiByte = returnByte AND 15                               'strip bits 4 through 7 (of 0-7)
```

Figure 17-12. The *ReqHiByte* routine

If you think back to the explanation of 8-, 12-, and 16-bit resolution in the last chapter, you will recall that 12-bit resolution does not use all eight bits of the high byte. Figure 17-12 shows how we get rid of the unwanted bits.

The number sent to the ADC-1 by the PRINT statement determines the kind of information the ADC-1 sends back. There is a total of 256 codes—the ASCII numbers from 0 through 255—and each one has its own meaning to the ADC-1. The code 161 retrieves the high byte without causing the converter to take any new action. There are other codes that would return the same information, but would simultaneously set or reset one of the digital channels.

We need only the lower four bits of the returned byte in our temperature calculation, but before stripping them out, the program checks bit 7 (the high bit—remember that they are numbered from 0 through 7). The ADC-1 sets bit 7 to 0 when it is returning a high byte. Confirming this bit confirms that the ADC-1 has had time to properly convert the channel we requested. The fact that bit 7 is a 0 is confirmed by ANDing the high byte with 128, which is 10000000 in binary notation. As you recall, these numbers are ANDed one bit at a time, so any number with a 0 as bit 7 will produce a 0 when ANDed with 128. The four bits we don't want—bits 4 through 7—are set to 0 by ANDing the high byte with 15, which is 00001111 in binary notation.

The low byte is requested in much the same manner—by sending code 145—and then the high and low bytes are combined, as shown in Figure 17-13.

The variable *response,* which we use in the *CrunchData* section to compute the temperature, is set equal to the low byte plus 256 times the high byte. Multiplying by 256 converts the high byte to the upper byte of a 2-byte number, which it was intended to be all along.

```
'*
'* Request low byte.
'*
ReqLoByte:
   PRINT #1, CHR$(145);
   GOSUB Reply
   loByte = returnedData
   response = loByte + 256 * hiByte                    'scale hiByte and add to low byte
   IF (returnByte AND 16) = 0 THEN response = -response          'checks bit 4 for sign
   RETURN
```

Figure 17-13. The *ReqLoByte* routine

All that remains is the simulation routine, which allows you to demonstrate this program without hooking your Macintosh to an analog-to-digital converter. As shown in Figure 17-14, this routine starts with the value of *response* (2980) specified in the *InitializeVariables* section. Each time through the simulation routine a value, randomly set to either +1 or −1, is added to *response,* so that the plotted point will fluctuate randomly, but gently.

Modifications to the program

You have to change only the formulas in the *CrunchData* section, and a few labels, to be able to use this program to plot something other than temperature. Remember that an A/D converter with 12-bit resolution will return a value between 0 and 4095, regardless of what it is monitoring. If the transducer is properly matched to

```
'*
'* Simulate response.
'*
Sim1:
   response = response + ((-1) ^ INT(2 * (RND(1) + 1)))
   RETURN
```

Figure 17-14. The *Sim1* subroutine

the condition being monitored and to the ADC-1 input, that range will represent the full range being monitored.

As an experiment, you might change the values of *top, left, bottom,* and *right,* to see the reason for making everything relative to these values. A few experiments requiring a little more effort are:

- Plotting a second line.
- Switching the point of the pen to the left edge of the plot area and scrolling right.
- Keeping track of the highest and lowest values plotted.
- Adding some bulletproofing to catch errors in the edit-field entries.
- Adding a delay so the program plots points at specific time intervals.

This is a fairly simple program, but it could easily be expanded to match your specific needs. I doubt that you will want to devote your Macintosh to the full-time monitoring of the temperature, but it could easily monitor your home's security while you are away, or your energy consumption. The complete program listing follows at the end of the chapter, in Figure 17-5.

Other A/D devices

As I said earlier, almost any device designed to plug into an RS-232 port can be connected via a modified cable to the Macintosh RS-422 port. The following analog-to-digital converters should work with the Macintosh, and I am sure their manufacturers would be happy to send additional information.

The Sensorbus

Transensory Devices, Inc.
44060 Old Warm Springs Blvd.
Fremont, CA 94538
(415) 490-3333

8232 Data Acquisition and Control System

Starbuck Data Company
P.O. Box 24
Newton Lower Falls, MA 02162
(617) 237-7695

PL-1000

Elexor Associates
P.O. Box 246
Morris Plains, NJ 07950
(201) 299-1615

WB-31 White Box

Omega Engineering
P.O. Box 4047
Stamford, CT 06907
(203) 359-7700

```
'* Temperature plotting progam, with simulation
'*

'*
'* Initialize variables and open communication port.
'*
   CLEAR , 9000, 3000
InitializeVariables:
   false = 0 : true = NOT false
   simFlag = true
   timeStep = 15
   baudRate$ = "9600"                              'for communication port
   maxTemp = 80 : minTemp = 72                     'max and min temperatures
   top = 55 : left = 40 : bottom = 280 : right = 490   'borders of plot area
   xValue = right - 1                              'horizontal point to plot
   response = 2980                                 'initial number for simulation routine
   RANDOMIZE TIMER                                 'reseed random # generator
   chnl = 24                                       'ADC-1 internal temp sensor
   IF NOT simFlag THEN GOSUB OpenCom               'open port if not simulating ADC-1
```

more...

Figure 17-15. The complete temperature-plotting program

```
'*
'* Open window and create labels, buttons, and plot area.
'*
InitializeDisplay:
   WINDOW 1, , (1, 20) - (512, 342), 2

   '*
   '* Position buttons and labels relative to plot area.
   '*
   BUTTON 1, 1, "Restart", (right - 140, top - 25) - (right - 80, top - 10), 1
   BUTTON 2, 1, "Scale", (right - 60, top - 25) - (right, top - 10), 1
   MOVETO left, top - 30
   PRINT "CURRENT ADC-1 TEMPERATURE IS ";
   GOSUB PrintTemps
   LINE (left, top) - (right, bottom), 33, b                    'frame plot area

'*
'* Program returns here if Restart or Scale buttons clicked.
'*
StartNewPlot:
   degPerPixel = (maxTemp - minTemp) / (bottom - top)           'degrees per pixel
   LINE (left + 1, top + 1) - (right - 1, bottom - 1), 30, bf   'clear plot area
   mark = TIMER                                                 'set start time
   OBSCURECURSOR                                                'hide cursor

'*
'* Convert raw data returned to temperatures, and display.
'*
CrunchData:
   IF simFlag = true THEN GOSUB Sim1 ELSE GOSUB SndChnl
   degKelvin = response / 10                                    'degrees Kelvin
   degFahr = (1.8 * (degKelvin - 273.16)) + 32                  'degrees Fahrenheit
   MOVETO left + 210, top - 30
   PRINT USING "###.#"; degFahr;                                'print temp
   IF degFahr <= minTemp THEN dir = -1 : GOSUB ScrollVert       'adjust range
   IF degFahr >= maxTemp THEN dir = 1 : GOSUB ScrollVert        'adjust range
   yValue = bottom - (degFahr - minTemp) / degPerPixel          'compute Y-value
   PSET(xValue, yValue)                                         'plot it
   SCROLL (left + 1, top + 1) - (right - 1, bottom - 1), -1, 0
   IF TIMER = mark + timeStep THEN GOSUB TimeMark               'check time
   event = DIALOG(0)                                  'see if button has been clicked
```

Figure 17-15. The complete temperature-plotting program *(continued)*

more...

```
   IF event <> 1 THEN GOTO CrunchData                    'continue plotting if no click
   butClick = DIALOG(1)
   IF butClick = 1 THEN GOTO StartNewPlot                                    'restart

'*
'* Create dialog box with two edit fields and OK button.
'*
CreateBox:
   WINDOW 2, , (200, 180) - (475, 280), -4
   MOVETO 8,12 : PRINT "Edit minimum and maximum temperatures"
   MOVETO 30, 40 : PRINT "Maximum"
   MOVETO 30, 70 : PRINT "Minimum"
   BUTTON 3, 1, "OK", (220, 80) - (260, 95), 1
   EDIT FIELD 1, STR$(maxTemp), (100, 30) - (180, 45), 1, 2
   EDIT FIELD 2, STR$(minTemp), (100, 60) - (180, 75), 1, 2
   edFld = 2
   event = 0
   WHILE event <> 1 AND event <> 6                              'wait for new values
      event = DIALOG(0)
      IF event = 2 THEN edFld = DIALOG(2) : EDIT FIELD edFld
      IF event = 7 THEN edFld = (edFld MOD 2) + 1 : EDIT FIELD edFld
   WEND
   maxTemp = VAL(EDIT$(1))                                  'new maximum temperature
   minTemp = VAL(EDIT$(2))                                  'new minimum temperature
   WINDOW CLOSE 2                                           'close window after editing
   GOSUB PrintTemps
   GOTO StartNewPlot                                            'start monitoring again

'*
'* Subroutines
'*
'*
'* Communication port is opened only if not simulating ADC-1.
'*
OpenCom:
   OPEN "com1:" + baudRate$ + ", n, 8, 2" AS #1
   garbage$ = INPUT$(LOC(1), 1)                                              'flush buffer
   RETURN
```

Figure 17-15. The complete temperature-plotting program *(continued)*

more...

```
'*
'* Send channel request.
'*
SndChnl:
   PRINT #1, CHR$(chnl);                          'tell ADC-1 which channel to check
   GOSUB Reply                                                    'garbage response

'*
'* Request high byte.
'*
ReqHiByte:
   PRINT #1, CHR$(161);
   GOSUB Reply                                                              'get it
   returnByte = returnedData
   IF (returnByte AND 128) <> 0 THEN ReqHiByte                    'bit 7 has to be 0
   hiByte = returnByte AND 15                          'strip bits 4 through 7 (of 0-7)

'*
'* Request low byte.
'*
ReqLoByte:
   PRINT #1, CHR$(145);
   GOSUB Reply
   loByte = returnedData
   response = loByte + 256 * hiByte                 'scale hiByte and add to low byte
   IF (returnByte AND 16) = 0 THEN response = -response          'checks bit 4 for sign
   RETURN

'*
'* Get reply.
'*
Reply:
   WHILE LOC(1) = 0 : WEND                                        'wait for response
   returnedData = ASC(INPUT$(1, 1))                                'convert to ASCII
   RETURN

'*
'* Simulate response.
'*
Sim1:
   response = response + ((-1) ^ INT(2 * (RND(1) + 1)))
   RETURN
```

Figure 17-15. The complete temperature-plotting program *(continued)*

more...

```
'*
'* Draw time-mark and scale line.
'*
TimeMark:
   mark = TIMER
   MOVETO xValue, top + 1 : LINETO xValue, bottom - 1
   degreeStep = (bottom - top) / (maxTemp - minTemp)
   FOR yMark = top TO bottom STEP degreeStep
      MOVETO xValue, yMark
      LINETO xValue - 2, yMark
   NEXT
   RETURN

'*
'* Display temperature range to left of scroll box.
'*
PrintTemps:
   TEXTMODE 1
   LINE (2, top - 5) - (left - 2, bottom), 30, bf
   MOVETO 2, top + 5 : PRINT USING "###"; maxTemp;
   MOVETO 2, bottom : PRINT USING "###"; minTemp;
   TEXTMODE 0
   RETURN

'*
'* Move screen and adjust range if range exceeded.
'*
ScrollVert:
   IF dir = 1 THEN xTemp = maxTemp ELSE xTemp = minTemp
   excess = INT(ABS(xTemp - degFahr) + 1)
   maxTemp = maxTemp + excess * dir
   minTemp = minTemp + excess * dir
   pixToScrol = excess / degPerPixel
   SCROLL (left + 1, top + 1) - (right - 1, bottom - 1), 0, pixToScrol * dir
   GOSUB PrintTemps
   RETURN
```

Figure 17-15. The complete temperature-plotting program *(continued)*

The HBC-1

Chapter 18

If you are willing to spend a little time tracking down components and hooking them together, you can build the HBC-1 (Home-Brew-Converter, version 1), a powerful analog-to-digital converter, for under $50.

The HBC-1 exists because I wanted a low-cost way for you to try the programs in this chapter. I would have settled for the crudest of instruments—anything that would hook the Macintosh to the outside world. But a reasonably thorough search produced nothing. And then I heard about a new A/D integrated circuit from Texas Instruments and sent a request for information. This resulted in a call from Gordon Mills, a linear field application engineer for Texas Instruments, who enthusiastically offered to design and help me build a converter using TI parts. Our original goal was to build the cheapest possible converter. I don't know if ours actually was the cheapest possible, but we did build an 11-channel 8-bit resolution converter for about $12 worth of parts. We then decided that for safety reasons we should isolate the power supply and the signal lines from the Macintosh. We added a plug-in power adaptor and a regulator circuit, rather than using power from the Macintosh, and optically isolated the input and output circuits, so that it would be nearly impossible for the converter to damage the Mac (or vice versa). One thing lead to another and we ended up with the HBC-1. Our total investment in parts is still under $50, and the machine is both safe and functional.

The HBC-1 is an inexpensive and flexible device that allows a computer to measure and control a variety of conditions. With the proper transducers it can monitor wind speed, temperature, light, weight, resistance, voltage, strain, moisture, pressure, joystick movement, power consumption, and practically any other measurable

analog phenomenon. It can also monitor digital levels and decode keys pressed on a keypad. Here are some significant features of this device:

- Eleven analog input channels, with a twelfth channel that provides a known output and can be used to test most of the rest of the circuitry.
- Resolution at 8 bits, with the possibility of adding a ninth bit by a process called dithering.
- Computer control of multiples of seven output channels (7, 14, 21, etc.) that can latch relays and other low-current devices to control electrical equipment.
- Powered by a plug-in AC power adaptor (which isolates the power supply from that of the computer) but draws very little current, so it could easily be converted to battery power.
- Input and output circuits that are optically isolated from the computer (under optimal conditions, up to 5000 volts peak), so there is little possibility of voltages monitored or generated by the HBC-1 damaging your computer (and vice versa).
- Operates at standard computer baud rates through 19.2K baud; with a different crystal, could operate at the highest BASIC-settable baud rate in the Macintosh: 57.6K baud.
- Input and output circuitry easily adapted to the protocols followed by the Macintosh, the IBM PC, the Commodore 64, and most other popular computers.

I realize that this is a book about programming, and that building hardware was probably the furthest thing from your mind when you picked the book up; but this project is not beyond the abilities of the average person, and there is no need for sophisticated test equipment to calibrate it. Besides, it can be a lot of fun and save you money to boot. Appendix C explains how to build the HBC-1 and how it works. Should you decide not to build the HBC-1, you can still enter and run this program, as it contains a simulation routine similar to the one in the program for the ADC-1.

A voltmeter program

The program presented in this chapter, which produces the display shown in Figure 18-1, was originally written to control the ADC-1. I modified it to control the HBC-1, but it could easily be switched back to the ADC-1 format by changing the subroutines that send the channel request to the converter and interpret the response.

The voltmeter program, listed in its entirety in Figure 18-15 at the end of the chapter, displays the value returned by one channel of a converter in three different formats: a digital readout, a dial, and a vertical bar chart. I have chosen to call this a voltmeter, but it could just as easily be measuring ohms, amps, temperature, pressure, strain, or just about anything else.

Like the previous program, this one has a simulation mode that allows it to run without being connected to a converter. Since most of the interesting parts of the program have to do with creating and modifying the display, which is done the same way

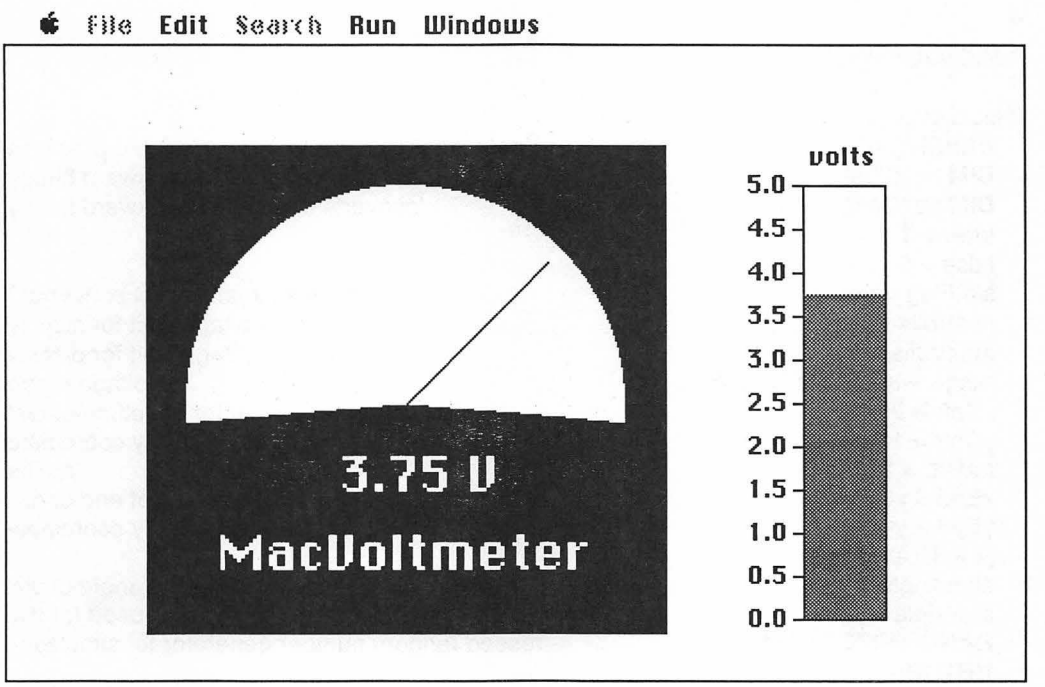


Figure 18-1. The voltmeter display

```
GOSUB InitializeVariables
GOSUB InitializeScreen
IF NOT simFlag THEN GOSUB InitializeCommunication
```

Figure 18-2. Routing the program with GOSUBs

regardless of the converter sending the information, let's first run through the program in the simulation mode, and then have a look at the changes required to run the program with the new machine.

The first three lines of the program, shown in Figure 18-2, route the program through the initialization routines. Of these three initialization routines, the first two are performed every time the program starts. The third routine is skipped if the program is in simulation mode, which is indicated when *simFlag* is set to *true.*

```
'*
'* Initialize variables.
'*
InitializeVariables:
   DEFSNG a - z                    'default for Decimal version is double precision
   DIM cnv(255)                    'converts response to backward binary
   DIM conChn(11)                  'converts channel to backward binary
   true = -1
   false = 0
   simFlag = true                  'are we simulating A/D converter?
   minVolts = 0                    'lower voltage limit for display
   maxVolts = 5                    'upper voltage limit for display
   range = maxVolts - minVolts     'voltage range
   xCntr = 200                     'x coordinate of center of voltmeter dial
   yCntr = 180                     'y coordinate
   radius = 100                    'radius
   xEnd = xCntr                    'x coordinate of end of dial
   yEnd = yCntr                    'y coordinate
   pi = 3.14159
   startAngle = -pi / 2            'used to compute angle of dial
   arcAngle = pi                   'used for dial
   RANDOMIZE TIMER                 'reseed random number generator for simulation
   RETURN
```

Figure 18-3. The *InitializeVariables* subroutine

The routine that initializes the variables is shown in Figure 18-3. The comments explain most of the new variables. You can see that the voltage range (*range*) is established here—from *minVolts* to *maxVolts*. This range is used in the formula that converts the raw data to volts and in the routine that creates the labels on the bar chart. The variables *pi, startAngle,* and *arcAngle* indicate we are going to get involved with trignometry—these are used to position the needle on the face of the dial.

The dial and bar chart are created by the *InitializeScreen* subroutine, shown in Figure 18-4. This subroutine opens a window and prints a little blurb, and then creates the dial, the bar chart, and the labels for the bar chart.

```
'*
'* Initialize  screen.
'*
InitializeScreen:
   WINDOW 1, , (1, 20) - (512, 342), 2
   TEXTFONT 0
   TEXTSIZE 12
   IF NOT simFlag THEN GOSUB GetInput

   '*
   '* Set up dial.
   '*
   side = radius * 1.3                                          'half width of rectangle
   LINE (xCntr - side, yCntr - side) - (xCntr + side, yCntr + .9 * side), , bf
   cntrRad = radius * 1.1                                       'radius of center circle
   rect%(0) = yCntr - cntrRad                                      'getting ready to ...
   rect%(1) = xCntr - cntrRad                                       'create the arc ...
   rect%(2) = yCntr + cntrRad                                          'used as a  ...
   rect%(3) = xCntr + cntrRad                                             'dial-face
   CALL ERASEARC(VARPTR(rect%(0)), 95, -190)

   '*
   '* Set up bar chart.
   '*
   gray%(0) = &HAA55
   gray%(1) = gray%(0)
   gray%(2) = gray%(0)
   gray%(3) = gray%(0)
```

Figure 18-4. The *InitializeScreen* subroutine

more...

```
span = 2 * cntrRad
top = yCntr - cntrRad
bottom = top + span
left = xCntr + radius + 100
right = left + 40
rect%(0) = top
rect%(1) = left
rect%(2) = bottom
rect%(3) = right

'*
'* Print numbers along left side.
'*
FOR increment = 0 TO 10
   value = bottom - span * increment / 10
   MOVETO left - 45, value + 4
   PRINT USING "###.#"; minVolts + (maxVolts - minVolts) * increment / 10;
   LINE(left - 2, value) - (left - 6, value)
   NEXT
   LINE(left - 1, top) - (right, bottom), , b
   MOVETO left + 2, top - 10
   PRINT "volts";
   oldBar = bottom
   TEXTSIZE 24
   TEXTMODE 2
   MOVETO xCntr - 95, yCntr + 85
   PRINT "MacVoltmeter"
   RETURN
```

Figure 18-4. The *InitializeScreen* subroutine *(continued)*

The body of the dial is created in two steps: first a black box is filled in with the LINE statement, and then an arc is erased within the box with the ERASEARC ROM call. The syntax of ERASEARC is similar to that of ROM calls you have already used to draw and erase rectangles and ovals:

CALL ERASEARC(VARPTR(*rectangle%*(0)),*startangle, arcangle*)

ERASEARC is, as usual, part of a family of ROM calls that also includes FRAMEARC, PAINTARC, INVERTARC, and FILLARC. Of the bunch, only FRAMEARC actually draws a curved line; each of the others creates a wedge.

The arc created by this call is actually a segment of the oval that would fit in the defined rectangle. If you imagine a compass card centered in the rectangle, with zero degrees at the top as shown in Figure 18-5, *startAngle* is the compass angle of one edge of the arc and *arcAngle* is the number of degrees in the arc. If *arcAngle* is positive, the arc extends in a clockwise direction from *startAngle;* if it is negative, it extends in a counterclockwise direction.

The first step in creating the bar chart is defining the pattern that will be used to fill it. I'm sure that you recognize this gray pattern—&HAA55—from the pattern-generation program in Section II. Once the pattern is set, the dimensions of the plot area are defined, based on the dimensions of the bar. Values for *top, left, bottom,* and *right* are assigned to *rect(0)* through *rect(3).* Those elements of this array that hold *top* and *bottom—rect(0)* and *rect(2)*—are redefined each time the height of the bar changes, and the array is used to fill and erase the pattern.

Next a FOR...NEXT loop determines where to print the bar-chart labels, and then computes and prints them. This is quite a bit of work for one little loop. As a final gesture it tacks on the tick-marks at each label position.

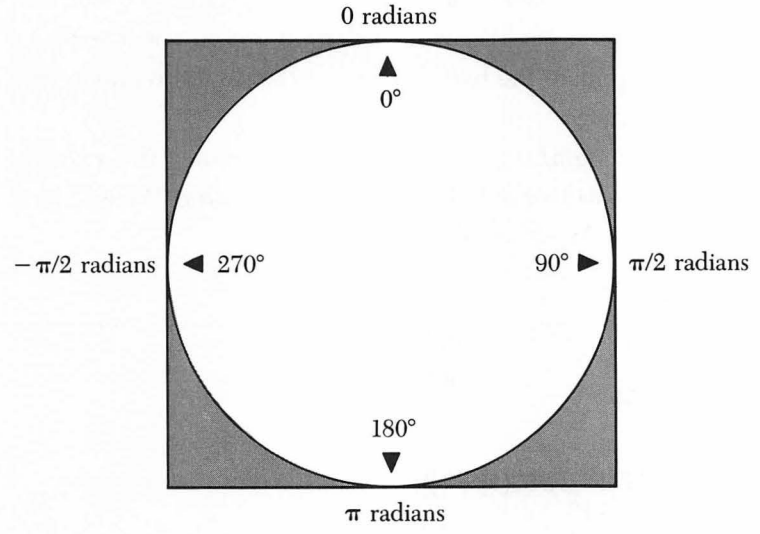


Figure 18-5. The degrees of an arc

```
'*
'* Compute voltage and update bar chart and dial.
'*
MainLoop:
   WHILE true
      IF simFlag THEN GOSUB sim2 ELSE GOSUB SndChnl
      volts = range * response / 256
      fraction = (volts - minVolts) / range
```

Figure 18-6. Computing the voltage

The only significant things left in this section are the LINE statement, which frames the plot area, and the implied LET statement that sets the variable *oldBar* equal to *bottom* (*oldBar=bottom* implies *LET oldbar=bottom*). Establishing the initial value of *oldBar* is important, as it determines the bottom of the bar; you will see in a moment just why this is significant.

After initializing the screen, the program flows into the section labeled *MainLoop* (remember, we have set *simFlag* to *true* for the time being). Although this is where the program will be most of the time, the preparation done in the other sections makes the work fairly light here. *MainLoop* is composed of four short sections that convert the reading to volts, update the bar chart, print the digital readout, and update the position of the dial pointer.

Figure 18-6 shows the first *MainLoop* section, which computes the voltage, based on the response from the converter or the simulation subroutine. We will look at the *SndChnl* subroutine later; *Sim2* is shown in Figure 18-7.

```
'*
'* No converter, so simulate response.
'*
sim2:
   response = response + ((-1) ^ INT(2 * (RND(1) + 1)))
   IF response < 5 THEN response = 128
   IF response > 250 THEN response = 128
   RETURN
```

Figure 18-7. The *Sim2* subroutine

The simulation routine is set up as though it were a converter with 8-bit resolution, meaning it would return a number between 0 and 255. Since we don't want to go off-scale, we start with a mid-range value for *response* and randomly add or subtract 1. If *response* gets below 5 or above 250, it is reset to 128.

Back in *MainLoop*, *response* is used to compute *volts*, the value of the voltage being measured. The formula that computes *volts* is the standard conversion formula explained in Chapter 16: the response times the total possible range of responses, divided by the possible number of steps from the converter (2 raised to the power of the number of bits of resolution). The variable *fraction* is then set equal to a formula that expresses the voltage as a fractional part of the entire range that can be plotted. This range is defined in the initialization section to be 5 volts. If the actual voltage measured were 2 volts then *fraction* would be ⅖, or two fifths of the total range. This fractional value is used in the next section of *MainLoop*, shown in Figure 18-8, to update the bar-chart display.

To create the bar chart, or modify an existing bar chart, the variable *newBar* is set equal to the y coordinate of the top of the new bar (*bottom* − *span* * *fraction*). The value of *newBar* is then compared to the value of *oldBar*, which is the top of the current bar. If the voltage hasn't changed since the last time it was measured, then *newBar* is equal to *oldBar* and the program skips to the end of *MainLoop*, bypassing the update sections entirely. Bypassing the updates if there is no change prevents the display from needlessly flickering as everything is erased and redrawn.

If *newBar* is less than *oldBar*, then the voltage must be going up (remember that *newBar* and *oldBar* are pixel locations, which decrease as they move toward the top of the screen, as opposed to the voltage plot values, which increase as they move up). If *newBar* is greater than *oldBar*, then the voltage must be going down.

```
'*
'* Place top of bar.
'*
newBar = bottom - span * fraction
IF newBar = oldBar THEN GOTO SkipLoop
IF newBar < oldBar THEN GOSUB MoreVolts
IF newBar > oldBar THEN GOSUB LessVolts
```

Figure 18-8. Updating the bar chart

Rather than erase and re-create the entire bar chart each time the voltage changes, which would take two ROM calls, the program only erases or creates the affected portion of the bar chart, which requires only one ROM call. If the voltage is going up, then a new piece is added to the top of the bar. If the voltage is going down, a chunk of the top is erased. The *MoreVolts* and *LessVolts* subroutines, shown in Figure 18-9, take care of these tasks.

Remembering that *rect(0)* is the top of a rectangle that is about to be filled or erased and that *rect(2)* is the bottom should make it pretty obvious what is going on here. *MoreVolts* defines a rectangle that extends from the new voltage down to the old, and then fills that rectangle with the gray pattern. (*rect(1)* and *rect(3)*, the two sides of the bar, never change.) *LessVolts* sets the top of the rectangle equal to the level of the old bar, and the bottom equal to the level of the new bar. It then erases the rectangle from the top of the old bar, leaving the new bar.

The third and fourth parts of *MainLoop* print the voltage just below the center of the dial and update the pointer position. These sections are shown in Figure 18-10.

There is nothing tricky about printing the digital readout, but drawing the line representing the dial pointer requires a little trigonometric magic. First a white line is

```
'*
'* Define rectangle to show increase in voltage on bar chart.
'*
MoreVolts:
   rect%(0) = newBar
   rect%(2) = oldBar
   FILLRECT VARPTR(rect%(0)), VARPTR(gray%(0))
   RETURN

'*
'* Erase some of bar chart, as voltage has gone down.
'*
LessVolts:
   rect%(0) = oldBar
   rect%(2) = newBar
   ERASERECT VARPTR(rect%(0))
   RETURN
```

Figure 18-9. The *MoreVolts* and *LessVolts* subroutines

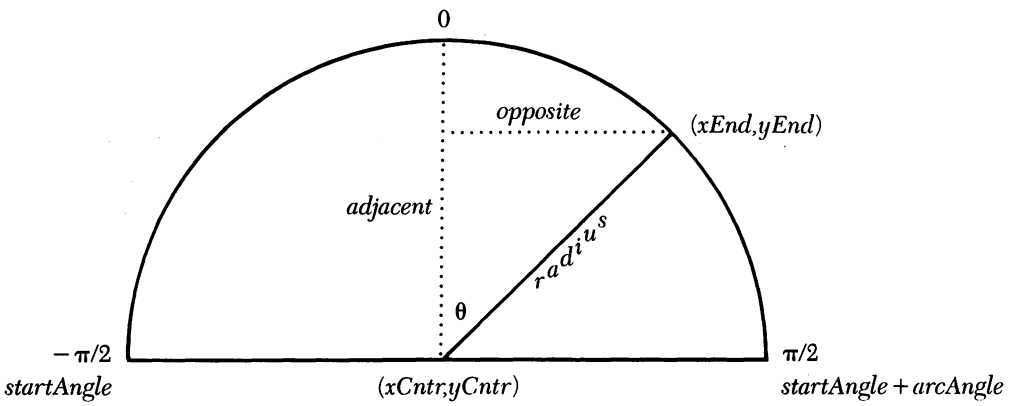
```
    '*
    '* Print digital display.
    '*
    LINE (xCntr - 50, yCntr +15) - (xCntr + 50, yCntr + 45), , bf
    MOVETO xCntr - 50, yCntr + 45
    PRINT USING "##.## V"; volts

    '*
    '* Place dial pointer.
    '*
    LINE(xCntr, yCntr) - (xEnd, yEnd), 30                    ' erase previous line
    theta = startAngle + arcAngle * fraction
    xEnd = xCntr + radius * SIN(theta)
    yEnd = yCntr - radius * COS(theta)
    LINE(xCntr, yCntr) - (xEnd, yEnd)
    oldBar = newBar
SkipLoop:
  WEND
```

Figure 18-10. Updating the dial

drawn using the current values of *xCntr* and *yCntr* and *xEnd* and *yEnd.* This puts the white line on top of the black dial-pointer line, erasing it. Next we want to draw a new line from the center of the dial to a point on the perimeter of an imaginary semicircle just smaller than the face of the dial. Use the SIN and COS functions to determine this point. Our imaginary semicircle's radius is defined by the variable *radius* (remember that the arc we erased to form the dial face had a radius of *radius* * *1.1*) and extends from $-pi/2$ radians to $pi/2$ radians, as shown in Figure 18-11 on the following page.

The sine of theta is equal to a ratio of the opposite side to the hypotenuse (the radius in this case), and the cosine is equal to the ratio of the side adjacent to the hypotenuse. If you enter this information into the formulas for *xEnd* and *yEnd,* and cancel out the common variable *radius,* you will find that *xEnd* is equal to *xCntr* plus the length of the opposite side, and *yEnd* is equal to *yCntr* minus the length of the adjacent side. The formula, of course, does the calculation for you whether you understand it or not, but sometimes it is nice to look at the logic behind the magic.

The trig functions provide the relative distances from the center, in pixels, of the x and y components of the point at the end of the dial pointer. Since the dial's center is



$\theta = startAngle + arcAngle * fraction$

if $fraction = 3/4$, then $\theta = \pi/4$

$$\begin{aligned} xEnd &= xCntr + radius * \text{SIN}(\theta) \\ &= xCntr + radius * opposite / radius \\ &= xCntr + opposite \end{aligned}$$

$$\begin{aligned} yEnd &= yCntr - radius * \text{COS}(\theta) \\ &= yCntr - radius * adjacent / radius \\ &= yCntr - adjacent \end{aligned}$$

Figure 18-11. Computing the dial-pointer position

not at point (0,0), the x distance is added to *xCntr* and the y distance is subtracted from *yCntr* to get the absolute coordinates of the point.

After the line representing the dial pointer is drawn, *oldBar* is set equal to *newBar* in preparation for the next pass through the loop.

The real thing

If you decide to get serious about analog-to-digital conversion, and actually build the HBC-1 described in Appendix C, you will have to add a few more sections to this program. In order to keep the hardware as simple as possible, we have placed a little extra burden on the software.

Just as for the ADC-1, you will have to open the communication port to send requests and receive responses. Unlike the ADC-1, this converter has 8-bit resolution, so

the response comes back as one byte. Another difference is that each channel request triggers the response to the previous channel request. So rather than sending a request and asking for a response, the program sends a request and immediately reads the response to the previous request. If you want to continuously interrogate the same channel, as this program and the one in the previous chapter do, then this is no problem: The first response is garbage and the last request is never answered, but everything in between is fine. If you are monitoring multiple channels or monitoring one channel intermittently, then the program will have to keep track of which response goes with which request.

One more difference—and this one is more significant—and then we will get on with the program: We have looked, in several of our programs, at the way bytes of information are stored and communicated as 8-bit ASCII characters. Serial communication, which is what we are doing when we exchange data through the Macintosh communication port, sends these bytes out in a stream, one bit after another. In our previous discussions of bits and bytes, we have always pictured a byte as eight bits, numbered 0 through 7, with the least significant bit (LSB, which is bit 0) on the right and the most significant bit (MSB, which is bit 7) on the left. That is the standard method of referring to bits in a computer. When a byte is sent out the communication port, however, the computer normally reverses it, with the MSB going out first and the LSB last. This is done to accommodate the integrated circuit, called a UART, that is normally used to receive serial communication in modems, printers, and other devices. The HBC-1 doesn't use a UART, so if we simply instructed the computer to send the byte that we wanted the HBC-1 to receive, it would reverse the byte and the HBC-1 would become confused. We can solve this problem by reversing the byte ourselves before telling the computer to send it, so when it is again reversed by the computer, it will be back to normal.

Rather than converting each number on the fly, I stored the converted values for the 12 channel numbers and the 256 possible responses in DATA statements, and as part of the communication initialization routine I load them into two arrays: one for the channel numbers, and the other for the responses. This technique of using a look-up table is usually faster than repeatedly performing the same math for each conversion. Doing the conversion this way involves a little extra work, but it shaved about 10 percent off the cost of the converter, since we avoided the need for an expensive chip that would have resulted in the same conversion.

```
'*
'* Initialize communication if simulation not being used.
'*
InitializeCommunication:

   '*
   '* Convert channel numbers 0 - 11 to backward binary.
   '*
   FOR channel = 0 TO 11
      READ conChn(channel)
   NEXT

   '*
   '* Convert 256 possible values of returned data to backward binary.
   '*
   FOR returnedData = 0 TO 255
      READ cnv(returnedData)
   NEXT

   '*
   '* Open communication port.
   '*
   OPEN "COM1:9600, n, 8, 2" AS #1

   '*
   '* Flush the input bufer.
   '*
   garbage$ = INPUT$(LOC(1), 1)
   RETURN

   '*
   '* Here are converted values for 12 channels and 256 responses.
   '*
   DATA 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13
   DATA 0, 128, 64, 192, 32, 160, 96, 224, 16
   DATA 144, 80, 208, 48, 176, 112, 240, 8
   DATA 136, 72, 200, 40, 168, 104, 232, 24
   DATA 152, 88, 216, 56, 184, 120, 248, 4
   DATA 132, 68, 196, 36, 164, 100, 228, 20
   DATA 148, 84, 212, 52, 180, 116, 244, 12
   DATA 140, 76, 204, 44, 172, 108, 236, 28
   DATA 156, 92, 220, 60, 188, 124, 252, 2
```

Figure 18-12. The *InitializeCommunication* subprogram

more...

```
DATA 130, 66, 194, 34, 162, 98, 226, 18
DATA 146, 82, 210, 50, 178, 114, 242, 10
DATA 138, 74, 202, 42, 170, 106, 234, 26
DATA 154, 90, 218, 58, 186, 122, 250, 6
DATA 134, 70, 198, 38, 166, 102, 230, 22
DATA 150, 86, 214, 54, 182, 118, 246, 14
DATA 142, 78, 206, 46, 174, 110, 238, 30
DATA 158, 94, 222, 62, 190, 126, 254, 1
DATA 129, 65, 193, 33, 161, 97, 225, 17
DATA 145, 81, 209, 49, 177, 113, 241, 9
DATA 137, 73, 201, 41, 169, 105, 233, 25
DATA 153, 89, 217, 57, 185, 121, 249, 5
DATA 133, 69, 197, 37, 165, 101, 229, 21
DATA 149, 85, 213, 53, 181, 117, 245, 13
DATA 141, 77, 205, 45, 173, 109, 237, 29
DATA 157, 93, 221, 61, 189, 125, 253, 3
DATA 131, 67, 195, 35, 163, 99, 227, 19
DATA 147, 83, 211, 51, 179, 115, 243, 11
DATA 139, 75, 203, 43, 171, 107, 235, 27
DATA 155, 91, 219, 59, 187, 123, 251, 7
DATA 135, 71, 199, 39, 167, 103, 231, 23
DATA 151, 87, 215, 55, 183, 119, 247, 15
DATA 143, 79, 207, 47, 175, 111, 239, 31
DATA 159, 95, 223, 63, 191, 127, 255
```

Figure 18-12. The *InitializeCommunication* subprogram *(continued)*

Initializing communication

The initialization routine and the DATA statements it reads are shown in Figure 18-12. There is nothing new or unusual here. The arrays *conChn* and *cnv* are dimensioned, the two FOR...NEXT loops read the DATA statements and assign the values to the arrays, the communication port is opened, and the buffer flushed. Other than the arrays, this is just like any other communication session.

If the program is started with *simFlag* set to *false,* it uses the *GetInput* subroutine, shown in Figure 18-13 on the following page, to find out which channel the user wants to monitor. This routine requests a channel number, verifies that it is in the proper range, and then converts it to backward binary by setting it equal to its counterpart in the *conChn* array.

```
'*
'* Ask user which channel to monitor.
'*
GetInput:
   MOVETO 30, 30
   INPUT "Display which channel (0 through 11) "; chanNum
   chanNum = conChn(chanNum)                          'channels numbered 0-11
   RETURN
```

Figure 18-13. The *GetInput* subroutine

Once the program knows which channel to monitor, it uses the *SndChnl* and *Reply* routines, shown in Figure 18-14, to interrogate the converter and retrieve the responses. *SndChnl* sends out the channel number that was just converted in *GetInput*, and *Reply* gets the response (actually, the response to the previous request). As soon as the response is retrieved, it is converted to normal binary.

That's about all there is to this program. The two programs we have looked at in this section of the book are rather simple demo programs: Neither makes full use of the capabilities of the ADC-1 or the HBC-1. I have written one rather long program

```
'*
'* Tell converter which channel to monitor.
'*
SndChnl:
   PRINT #1, CHR$(chanNum);
   GOSUB Reply
   RETURN

'*
'* Get response.
'*
Reply:
   WHILE LOC(1) = 0 : WEND
   returnedData = ASC(INPUT$(1, 1))
   response = cnv(returnedData)
   RETURN
```

Figure 18-14. The *SndChnl* and *Reply* subroutines

that does allow you to use all the features of the HBC-1, but since it introduces few new BASIC concepts, it would not be very enlightening to drag you through an explanation of the whole thing—especially since it is too long to type in just for fun. I have included the full listing of this program on the companion disk, for those of you who would like to use it, either as is or as the basis for ideas.

```
'* Voltmeter program with simulation
'*

   GOSUB InitializeVariables
   IF NOT simFlag THEN GOSUB InitializeCommunication
   GOSUB InitializeScreen

'*
'* Compute voltage and update bar chart and dial.
'*
MainLoop:
   WHILE true
      IF simFlag THEN GOSUB sim2 ELSE GOSUB SndChnl
      volts = range * response / 256
      fraction = (volts - minVolts) / range

      '*
      '* Place top of bar.
      '*
      newBar = bottom - span * fraction
      IF newBar = oldBar THEN GOTO SkipLoop
      IF newBar < oldBar THEN GOSUB MoreVolts
      IF newBar > oldBar THEN GOSUB LessVolts

      '*
      '* Print digital display.
      '*
      LINE (xCntr - 50, yCntr +15) - (xCntr + 50, yCntr + 45), , bf
      MOVETO xCntr - 50, yCntr + 45
      PRINT USING "##.## V"; volts
```

more...

Figure 18-15. The complete voltmeter program

```
      '*
      '* Place dial pointer.
      '*
      LINE(xCntr, yCntr) - (xEnd, yEnd), 30                    ' erase previous line
      theta = startAngle + arcAngle * fraction
      xEnd = xCntr + radius * SIN(theta)
      yEnd = yCntr - radius * COS(theta)
      LINE(xCntr, yCntr) - (xEnd, yEnd)
      oldBar = newBar
SkipLoop:
   WEND

'*
'* Some subroutines
'*
'*
'* Ask user which channel to monitor.
'*
GetInput:
   MOVETO 30, 30
   INPUT "Display which channel (0 through 11) "; chanNum
   chanNum = conChn(chanNum)                                'channels numbered 0-11
   RETURN

'*
'* Tell converter which channel to monitor.
'*
SndChnl:
   PRINT #1, CHR$(chanNum);
   GOSUB Reply
   RETURN

'*
'* Get response.
'*
Reply:
   WHILE LOC(1) = 0 : WEND
   returnedData = ASC(INPUT$(1, 1))
   response = cnv(returnedData)
   RETURN
```

Figure 18-15. The complete voltmeter program *(continued)*

more...

```
'*
'* No converter, so simulate response.
'*
sim2:
   response = response + ((-1) ^ INT(2 * (RND(1) + 1)))
   IF response < 5 THEN response = 128
   IF response > 250 THEN response = 128
   RETURN

'*
'* Define rectangle to show increase in voltage on bar chart.
'*
MoreVolts:
   rect%(0) = newBar
   rect%(2) = oldBar
   FILLRECT VARPTR(rect%(0)), VARPTR(gray%(0))
   RETURN

'*
'* Erase some of bar chart, as voltage has gone down.
'*
LessVolts:
   rect%(0) = oldBar
   rect%(2) = newBar
   ERASERECT VARPTR(rect%(0))
   RETURN

'*
'* Initialize variables.
'*
InitializeVariables:
   DEFSNG a - z                    'default for Decimal version is double precision
   DIM cnv(255)                    'converts response to backward binary
   DIM conChn(11)                  'converts channel to backward binary
   true = -1
   false = 0
   simFlag = true                  'are we simulating A/D converter?
   minVolts = 0                    'lower voltage limit for display
   maxVolts = 5                    'upper voltage limit for display
   range = maxVolts - minVolts     'voltage range
   xCntr = 200                     'x coordinate of center of voltmeter dial
   yCntr = 180                     'y coordinate
```

Figure 18-15. The complete voltmeter program *(continued)*

more...

```
    radius = 100                                                    'radius
    xEnd = xCntr                                                    'x coordinate of end of dial
    yEnd = yCntr                                                    'y coordinate
    pi = 3.14159
    startAngle = -pi / 2                                            'used to compute angle of dial
    arcAngle = pi                                                   'used for dial
    RANDOMIZE TIMER                                 'reseed random number generator for simulation
    RETURN

'*
'* Initialize  screen.
'*
InitializeScreen:
    WINDOW 1, , (1, 20) - (512, 342), 2
    TEXTFONT 0
    TEXTSIZE 12
    IF NOT simFlag THEN GOSUB GetInput

    '*
    '* Set up dial.
    '*
    side = radius * 1.3                                             'half width of rectangle
    LINE (xCntr - side, yCntr - side) - (xCntr + side, yCntr + .9 * side), , bf
    cntrRad = radius * 1.1                                          'radius of center circle
    rect%(0) = yCntr - cntrRad                                        'getting ready to ...
    rect%(1) = xCntr - cntrRad                                        'create the arc ...
    rect%(2) = yCntr + cntrRad                                          'used as a  ...
    rect%(3) = xCntr + cntrRad                                          'dial-face
    CALL ERASEARC(VARPTR(rect%(0)), 95, -190)

    '*
    '* Set up bar chart.
    '*
    gray%(0) = &HAA55
    gray%(1) = gray%(0)
    gray%(2) = gray%(0)
    gray%(3) = gray%(0)
    span = 2 * cntrRad
    top = yCntr - cntrRad
    bottom = top + span
    left = xCntr + radius + 100
    right = left + 40
```

Figure 18-15. The complete voltmeter program *(continued)*

more...

```
    rect%(0) = top
    rect%(1) = left
    rect%(2) = bottom
    rect%(3) = right

    '*
    '* Print numbers along left side.
    '*
    FOR increment = 0 TO 10
       value = bottom - span * increment / 10
       MOVETO left - 45, value + 4
       PRINT USING "###.#"; minVolts + (maxVolts - minVolts) * increment / 10;
       LINE(left - 2, value) - (left - 6, value)
    NEXT
    LINE(left - 1, top) - (right, bottom), , b
    MOVETO left + 2, top - 10
    PRINT "volts";
    oldBar = bottom
    TEXTSIZE 24
    TEXTMODE 2
    MOVETO xCntr - 95, yCntr + 85
    PRINT "MacVoltmeter"
    RETURN

'*
'* Initialize communication if simulation not being used.
'*
InitializeCommunication:

    '*
    '* Convert channel numbers 0 - 11 to backward binary.
    '*
    FOR channel = 0 TO 11
       READ conChn(channel)
    NEXT

    '*
    '* Convert 256 possible values of returned data to backward binary.
    '*
    FOR returnedData = 0 TO 255
       READ cnv(returnedData)
    NEXT
```

Figure 18-15. The complete voltmeter program *(continued)*

more...

```
'*
'* Open communication port.
'*
OPEN "COM1:9600, n, 8, 2" AS #1

'*
'* Flush the input bufer.
'*
garbage$ = INPUT$(LOC(1), 1)
RETURN

'*
'* Here are converted values for 12 channels and 256 responses.
'*
DATA 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13
DATA 0, 128, 64, 192, 32, 160, 96, 224, 16
DATA 144, 80, 208, 48, 176, 112, 240, 8
DATA 136, 72, 200, 40, 168, 104, 232, 24
DATA 152, 88, 216, 56, 184, 120, 248, 4
DATA 132, 68, 196, 36, 164, 100, 228, 20
DATA 148, 84, 212, 52, 180, 116, 244, 12
DATA 140, 76, 204, 44, 172, 108, 236, 28
DATA 156, 92, 220, 60, 188, 124, 252, 2
DATA 130, 66, 194, 34, 162, 98, 226, 18
DATA 146, 82, 210, 50, 178, 114, 242, 10
DATA 138, 74, 202, 42, 170, 106, 234, 26
DATA 154, 90, 218, 58, 186, 122, 250, 6
DATA 134, 70, 198, 38, 166, 102, 230, 22
DATA 150, 86, 214, 54, 182, 118, 246, 14
DATA 142, 78, 206, 46, 174, 110, 238, 30
DATA 158, 94, 222, 62, 190, 126, 254, 1
DATA 129, 65, 193, 33, 161, 97, 225, 17
DATA 145, 81, 209, 49, 177, 113, 241, 9
DATA 137, 73, 201, 41, 169, 105, 233, 25
DATA 153, 89, 217, 57, 185, 121, 249, 5
DATA 133, 69, 197, 37, 165, 101, 229, 21
DATA 149, 85, 213, 53, 181, 117, 245, 13
DATA 141, 77, 205, 45, 173, 109, 237, 29
DATA 157, 93, 221, 61, 189, 125, 253, 3
DATA 131, 67, 195, 35, 163, 99, 227, 19
```

Figure 18-15. The complete voltmeter program *(continued)*

more...

```
DATA 147, 83, 211, 51, 179, 115, 243, 11
DATA 139, 75, 203, 43, 171, 107, 235, 27
DATA 155, 91, 219, 59, 187, 123, 251, 7
DATA 135, 71, 199, 39, 167, 103, 231, 23
DATA 151, 87, 215, 55, 183, 119, 247, 15
DATA 143, 79, 207, 47, 175, 111, 239, 31
DATA 159, 95, 223, 63, 191, 127, 255
```

Figure 18-15. The complete voltmeter program *(continued)*

SECTION VI

Appendices

VI

Alphabetical List of Commands

Appendix A

This list of the statements, functions, and ROM calls available in BASIC is primarily meant to serve as a reminder of their purpose. Page numbers, where given, indicate the first use in this book of the command. For a more detailed explanation and additional examples of each, consult the manual for Microsoft BASIC for the Apple Macintosh, provided with your copy of BASIC.

ABS function, 269

ABS(*X*)

Returns the absolute value of the number *X*. The absolute value is the value of the number regardless of the positive or negative sign that may precede it.

ASC function, 449

ASC(*X$*)

Returns the ASCII code for the first character of the string *X$*. ASCII codes, which are listed in the appendix of the BASIC manual, range from 0 through 255.

ATN function

ATN(*X*)

Returns the arctangent of *X*, where *X* is a ratio of the side opposite an obtuse angle to the side adjacent in a right triangle. The result is in radians, in the range of $-\pi/2$ through $\pi/2$. The arctangent is an inverse trigonometric function; that is, arctangent *X* is an angle whose tangent is *X*.

BEEP statement, 193

BEEP

Causes a short beep from the Macintosh speaker. Used to draw attention or alert the user to a problem. Same as the statement PRINT CHR$(7).

BREAK ON statement

BREAK ON

Enables trapping of an attempt to stop the program by pressing Command-period, or by selecting Stop from the Run menu. Normally used in conjunction with the ON BREAK statement.

BREAK OFF statement

BREAK OFF

Disables the event trapping that was enabled with the BREAK ON statement.

BREAK STOP statement

BREAK STOP

Suspends trapping of Command-period and Stop events, but keeps track of them and takes appropriate action when BREAK ON is executed.

BUTTON statement, 59

BUTTON *ID, state*[, *title, rectangle*[, *type*]]

Creates a button in the current output window. The *ID* is an integer greater than 0 used to refer to the button in subsequent commands. A *state* of 0, 1, or 2 makes the button inactive, active, or active and currently selected. The *title* is the text that will appear in or adjacent to the button. The coordinates given in *rectangle* specify where the button will appear relative to the upper left corner of the window. A *type* of 1, 2, or 3 determines the shape of the button. *Title, rectangle,* and *type* are optional if the button has previously been defined with them, in which case the statement can be issued

with just the *ID* and the *state,* in order to change the *state.* Remember that text can be replaced by string variables, and numbers can be replaced by expressions that evaluate to a valid number.

BUTTON function

BUTTON(*ID*)

Returns the *state,* 0, 1, or 2, of the named button.

BUTTON CLOSE statement

BUTTON CLOSE *ID*

Removes button *ID* from the current output window. In order to redisplay the button, the full BUTTON statement, including *rectangle,* must be given. To temporarily prevent the user from selecting a button, there is no need to close it; simply change its *state* to 0 with BUTTON *ID, 0.*

CALL statement, 35

CALL *name*[(*argument-list*)]

name [*argument-list*]

Transfers program flow either to a machine-language subroutine or to a BASIC subprogram. The most common machine-language subprograms are the Macintosh toolbox calls, usually referred to as ROM calls, listed separately at the end of this appendix. Other than in certain cases, as from within an IF...THEN statement, the word *CALL* and the parentheses around the argument list are optional—but not individually: If you include CALL you must include the parentheses.

CDBL function

CDBL(*X*)

Converts *X* to a double-precision number. Primarily used to express the result of single-precision or integer division with more accuracy.

CHAIN statement

CHAIN [MERGE] *filespec*[, [*expression*][, [ALL][, DELETE *range*]]]

Used to run a separate program or subprogram from within the currently running program. Typically used to partition large programs into short segments that are stored separately on disk and run as needed. The MERGE option appends the new program (which must be stored as an ASCII file) to the program currently in memory. The *filespec* is the name of the file to chain to, and can include a volume name. The *expression* is a line number (not an alphanumeric label) that specifies where to start execution of the chained program, if other than at the first line. The ALL option passes all variables in the current program to the chained program. The DELETE option does the opposite of MERGE, removing the specified range of program lines. The range to be deleted can be given in either line numbers or labels.

CHR$ function, 459

CHR$(*I*)

Returns a string consisting of the character that has the ASCII value *I*. *I* must be in the range 0 through 255. This function is often used to send control codes to the printer or screen.

CINT function

CINT(*X*)

Converts *X*, which must be in the range from −32768 to 32767, to an integer by rounding any fractional portion.

CIRCLE statement, 283

CIRCLE [STEP] (*x,y*), *radius*[, *color*[, *start, end*[, *aspect*]]]

Draws a circle or ellipse with the specified (*x,y*) center and *radius*. The coordinates of the center can be given as absolute *(x,y)* values, or, if the STEP option is used, as the relative distance from the current position of the pen. The *radius* is given in pixels. The *color* can be black (33) or white (30), with black as the default. The *start*

and *end* parameters indicate the beginning and ending angle in radians (ranging from -2π through 2π), allowing you to draw an arc that is a portion of the circle or ellipse defined by the statement. A negative *start* or *end* angle causes the ends of the arc to be connected to the center (as with Pacman). The *aspect*, which defaults to 1, is the ratio of the *y* to the *x* radius. If the aspect ratio is not 1, then the longer radius is used for the *radius* parameter.

CLEAR statement, 165

CLEAR[, [*data-segment-size*][, *stack-size*]]

Used to manage the distribution of the Mac's available random-access memory between three areas: the data-segment storage area, the stack storage area, and the heap storage area. Memory management on a Fat Mac is rarely a significant issue, but on a Slim Mac you may have to reallocate sections of memory to allow a program to run, or to make it run faster. The default allocations are approximately:

Area	Slim Mac	Fat Mac
Data-segment	21K	¾ of available memory
Stack	6K	15K
Heap	remainder	remainder

In addition to allocating memory, CLEAR closes all files, resets all variables, resets all DEF FN and DEFSNG/DBL/STR statements, and releases all disk buffers.

CLOSE statement, 63

CLOSE [[#]*filenumber*[, [#]*filenumber*...]]

Closes the specified file or files. If no *filenumber* is specified, closes all open files.

CLS statement, 59

CLS

Clears graphics and text (not buttons or edit fields) from the current output window and moves the pen to the upper left corner (0,0).

COMMON statement

COMMON *variable-list*

Used with the CHAIN statement when the ALL option is not used to pass variables. The *variable-list* in the COMMON statement is a list of the variables that are common to the current program and to the chained program. You can have more than one COMMON statement in a program, but a variable can appear in only one.

CONT statement

CONT

Continues the execution of a program that was stopped by pressing Command-period, or choosing Stop or Step from the Run menu. This is handy for debugging, as variable values can be examined and changed while the program is stopped, and then the program can be continued.

COS function, 381

COS(*X*)

Returns the cosine of *X*, where *X* is in radians. The cosine is the ratio between the side adjacent to a given acute angle in a right triangle and the hypotenuse. As the angle increases from 0 through 360 degrees, *X* will increase from 0 radians to 2π radians. COS(0) is 1, COS(π / 2) is 0, COS(π) is -1, COS(3 * π / 2) is 0, and COS(2 * π) is 1.

CSNG function

CSNG(*X*)

Converts *X* to a single-precision number.

CSRLIN function

CSRLIN

Returns the approximate line number (using the size of the character *O* in the current font as a measure) of the pen in the current output window. This statement is included to be compatible with versions of BASIC for other computers, but is not the

most effective way to get this information on the Macintosh, which displays a variety of fonts and sizes on the screen and therefore does not have a set number of lines. See the WINDOW(4) and WINDOW(5) functions and the GETPEN ROM call for information about better methods.

CVI function, 477

CVI(*2-byte string*)

CVS function, 477

CVS(*4-byte string*)

CVD function, 477

CVD(*8-byte string*)

Convert random-file numeric strings to numeric variables. CVI converts a 2-byte string to an integer, CVS converts a 4-byte string to a single-precision number, and CVD converts an 8-byte string to a double-precision number.

CVDBCD function

CVDBCD(*X$*)

CVSBCD function

CVSBCD(*X$*)

Convert single- (CVSBD) and double- (CVDBCD) precision numbers from the format used by the Decimal version of BASIC to the format used by the Binary version. This is necessary if a program written in the binary version is to read random-access files created by a program written in the Decimal version.

DATA statement, 132

DATA *constant-list*

Used to store numeric and string constants that will be read by a READ statement. DATA statements can contain as many constants, separated by commas, as will fit on a line. String constants need be enclosed in quotation marks only if they contain

commas, colons, or significant leading or trailing spaces. DATA statements can be scattered throughout a program, and will be read by READ statements in the order in which they appear.

DATE$ statement

DATE$ = *string-expression*

Allows you to set the Macintosh clock to a new date. This could be useful if you are running a program that uses the DATE$ function to date-stamp reports and you want to produce reports for dates other than today. The date must be entered in one of these formats:

mm-dd-yy
mm-dd-yyyy
mm/dd/yy
mm/dd/yyyy

DATE$ function MC

DATE$

Returns the date according to the Macintosh clock.

DEF FN statement

DEF FN *name*[(*parameter-list*)] = *function-definition*

Allows you to create your own functions, to supplement those provided by BASIC. The *name* parameter entered immediately after DEF FN (no space between them) is the name you will use to call the function from within your program. The *parameter-list* represents the variables that will be passed to the function when it is called, and the *function-definition* is the actual expression that performs the operation. DEF FN is conceptually similar to a subprogram, but it is limited in length to only one line.

DEFINT statement, 67

DEFINT *letter-range*

DEFSNG statement, 67

DEFSNG *letter-range*

DEFDBL statement, 67

DEFDBL *letter-range*

DEFSTR statement, 67

DEFSTR *letter-range*

Used to declare groups of variables starting with a letter in the *letter-range* to be integer, single-precision, double-precision, or string variables. The subsequent use of type declaration characters (% for integer, ! for single-precision, # for double-precision, and $ for string) to declare individual variables takes precedence over the blanket declaration of a DEF *type* statement. If a variable is encountered that is not covered by one of these statements and that is not followed by a declaration character, then in the Binary version of BASIC it defaults to single precision, and in the Decimal version to double precision. Most of my programs start by defining all variables as integers, to increase the speed at which the program runs.

DELETE statement

DELETE [*line*][- *line*]

Removes the specified line or range of lines from the program.

DIALOG function, 61

DIALOG(*n*)

Returns information about several types of events that might be taking place on the screen. There are six possible values of the parameter *n,* ranging from 0 through 5. See the text for examples of the use of the DIALOG function.

DIALOG ON statement, 122

DIALOG ON

Enables trapping of dialog events. Used after an ON DIALOG statement, which tells what to do when the event is trapped, to actually start trapping events.

DIALOG OFF statement, 326

DIALOG OFF

Disables the trapping of dialog events.

DIALOG STOP statement, 326

DIALOG STOP

Suspends the reporting of dialog events until the next DIALOG ON statement, at which time any events trapped during the suspension are made available.

DIM statement, 69

DIM *subscripted-variable-list*

Used to dimension variable arrays. The *subscripted-variable-list* contains one or more array names, each followed in parentheses by the maximum value of that array's subscript. Since the subscript can range from 0 through the maximum value, there can be one more element in the array than the maximum subscript.

EDIT FIELD statement, 220

EDIT FIELD *ID*[, *default, rectangle*[, [*type*][, *justify*]]]

Creates the stock Macintosh edit field used to enter information into a program.

EDIT FIELD CLOSE statement

EDIT FIELD CLOSE *ID*

Removes a previously defined edit field from the active window.

EDIT$ function, 223

EDIT$(*ID*)

Retrieves information entered in an edit field by the user.

END statement, 165

END

Terminates a program and returns to the List, Command, or Output window that was active in BASIC before the program was run.

EOF function, 447

EOF(*filenumber*)

Tests for the end of a specified file, returning −1 (true) when an attempt is made to read beyond the last record. Used to avoid the "input past end" error message.

ERASE statement

ERASE *array-variable-list*

Eliminates the specified arrays from memory, so they can be re-dimensioned or so that the space they occupied can be used in some other manner.

ERL function

ERL

Returns the number of the line on which the last error occurred. Does not return labels, so is of little use if you don't use numbers in your programs.

ERR function, 230

ERR

Returns the number of the last error to occur. Used after trapping an error (with the ON ERROR GOTO statement) to decide how to respond to it.

ERROR statement

ERROR *integer-expression*

If *integer-expression* is an existing BASIC error code, then that error is simulated when this statement is used.

EXIT SUB statement, 101

EXIT SUB

Used to branch out of a subprogram before the END SUB statement. There can be more than one EXIT SUB statement in a subprogram.

EXP function

EXP(*X*)

Returns e (base of natural logarithms) to the power of *X*.

FIELD statement, 476

FIELD [#]*filenumber, fieldwidth* AS *string-variable*...

Allocates space in a file buffer for variables in a random-access file.

FILES statement

FILES [*filespec*]

Prints a list of the files on the disk specified in *filespec*. If *filespec* is omitted, all files on the disk in the internal drive are listed.

FILES$ function, 64

FILES$(*n*[, *prompt-string*])

The parameter *n* can be either 0 or 1:

Value	Action
0	Produces dialog box similar to standard Save As dialog box and displays *prompt-string* to prompt user for name of file
1	Produces dialog box similar to standard Open dialog box and allows user to specify file to open

FIX function

FIX(*X*)

Returns the truncated integer part of *X*.

FOR...NEXT statement, 47

FOR *variable* = *x* TO *y* [STEP *z*]

NEXT [*variable*][, *variable*...]

Commonly referred to as a FOR-NEXT loop. Allows the section of program between the FOR statement and the NEXT statement to be run a number of times determined by the values of *x*, *y*, and *z*.

FRE function

FRE(*n*)

The parameter *n* determines the value returned by this function, as follows:

Value	Returns
−1	Number of bytes in Macintosh heap not being used by BASIC
−2	Number of bytes in stack that have never been used
any other number	Number of bytes in BASIC's data segment not being used

All three versions compact string space. A fourth version:

FRE("")

simply compacts string space.

GET statement, 76

GET [#]*filenumber*[, *recordnumber*]

Reads a record from a random-access file. The parameter *filenumber* is the number the file was opened under, and *recordnumber* is the number of the record to be read (defaults to the number after the last record read if omitted). A second version of the GET statement:

GET (*x1,y1*) - (*x2,y2*), *array-name* [(*index*[, *index*..., *index*])]

is called the screen GET. It retrieves the image from within the *(x1,y1)* - *(x2,y2)* boundaries on the screen and stores it in the specified integer array.

GOSUB...RETURN statement, 72

GOSUB *line*

RETURN [*line*]

Interrupts the normal linear flow of the program to divert it to a subroutine identified by the line number or label specified with the *line* parameter. The RETURN statement terminates the subroutine. If the *line* option is not used, program flow continues with the statement after the most recent GOSUB. If *line* is specified, the program continues at the specified line.

GOTO statement, 10

GOTO *line*

Diverts the program to the specified line.

HEX$ function, 105

HEX$(*X*)

Returns a string representing the hexadecimal value of the decimal argument *X*.

IF...THEN...ELSE statement, 65

IF *expression* THEN *then-clause* [ELSE *else-clause*]

Tests the *expression,* and if it is *true* executes the *then-clause.* If the *expression* is *false* and the optional *else-clause* is included, then it is executed, otherwise the program continues with the statement on the next line.

IF...GOTO...ELSE statement

IF *expression* GOTO *line* [ELSE *else-clause*]

Diverts the program to *line* if it is *true.* If it is *false,* the *else-clause,* if present, is executed, otherwise the program continues with the next line.

INKEY$ function, 163

INKEY$

Used to retrieve single characters entered from the keyboard. When invoked it returns a one-character string consisting of the oldest character in the keyboard buffer. If the buffer is empty, it returns a null string.

INPUT statement

INPUT[;]*prompt-string;*] *variable-list*

Allows variables to be assigned strings and values from the keyboard during program execution. If the optional *prompt-string* is included, it is displayed on the screen and the program waits for a reply. One or more items can be entered, separated by commas, and are assigned to the variables in *variable-list.* The type of each entered item (integer, string, and so on) must match the associated variable in the list. If you're entering just text, the EDIT FIELD statement is a cleaner method of accomplishing the same thing.

INPUT$ function, 63

INPUT$(*X*[, [#]*filenumber*])

Returns a string of *X* characters from *filenumber,* or from the keyboard if *filenumber* is not specified.

INPUT # statement

INPUT *#filenumber, variable-list*

Reads items from the sequential file previously opened as *filenumber,* and assigns them to the variables in *variable-list.*

INSTR function, 206

INSTR([*I,*]*X$, Y$*)

Returns the position (in characters) of the first occurrence of *Y$* in *X$.* If *X$* does not contain *Y$,* then INSTR returns 0. The optional offset, *I,* determines the starting position for the search.

INT function, 97

INT(*X*)

Returns the largest integer less than or equal to *X*.

KILL statement

KILL *filespec*

Deletes the file *filespec*.

LBOUND function

LBOUND(*array-name*[, *dimension*])

UBOUND function

UBOUND(*array-name*[, *dimension*])

Return the minimum and maximum subscripts of *array-name*. The *dimension* option is used to specify which dimension to test in a multi-dimension array.

LCOPY statement

LCOPY

Prints a copy of the Macintosh screen on a graphics printer.

LEFT$ function, 173

LEFT$(*X$*, *I*)

Returns a string composed of the leftmost *I* characters of *X$*.

LEN function, 209

LEN(*X$*)

Returns the number of characters, including nonprinting characters and blanks, contained in *X$*.

LET statement

[LET] *variable* = *expression*

Assigns the value of an expression to a variable. The expression and variable must be of the same type (integer, string, and so on). Notice that LET is optional.

LINE statement, 117

LINE [[STEP] (*x1, y1*)] - [STEP] (*x2,y2*)[, [*color*][, b[f]]]

Draws a line, a box, or a filled box in the current output window. In its simplest form, without any of the optional parameters, draws a line from point (*x1,y1*) to point (*x2,y2*). Adding the *b* parameter causes it to draw a box with its opposite corners at these two points. The *f* parameter fills the box. The *color* can be either black or white, with black being the default if none is specified. The STEP option, which can be used before either or both sets of coordinates, causes the coordinate to be considered relative to the current position of the graphic pen.

LINE INPUT statement

LINE INPUT[;] [*prompt-string*;] *string-variable*

Displays (optionally) a prompt, and assigns everything typed by the user—up to the carriage return ending the line—to *string-variable.*

LINE INPUT # statement, 444

LINE INPUT *#filenumber, string-variable*

Reads an entire line (up to the carriage return) from a sequential file, and assigns it to *string-variable.*

LIST statement

LIST [*line*]

LIST [*line*][- [*line*]], *filename*

Prints all or a specified portion of the program currently in memory to the List window, to a disk file, or to the printer (remember that devices can be opened as files).

LLIST statement

LLIST [*line*][- [*line*]]

Prints all or a specified portion of the current program on the printer.

LOAD statement

LOAD [*filespec*[, *R*]]

Loads a file from disk (and automatically runs it if the *R* option is included). Without the *R* this is essentially the same as choosing Open from the File menu, and if *filespec* is not included, the standard Open dialog box appears to prompt for the name of the file to load.

LOC function, 162

LOC(*filenumber*)

Returns information about the specified *filenumber*. The information returned depends on the type of file represented by *filenumber*. For random-access files, LOC returns the number of the most recently read or written record. For sequential files, it returns the increment, which is the total number of bytes written to or read from the file divided by the record size. For files opened to KYBD:, LOC returns the value 1 if there are any characters in the keyboard buffer waiting to be read, and returns the value 0 if there are none. For files opened to CLIP: or COM:, LOC returns the number of characters waiting to be read.

LOCATE statement

LOCATE [*row*][, *column*]

Positions the pen at a specified column and line in the current output window. Like the CSRLIN function, this is really a leftover from previous versions of BASIC. The same task can be more accurately accomplished with the MOVETO ROM call.

LOF function, 63

LOF(*filenumber*)

Returns the length, in bytes, of the file opened as *filenumber*.

LOG function

LOG(*X*)

Returns the natural logarithm (log base e) of *X*. *X* must be greater than zero.

LPOS function

LPOS(*X*)

Returns the current position of the line printer's print head within the line printer buffer; this may not be the physical position of the print head over the paper.

LPRINT statement

LPRINT [*expression-list*]

Prints *expression-list* on the printer, just as the PRINT statement would print it on the screen.

LPRINT USING statement

LPRINT USING *string-expression; expression-list*

Prints *expression-list* to the printer using formatting specified by *string-expression.* Quite a variety of formatting expressions can be used to control the appearance of your printed text and numbers: See the Microsoft BASIC manual for a complete list.

LSET statement, 477

LSET *string-variable = string-expression*

RSET statement, 477

RSET *string-variable = string-expression*

Moves string variables from memory to a random file buffer in preparation for using the PUT statement to store them in a random-access file. If *string-variable* is shorter than the space allocated to *string-expression* in the FIELD statement, LSET left justifies and RSET right justifies it.

MENU statement, 158

MENU

MENU *menu-ID, item-ID, state*[, *title-string*]

Without parameters, restores the currently selected menu header to the normal (unselected) black-on-white video. The version with parameters allows you to create a custom menu. This statement is used once for the header and for each item you want to appear on the menu.

The *menu-ID* can be a number from 1 through 10, and indicates the position across the menu bar where the menu will appear. The *item-ID* can range from 0 through 20, with 0 indicating the command applies to the name or condition of the entire menu. The *state* argument can range from 0 through 2: 0 disables the menu or the item, 1 enables it, and 2 enables it and also places a check mark by it. The *title-string* is the text that appears on this particular menu item.

MENU function, 193

MENU(*n*)

Returns information about a menu selection. The information returned depends on the value of *n* which can be either 0 or 1. See the text for examples of the usage of this function.

MENU ON statement, 160

MENU ON

Enables the trapping of menu events—the selection of an item from a menu. This statement is used after an ON MENU...GOSUB, which specifies where to go when the event is trapped.

MENU OFF statement, 326

MENU OFF

Disables the trapping of menu events.

MENU RESET statement, 188

MENU RESET

Restores BASIC's default menu bar, after you have created a custom menu.

MENU STOP statement, 202

MENU STOP

Suspends the reporting of menu events, but keeps track of them and reports them when event trapping is again enabled.

MERGE statement

MERGE *filespec*

Appends *filespec*, which must be a file that was saved in the ASCII format, to the end of the program currently in memory.

MID$ statement, 217

MID$(*string-exp1*, *n*[, *m*]) = *string-exp2*

Replaces a portion of *string-exp1*, beginning at position *n*, with *m* characters from *string-exp2*. If *m* is omitted, then all of *string-exp2* is used—as long as the length of *string-exp1* is not increased.

MID$ function, 217

MID$(*X$*, *n* [, *m*])

Returns a string of *m* characters from *X$*, beginning with the *n*th character.

MKI$ function, 477

MKI$(*integer-expression*)

MKS$ function, 477

MKS$(*single-precision-expression*)

MKD$ function, 477

MKD$(*double-precision-expression*)

Convert numeric values to string variables in order to store them in random-access files.

MKSBCD$ function

MKSBCD$(*single-precision-expression*)

MKDBCD$ function

MKDBCD$(*double-precision-expression*)

Convert single- and double-precision numbers from Binary BASIC storage format to Decimal BASIC storage format, prior to storing them in a random-access file.

MOUSE function, 38

MOUSE(*n*)

Returns information about the state of the mouse button or the location of the mouse pointer relative to the upper left corner of the current output window. The information returned by the MOUSE function is determined by the value of the argument *n*, which can range from 0 through 6. See the text for examples of each type of information.

MOUSE ON statement, 326

MOUSE ON

Enables trapping of mouse events. Used in conjunction with an ON...MOUSE GOSUB statement, which specifies where to go when the event is trapped.

MOUSE OFF statement

MOUSE OFF

Disables trapping of mouse events.

MOUSE STOP statement, 326

MOUSE STOP

Suspends trapping of mouse events until the next MOUSE ON, at which time any events that occurred during the suspension are reported.

NAME statement

NAME *old-filename* AS *new-filename*[, *filetype*]

Used to rename a file that exists on disk with a file name not currently in use on the disk. You cannot change the volume name, which would place the file on a different disk, with this command, but you can change the four-letter code used to indicate the type of file.

NEW statement

NEW

Can be entered only in the Command window, and has the same effect as choosing New from the File menu.

NEXT statement, 47

NEXT [*variable*[, *variable*...]]

The end of a FOR...NEXT loop. If the *variable* is omitted the NEXT is applied to the most recent FOR. If FOR...NEXT loops are nested, multiple variables may be listed after one NEXT statement to terminate more than one loop.

OCT$ function

OCT$(*X*)

Returns a string that represents the octal value of the decimal argument *X*.

ON BREAK GOSUB statement

ON BREAK GOSUB *line*

Specifies the subroutine that program control will be transferred to if the user presses Command-period or chooses Stop from the Run menu. The actual trapping of the break event does not take place until after a BREAK ON statement is issued. The *line* parameter can be any valid line number or label, or can be 0 to disable event trapping, or RETURN to ignore the trapped event.

ON DIALOG GOSUB statement, 121

ON DIALOG GOSUB *line*

Transfers program control to the subroutine beginning at *line* (or label) when a dialog-box-related event is trapped. This would be any event that causes DIALOG(0) to be something other than zero. ON DIALOG GOSUB has no effect until dialog trapping is enabled by the DIALOG ON statement.

ON ERROR GOTO statement, 229

ON ERROR GOTO *line*

Transfers program control to an error-handling routine if any error occurs subsequent to this command. Using an ON ERROR GOTO 0 statement in a program disables error trapping.

ON...GOSUB statement, 123

ON *expression* GOSUB *line-list*

Called a computed GOSUB statement. Computes the value of *expression* (rounding it to an integer if it is not already an integer) and branches to the subroutine in the *line-list* that is that value into the list. For example, if the value is 3, the program branches to the third subroutine in the list; if the value is 7, the program branches to the seventh subroutine.

ON...GOTO statement, 125

ON *expression* GOTO *line-list*

Same as the ON...GOSUB statement, except that the program branches to a line or label that is not the beginning of a subroutine.

ON MENU GOSUB statement, 159

ON MENU GOSUB *line*

Transfers program control to a subroutine when the user selects a menu item. The statement specifies which line or label the program will branch to, and becomes effective after a MENU ON statement.

ON MOUSE GOSUB statement

ON MOUSE GOSUB *line*

Transfers program control to a subroutine when the user presses the mouse button. This statement does not take effect until enabled by a MOUSE ON statement.

ON TIMER...GOSUB statement

ON TIMER (*n*) GOSUB *line*

Transfers program control to the specified subroutine every *n* seconds, where *n* is an integer from 1 through 86400 (the number of seconds in 24 hours). This event trap is disabled by re-issuing the statement with *line* equal to 0.

OPEN statement, 61

OPEN *mode*, [#]*filenumber, filespec*[, *file-buffer-size*]

OPEN *filespec* [*FOR mode*] *AS* [#]*filenumber* [*LEN* = *file-buffer-size*]

Used to open a file or a device for input or output. See text for examples of the various formats.

OPTION BASE statement

OPTION BASE *n*

Declares the minimum value for array subscripts to be either 0 or 1 (default = 0).

PEEK function

PEEK(*I*)

Returns the byte read from memory location *I*.

PICTURE statement, 66

PICTURE [(*x1,y1*)[- (*x2,y2*)]][, *P$*]

Displays a picture (produced by screen graphics statements) that was previously recorded using the PICTURE ON statement. If the optional coordinates are given, the picture is scaled to fit in the area defined; if only the first coordinate is given, the picture is drawn full-scale with its upper left corner at that coordinate. The variable *P$* is the string in which PICTURE ON stored the picture; if omitted, the most recently stored picture is displayed.

PICTURE ON statement, 80

PICTURE ON

PICTURE OFF statement, 80

PICTURE OFF

Starts and stops the recording of screen graphics statements (such as LINE, CLS, CIRCLE, PRINT, and ROM calls). These recorded statements can be stored as a string and later repeated with the PICTURE statement to reproduce their effect.

PICTURE$ function, 81

PICTURE$

Returns a string consisting of a set of encoded Macintosh instructions that recreate the effect of the graphics statements between the last PICTURE ON and PICTURE OFF statements.

POINT function

POINT (*x,y*)

Returns the color (black = 33, white = 30) of the specified point on the screen.

POKE statement

POKE *I, J*

Writes the byte *J* into memory location *I*.

POS function

POS(*I*)

Returns the current column position of the cursor. *I* is a dummy argument and has no significance.

PRESET statement, 357

PRESET [STEP] (*x,y*)[, *color*]

Draws a point in the current output window at the location specified by the coordinates *(x,y)*. If the STEP option is used, these coordinates are relative to the current pen position; otherwise they are relative to the upper left corner of the window. If *color* is 30, the point is white (the default); if it is 33, the point is black.

PRINT statement, 36

PRINT [*expression-list*]

Displays *expression-list* on the screen. Numbers, or expressions that evaluate to numbers, are separated in the *expression-list* by commas or semicolons; string expressions are enclosed in quotation marks and separated by commas or semicolons. If an expression is separated from the previous expression with a comma, it is printed at the next comma stop, a position determined by the WIDTH statement. If it is separated by a semicolon, it is printed adjacent to the previous expression.

PRINT USING statement, 105

PRINT USING *string-expression; expression-list*

A version of the PRINT statement that allows the formatting of the printed display to be determined by characters, such as !, //, &, and # that are included in *string-expression.*

PRINT # statement, 165

PRINT *#filenumber, expression-list*

PRINT # USING statement

PRINT *#filenumber,* [USING *string-expression*;] expression-list

Like PRINT and PRINT USING, except output goes to a sequential file instead of to the screen.

PSET statement, 356

PSET [STEP] *(x,y)*[, *color*]

Draws a point in the current output window at the location specified by the coordinates *(x,y).* If the STEP option is used, these coordinates are relative to the current pen position; otherwise they are relative to the upper left corner of the window. If *color* is 30, the point is white; if it is 33, the point is black (the default).

PTAB function

PTAB(*X*)

Moves the print position (horizontally) to pixel *X* on the screen.

PUT statement, 77

PUT [#]*filenumber*[, *recordnumber*]

The complement to the random-file GET statement, PUT transfers data from a random file buffer to the file. A second version:

PUT(*x1,y1*) [- (*x2,y2*)], *array*[(*index*[, *index*..., *index*])][, *action-verb*]

is the complement to the screen GET statement. It places a graphic image (previously stored by the GET statement) on the screen, scaled to fit within the boundaries of the specified coordinates. If only the first coordinate is included, the image is reproduced at full-scale, with its upper left corner at that coordinate. The *array(index)* parameter provides the starting point, in the array filled by the GET statement, of the image to be reproduced. The *action-verb*—PSET, PRESET, AND, OR, or XOR (default)—determines the effect of the stored image on the color of the pixels it is reproduced on top of. Experiment with this statement, using each *action-verb*.

RANDOMIZE statement, 282

RANDOMIZE [*expression*]

Reseeds (provides a new starting point) for the random-number generator. The argument *expression* must be a valid integer (−32768 through 32767) or the BASIC function TIMER, which returns the number of seconds since midnight.

READ statement, 120

READ *variable-list*

Assigns numeric or string values in DATA statements to corresponding numeric or string variables in the *variable-list*.

REM statement, 8

REM *remark*

A nonexecutable statement placed in a BASIC program, usually for explanatory purposes. The REM statement can be replaced by an apostrophe.

RESET statement, 165

RESET

Closes all open files and updates directory information on disk.

RESTORE statement, 167

RESTORE [*line*]

Specifies where the next READ will get its data. DATA statements are normally read sequentially from the the first one in the program to the last one. The RESTORE statement redefines the beginning point and restarts the read cycle. If the *line* argument is omitted, the next READ starts with the first DATA statement in the program.

RESUME statement, 232

Continues program execution after the program has branched to an error-recovery routine. The version you use determines where the program resumes.

RESUME

RESUME 0

Resumes at the statement that caused the error; presumably action was taken in the error-handling routine that alleviated the cause of the error.

RESUME NEXT

Resumes at the statement immediately following the one that caused the error.

RESUME *line*

Resumes at *line*. Note that *line* cannot be within a subprogram.

RETURN statement, 72

RETURN [*line*]

Ends a subroutine and transfers program control to another place in the program. Used without the *line* option, program execution continues with the statement immediately following the last executed GOSUB. Used with the *line* option, execution continues at the specified line number or label.

RIGHT$ function, 206

RIGHT$(*X$, I*)

Returns the rightmost *I* characters of *X$*.

RND function, 286

RND[(*X*)]

Returns a random number between 0 and 1. The argument *X* determines which number in the random-number sequence is returned: if *X* is less than 0, the sequence is restarted; if *X* is greater than 0 or omitted, then the next random number in the sequence is returned; if *X* is equal to 0, the last number returned is repeated.

RSET statement, 477

RSET *string-variable* = *string-expression*

Moves string variables from memory to a random file buffer in preparation for using the PUT statement to store them in a random-access file. If *string-variable* is shorter than the space allocated to *string-expression* in the FIELD statement, RSET right justifies it.

RUN statement

RUN [*line*]

Starts execution of the program in memory, at the specified line if it is included.

RUN *filename*[, *R*]

Loads *filename* and starts execution. If the *R* option is included, all data files that are open remain open.

SAVE statement

SAVE [*filename*[, *A*]]

SAVE [*filename*[, *P*]]

SAVE [*filename*[, *B*]]

Saves the program currently in memory to a disk file with the name *filename*. If *filename* is not included in the statement, a dialog box appears requesting it. The *A*, *P*, and *B* options determine the format in which the program is saved: *A* is ASCII; *P* is protected (can't be listed or edited); and *B* is compressed binary (takes less space than ASCII).

SCROLL statement, 357

SCROLL *rectangle, delta-x, delta-y*

Moves the defined *rectangle* to the right or left *delta-x* pixels, and up or down *delta-y* pixels.

SGN function, 288

SGN(*X*)

Returns a number that indicates whether *X* is greater than 0 (returns 1), equal to 0 (returns 0), or less than 0 (returns −1).

SHARED statement, 100

SHARED *variable-list*

Used within a subprogram to specify that variables in *variable-list* are common to variables of the same name in the main program.

SIN function, 289

SIN(*X*)

Returns the sine (in a right triangle, the ratio of the side opposite an acute angle to the hypotenuse) of *X*, where *X* is given in radians.

SOUND statement

SOUND *frequency, duration*[, [*volume*][, *voice*]]

Causes the Macintosh speaker to emit noise that some say passes for music. The *frequency* argument indicates the pitch in cycles per second, *duration* indicates the length of time the tone lasts (can vary from 1 through 77, with each second equal to 18.2), *volume* indicates the loudness (measured on a scale from 0 through 255), and *voice* indicates which of four possible soundtracks (created by the WAVE statement) is to be used.

SOUND RESUME statement

SOUND RESUME

Continues the execution of SOUND statements, including any queued during a SOUND WAIT.

SOUND WAIT statement

SOUND WAIT

Queues subsequent SOUND statements. Continues queuing until a SOUND RESUME statement is executed.

SPACE$ function

SPACE$(*X*)

Returns a string of spaces, of length *X*.

SPC function, 447

SPC(*I*)

Used in a PRINT or LPRINT statement to skip *I* spaces.

SQR function

SQR(*X*)

Returns the square root of *X*.

STOP statement

STOP

Terminates program execution and returns to the immediate mode.

STR$ function, 49

STR$(*X*)

Returns a string representation of the value of *X*.

STRING$ function

STRING$(*I, J*)

Returns a string *I* characters long, with each character equal to that represented by the ASCII code *J*.

STRING$(*I, X$*)

Returns a string *I* characters long, with each character the same as the first character of *X$*.

SUB statement, 99

SUB *subprogram-name* [(*formal-parameter-list*)] STATIC

Indicates the beginning of a subprogram. The *formal-parameter-list* is a list of variable names used in the subprogram that are matched to variables supplied in the statement that called the subprogram. This is a method of passing variables to a subprogram. The STATIC argument is required in versions of BASIC at least through 2.1, and means that all variables within the subprogram retain their value between times the subprogram is called.

END SUB statement, 101

END SUB

Terminates a subprogram and returns program flow to the statement following the one that called the subprogram. There can be only one END SUB statement in a subroutine.

EXIT SUB statement, 101

EXIT SUB

Used to branch out of a subprogram before the END SUB statement. There can be more than one EXIT SUB statement in a subprogram.

SWAP statement, 74

SWAP *variable, variable*

Exchanges the values of the two variables listed.

SYSTEM statement, 164

SYSTEM

Closes everything and returns to the Macintosh finder.

TAB function, 450

TAB(*I*)

Used in PRINT and LPRINT statements to move the print position to *I*, where *I* is the number of column spaces. If the current print position is already past *I*, it is moved to *I* on the next line.

TAN function

TAN(*X*)

Returns the tangent (in a right triangle, the tangent is the ratio of the side opposite a given obtuse angle to the side adjacent to the angle) of *X*, where *X* is an angle expressed in radians.

TIME$ statement

TIME$ = *string-expression*

Sets the Macintosh clock to the time specified in *string-expression*. The format for *string-expression* is *hh*[:*mm*[:*ss*]], with the omitted options defaulting to 0.

TIME$ function, 216

TIME$

Returns the current time according to the Macintosh clock.

TIMER function, 282

TIMER

Returns the number of seconds elapsed since midnight.

TIMER ON statement

TIMER ON

Enables event trapping based on time.

TIMER OFF statement

TIMER OFF

Disables event trapping based on time.

TIMER STOP statement

TIMER STOP

Suspends reporting of events based on time, but keeps track of them and reports them after the next TIMER ON statement.

TRON statement, 293

TRON

TROFF statement

TROFF

Turn tracing of program execution on and off.

UBOUND function

UBOUND(*array-name*[, *dimension*])

LBOUND function

LBOUND(*array-name*[, *dimension*])

Return the maximum and minimum subscript of the dimensions of an array.

UCASE$ function, 449

UCASE$ (*string-expression*)

Returns a string with all alphabetic characters in uppercase.

VAL function, 172

VAL(*X$*)

Returns the numeric value of *X$*.

VARPTR function

VARPTR(*variable-name*)

Returns the address in memory at which the first byte of the variable *variable-name* is stored. Usually used to pass the location of a variable or array to an assembly-language program.

WAVE statement

WAVE *voice*[, [*wave-definition*][, *phase*]]

Used to define multiple voices to be used by the SOUND statement.

WHILE...WEND statement, 37

WHILE *expression* [*statements*]

WEND

Executes the series of statements in the loop between WHILE and WEND as long as *expression* evaluates to a *true* condition.

WIDTH statement, 49

WIDTH *output-device,* [*size*][, *print-zone*]

Sets the width in characters of the print zone (tabs, comma stops) and the line on different output devices. The *output-device* may be SCRN:, CLIP:, COM1:, or LPT1:, with SCRN: as the default. The *size* argument is the maximum width of a line, given in the number of "standard" characters it will hold. With the proportionally spaced fonts of the Macintosh this is not very standard: The width of a numeral in the current font is the standard.

WIDTH *#filenumber,* [*size*][, *print-zone*]

Sets the line and print-zone width of a file opened as *filenumber.*

WIDTH [*size*][, *print-zone*]

Default syntax, applies to the screen (SCRN:).

WIDTH LPRINT [*size*][, *print-zone*]

An alternate to specifying LPT1: in the first syntax.

WIDTH function

WIDTH (*string-expression*)

Returns the width of a string, in pixels.

WINDOW statement, 33

WINDOW *ID*[, [*title*][, [*rectangle*][, *type*]]]

Creates a window with the given *ID* number and *title*. The window boundaries specified by *rectangle* are relative to the upper left corner of the screen. You can create two variations of each of the four window types that are standard in Macintosh applications. The *type* is a number from 1 through 4, or from −1 through −4, with the minus sign indicating that the window is modal, meaning that all activity is confined to that window.

WINDOW function, 34

WINDOW(*n*)

Returns six different types of information about output windows, with the type determined by the value of *n*, which can range from 0 through 5. See the text for examples of the use of this function.

WINDOW CLOSE statement, 34

WINDOW CLOSE *ID*

Closes the specified window.

WINDOW OUTPUT statement, 34

WINDOW OUTPUT *ID*

Makes the specified window the current output window, without making it the active window. This allows you to direct output to one window while soliciting input from another.

WINDOW OUTPUT *#filenumber*

Sends the results of graphic statements such as CIRCLE, PSET, PICTURE, and ROM calls to an output device other than the screen.

WRITE statement

WRITE [*expression-list*]

Displays the data in *expression-list* on the screen; multiple items in the list are separated by commas. WRITE displays positive numbers without the leading blank inserted by PRINT, which is occasionally useful.

WRITE # statement

WRITE *#filenumber, expression-list*

Places string and numeric variables into a sequential file, placing a comma between each and a quotation mark before and after each string variable.

ROM CALLS

You can use the BASIC CALL statement to access many of the machine-language routines stored by Apple in the Macintosh ROM. These are the same QuickDraw graphic calls used by commercial application programs to rapidly create shapes and patterns, and to control text and cursor attributes. There are several concepts common to many of these calls:

- With several exceptions, the word *CALL* and the outside set of parentheses around the parameters are optional.

- The variables passed are expected to be integers: Append the % sign to each or use the DEFINT statement in the calling program to define an entire class of variables as integer.
- The usual method of passing array variables is to reference the first element of the array with the VARPTR function, which passes the memory location of that element to the machine-language routine.
- The screen coordinates passed refer to the upper left corner of a pixel.
- If a call, such as the pen-pattern call, sets an attribute, the attribute applies only to the current output window, but is stored with that window. If you move between windows, the last attribute that was set in each window comes into effect.

General-purpose calls

CALL BACKPAT

CALL BACKPAT(VARPTR(*pattern%(0)*))

Sets the background pattern for the output window. Before calling BACKPAT, four elements that describe the pattern must be stored in the *pattern* array.

CALL ERASE...

See specific family in Shape section.

CALL FILL...

See specific family in Shape section.

CALL FRAME...

See specific family in Shape section.

CALL GETPEN

CALL GETPEN(VARPTR(*penlocation%* (*n*)))

Returns the current location of the graphic pen: If *n* is 0 the vertical coordinate is returned; if *n* is 1 the horizontal coordinate is returned.

CALL HIDECURSOR, 355

CALL HIDECURSOR

Makes the cursor invisible. It still exists, its location and condition are still trapped with the MOUSE function, and it functions as usual if you click or drag with it.

CALL HIDEPEN

CALL HIDEPEN

Turns off the visible output of the pen. The lines that you draw exist, but they can't be seen.

CALL INITCURSOR

CALL INITCURSOR

Resets the mouse cursor to the standard arrow shape, and makes it visible if it has been made invisible with HIDECURSOR or OBSCURECURSOR.

CALL INVERT...

See specific family in Shape section.

CALL LINE, 118

CALL LINE (*xdelta,ydelta*)

Draws a line from the current pen location to the location *xdelta* to the right (or left if *xdelta* is negative) and *ydelta* down (or up if *ydelta* is negative). This is a case in which CALL and the parentheses are required, to avoid confusion with the BASIC LINE statement.

CALL LINETO, 49

CALL LINETO (*x,y*)

Draws a line from the current pen location to the specified coordinates. The pen returns to the current location after drawing the line.

CALL MOVE

CALL MOVE (*xdelta,ydelta*)

Moves the pen to the position *xdelta* horizontally and *ydelta* vertically from the current position. Positive values of *xdelta* and *ydelta* move the pen to the right and down, negative values move it to the left and up.

CALL MOVETO, 36

CALL MOVETO (*x,y*)

Moves the pen to the specified (*x,y*) coordinate.

CALL OBSCURECURSOR, 355

CALL OBSCURECURSOR

Makes the mouse cursor invisible until the mouse is moved.

CALL PAINT...

See specific family in Shape section.

CALL PENMODE, 70

CALL PENMODE (*mode*)

Determines how subsequent graphic calls affect existing screen images.

CALL PENNORMAL

CALL PENNORMAL

Restores the pen characteristics to the default size (1 pixel by 1 pixel), pattern (black), and mode (copy).

CALL PENPAT, 88

CALL PENPAT(VARPTR(*pattern%*(0)))

Assigns the pattern stored in the *pattern* array to the pen. Applies only to lines drawn by ROM calls, not to lines and circles created by BASIC statements.

CALL PENSIZE, 118

CALL PENSIZE (*width, height*)

Determines the width and height, in pixels, of the pen point.

CALL SETCURSOR

CALL SETCURSOR(VARPTR(*cursor%*(0)))

Assigns a new shape to the mouse cursor by defining a 16- by 16-bit image in the *cursor* array.

CALL SHOWCURSOR, 355

CALL SHOWCURSOR

Makes the cursor visible. Used to restore the cursor after a HIDECURSOR call.

CALL SHOWPEN

CALL SHOWPEN

Used after a HIDEPEN call to turn the pen output back on.

CALL TEXTFACE, 473

CALL TEXTFACE (*face*)

CALL TEXTFONT, 46

CALL TEXTFONT (*font*)

CALL TEXTMODE, 157

CALL TEXTMODE (*mode*)

CALL TEXTSIZE, 47

CALL TEXTSIZE (*size*)

Set text characteristics.

Shapes

The ROM calls in this section create and modify geometric shapes. Common to all is the technique of defining the top, left, bottom, and right boundaries of a rectangle that the shape will fit within. There are five operations that can be performed with each shape:

Call	Action
FRAME	Draws outline of shape
PAINT	Paints shape with current pen pattern
ERASE	Paints shape with current background pattern
INVERT	Inverts pixels enclosed by shape
FILL	Fills shape with specified pattern

...ARC, 376

CALL FRAMEARC(VARPTR(*rectangle%*(0)), *startangle, arcangle*)

CALL PAINTARC(VARPTR(*rectangle%*(0)), *startangle, arcangle*)

CALL ERASEARC(VARPTR(*rectangle%*(0)), *startangle, arcangle*)

CALL INVERTARC(VARPTR(*rectangle%*(0)), *startangle, arcangle*)

CALL FILLARC(VARPTR(*rectangle%*(0)), startangle, arcangle,
VARPTR(*pattern%*(0)))

In addition to the usual array that defines the *rectangle* the shape will fit within, these calls include variables to specify the *startangle* and *arcangle* (and the *pattern* for FILLARC). Angles are expressed in degrees, either positive or negative from zero at the top center of the *rectangle.* The *startangle* is the point at which the arc starts, and *arcangle* is the number of degrees spanned by the arc.

...OVAL, 126

CALL FRAMEOVAL(VARPTR(*rectangle%*(0)))

CALL PAINTOVAL(VARPTR(*rectangle%*(0)))

CALL ERASEOVAL(VARPTR(*rectangle%*(0)))

CALL INVERTOVAL(VARPTR(*rectangle%*(0)))

CALL FILLOVAL(VARPTR(*rectangle%*(0)), VARPTR(*pattern%*(0)))

The *rectangle* argument describes the boundaries the oval will fit within; if its height and width are equal, a circle will be drawn.

...POLY, 309

CALL FRAMEPOLY(VARPTR(*polygon%*(0)))

CALL PAINTPOLY(VARPTR(*polygon%*(0)))

CALL ERASEPOLY(VARPTR(*polygon%*(0)))

CALL INVERTPOLY(VARPTR(*polygon%*(0)))

CALL FILLPOLY(VARPTR(*polygon%*(0)), VARPTR(*pattern%*(0)))

The shapes created by this set of ROM calls can be composed of any number of connected lines. The number of elements in the *polygon* array is determined by the number of lines in the polygon: There will be five elements plus two for each corner of the polygon. Of the five basic elements, the first one specifies how many elements are in the array and the second through the fifth define the usual upper, left, lower, and right boundaries of the shape. The remaining elements are in groups of two, with the first specifying the y coordinate of a corner and the second specifying the x coordinate of the same corner (note the reversal of the normal (x,y) order). The corner coordinates are given in the order the corners are to be drawn.

...RECT, 74

CALL FRAMERECT(VARPTR(*rectangle%*(0)))

CALL PAINTRECT(VARPTR(*rectangle%*(0)))

CALL ERASERECT(VARPTR(*rectangle%*(0)))

CALL INVERTRECT(VARPTR(*rectangle%*(0)))

CALL FILLRECT(VARPTR(*rectangle%*(0)), VARPTR(*pattern%*(0)))

The shapes created or modified by these ROM calls are rectangles defined by the *rectangle* array.

...ROUNDRECT

CALL FRAMEROUNDRECT(VARPTR(*rectangle%*(0)))

CALL PAINTROUNDRECT(VARPTR(*rectangle%*(0)))

CALL ERASEROUNDRECT(VARPTR(*rectangle%*(0)))

CALL INVERTROUNDRECT(VARPTR(*rectangle%*(0)))

CALL FILLROUNDRECT(VARPTR(*rectangle%*(0)), VARPTR(*pattern%*(0)))

These ROM calls create rectangles with rounded corners: The shape of the corner is determined by the arguments *ovalwidth* and *ovalheight,* which specify the number of pixels in each direction of the radius of the corner.

A Few Short Utility Programs

Appendix B

The word *utility* implies usefulness and practicality—whether it is referring to water and power companies (public utilities), baseball players (utility infielders and outfielders), or computer programs. All computer programs, of course, should be useful, but utilities are a special class of useful programs. Typically they perform a single task—often of a housekeeping nature, such as reconstructing a damaged file, copying a disk, or displaying some sort of information. The Single-Drive Copy program supplied with the Macintosh is an example of a utility, as are the items available on the Apple menu (the Alarm Clock, Calculator, and so on).

Utility programs often evolve through dissatisfaction and disaster. Many times, a few minutes spent writing a short utility program can save the day—or at least several hours of it—if it occurs to you to write the program. The incident that resulted in the first utility we will look at is an example of this.

As a person who writes for Microsoft Press—often about Microsoft products—I enjoy the mixed blessing of early access to many of their programs. Microsoft Word is one such program. When I started working on this book, the programmers had *almost* all the bugs worked out of the Macintosh version of Word, so I decided to use it. There were minor frustrations, but I saved my work often, and the few times the system crashed it left a "temporary" or "rescue" file from which I could recover all but the last few paragraphs. Until 2 o'clock on the morning of January 6th, when I uncovered a bug—and it bit me. Tired and ready to crawl into bed, I did a final save and chose Quit from the File menu. The program announced that I had unsaved changes to the glossary and suggested I save them as Standard Glossary, which I agreed to. When all the whirring stopped and I was looking at the desktop, I woke up in a hurry: My chapter was gone! Totally. No temporary file, no rescue file, nothing. My backup copy

on a different disk was only six hours old, but those six hours represented 5000 characters of more or less creative effort.

While desperately rummaging through the disk directory, I clicked *Display by size* and discovered that the Standard Glossary file was 67K bytes, rather than its usual 700 or so bytes. My entire chapter had been absorbed into the glossary file. After half an hour of trying to get Word to open the glossary, I was tired again and went to bed, resolved to retype the missing 5K in the morning.

I am almost embarrassed to admit that it never occurred to me to write a BASIC utility to recover the file. Fortunately, the first person I complained to in the morning suggested I do that, and ten minutes later I'd recovered my work. If I had just stopped to think for a few minutes the night before, I could have solved the problem then and had a much better night's sleep.

The reason I am taking up space with this personal confession is that I have noticed I am not the only person who overlooks the obvious. When you run into a problem, especially if it has to do with information stored in a file, stop for a moment to consider whether any of your BASIC skills can be of use. "Professionally" written programs are often available that will *almost* solve the problem, but the difference between almost and entirely might be several hours of frustration. Try looking to your own talents when faced with a problem. And don't wait until a problem comes along to think about writing a BASIC utility. Each time you perform some routine, repetitive task, ask yourself whether a BASIC program could handle it for you.

A file-recovery program

Word no longer does the Standard Glossary trick, and with any luck you will never lose a file—to Word or to any other program. But you can still profit from this recovery utility, as it is the base for a variety of others.

The first stage involved in recovering my file from Standard Glossary was discovering just what the glossary file contained. To do this, I wrote the short program shown in Figure B-1.

The *LINE INPUT #1* statement reads one "line" from the sequential file opened as file #1. A line, in this case, is defined as all characters up to a carriage return, and since this file was created by Word, which places a carriage return only at the end of a paragraph, each line will be a paragraph from the file.

```
'* File display program
'*

   OPEN FILES$(1) FOR INPUT AS #1
   lineCount = 1
   WHILE NOT EOF(1)
      LINE INPUT #1, text$
      PRINT lineCount; text$
      lineCount = lineCount + 1
   WEND
```

Figure B-1. Displaying a file

The *LINE INPUT # filenumber, string-variable* statement reads a paragraph and assigns it to the *string-variable* included in the statement—*text$* in this program. A string variable can be up to 32,767 characters long, which is certainly longer than any paragraph we are likely to encounter.

The PRINT statement prints a line number and then the line—one long line, stretching off the right side of the Macintosh screen. It wouldn't take much more effort to neatly wrap the lines around so the entire paragraph would be displayed. We will do just that in another utility later in this appendix, but this display is adequate for the job at hand, which is seeing what is in the glossary.

I included the line numbers because the section I wanted to recover was about three pages long and buried in the middle of forty pages. There was no point in taking the time or the disk space to copy the entire file. I was sure I could spot the beginning and ending points of the section by watching the opening line of each paragraph scroll past on the screen. And I did: The section started around line 250 and ran to about line 265. A quick edit changed the program to the one in Figure B-2 (shown on the next page), and a few minutes later I had recovered my work of the previous night. These programs are short and simple, but they did the job.

An outline program

The second utility we will look at is another program born out of frustration. My publisher asked me to produce outlines of two manuscripts I had almost completed. Having been through this routine before, I had worked out an outlining method: After loading Word and opening two document windows, I would load a chapter into

```
'* File recovery program
'*

  OPEN "standard glossary" FOR INPUT AS #1
  OPEN "goodstuff" FOR OUTPUT AS #2
  FOR lineCount=1 TO 300
    LINE INPUT #1, text$
    IF lineCount > 245 AND lineCount < 271 THEN PRINT #2, text$
  NEXT
  CLOSE 1, 2
```

Figure B-2. Recovering the section

one window and then use the mouse to select and copy the section headings to the second window. Since I include typesetting code as I write, the headings were easy to spot, as each was preceded by the code AAA), BBB), or CCC).

I immediately set to work on the first book, and two hours later had a 12-page outline. I also had a sore back (concentration is tiring) and a desire not to repeat the process for the second book. I stopped to consider whether or not I was using the best method. My brain finally clicked into gear, and in less than 20 minutes I had both written a BASIC utility and used it to create the outline of the second book.

The outline program is really just a slight enhancement of the previous program. It opens a file for the outline, and then reads each chapter and writes specified portions of it to the outline file. The primary prerequisite for making this program work is having some way to identify the portions of the input file that you want to transfer to the output file. If there aren't natural markers, such as my typesetting codes, in the file, you may have to scroll through with your word processor and uniquely mark each segment to transfer. The BASIC program can, of course, delete the unwanted markers as it makes the transfer.

The program is listed in Figure B-3. Notice, in the *Begin* section, that the program can be terminated by not selecting a new input file from the standard *Open* dialog box provided by the FILES$(1) function. If a file is selected, its name is printed to the output file, with a carriage return—CHR$(13)—before and after it, as a divider between chapters.

The section labeled *Strip* checks the first four characters of each paragraph to see if they are the characters used to indicate a section heading. If so, it sets the value

```
'* Outlining program
'*

   DEFINT a-z
Begin:
   doc$ = FILES$(1)
   IF doc$ = "" THEN  END
   OPEN doc$ FOR INPUT AS #1
   outline$ = FILES$(0, "enter the outline file name")
   OPEN outline$ FOR APPEND AS #2
   PRINT #2, CHR$(13); doc$; CHR$(13)
Strip:
   WHILE NOT EOF(1)
      LINE INPUT #1, para$

      '*
      '* Extract first, second, and third level headings.
      '*
      lineStart$ = LEFT$(para$, 4)
      IF lineStart$ = "AAA)" THEN space = 0 : GOSUB PrintHead
      IF lineStart$ = "BBB)" THEN space = 4 : GOSUB PrintHead
      IF lineStart$ = "CCC)" THEN space = 8 : GOSUB PrintHead
   WEND
   GOTO Begin
PrintHead:
   para$ = MID$(para$, 5)
   PRINT SPC(space); para$
   PRINT #2, SPC(space); para$
   RETURN
```

Figure B-3. Creating an outline

of *space* to the appropriate number of spaces and sends the program to the section that's labeled *PrintHead,* where the typesetting code is removed and the indent and head are printed.

A program to count words and characters

The next program (Figure B-4 on the following pages) continues the theme of programs primarily of interest to writers. It displays a file while counting the number of characters and words. There are various reasons why you might want to know the

amount of printable text in a file; for example, the pay scale for magazine articles is often based on a character or word count. A side benefit of knowing these two figures is that you (or the Macintosh) can then compute the average word length, which is a good indicator of the reading level of the article and hence whether it is appropriate for the audience at which it is aimed.

```
'* Character and word counting program
'*

   DEFINT a - z
   WINDOW 1, , (2, 20) - (510, 340), 3                    'the output window

'*
'* Initialize variables.
'*
Begin:
   numChar = 0                                        'number of characters counted
   numWord = 0                                          'number of words counted
   numPara = 0                                       'number of paragraphs counted
   wide = 75                                          'width of output in characters
   lastBreak = 1                                                'place to break line
   count = 0                                     'number of characters in current line
   newLine$ = ""                                                      'blank line
   true = -1                                                        'logical true
   false = 0                                                       'logical false

'*
'* Give info and see if there is word to search for.
'*
GetWord:
   WINDOW 2, , (20, 200) - (430, 335), 4
   PRINT "This progam opens a file and counts the number of characters, "
   PRINT "words, and paragraphs. If you would like it also to count how"
   PRINT "many times a particular word occurs, enter that word here."
   EDIT FIELD 1,"", (20, 100) - (200, 115)
   BUTTON 1, 1, "Continue", (220, 100) - (280, 120)
   BUTTON 2, 1, "Quit", (320, 100) - (380, 120)
Waiting:
   event = DIALOG(0)
   IF event <> 1 AND event <> 6 THEN GOTO Waiting
```

Figure B-4. Counting words and characters

more...

```
   IF event = 1 AND DIALOG(1) = 2 THEN END                    'Quit button clicked
   keyWord$ = EDIT$(1)
   searchFor$ = UCASE$(keyWord$)
   IF searchFor$ <> "" THEN findFlag = true
   WINDOW CLOSE 2

'*
'* Get name of file to be searched.
'*
GetFile:
   doc$ = FILES$(1, "TEXTWORDWDBN")
   IF doc$ = "" THEN  END
   WINDOW CLOSE 2
   CLS
   OPEN doc$ FOR INPUT AS #1

'*
'* Start display and search.
'*
Start:
   WHILE NOT EOF(1)
      LINE INPUT #1, para$
      numPara = numPara + 1

      '*
      '* Check each letter in paragraph.
      '*
      FOR position = 1 TO LEN(para$)
         ltr =  ASC(MID$(para$, position, 1))
         count = count + 1
         IF ltr = 32 OR ltr = 45 THEN lastBreak = count       'space or hyphen
         newLine$ = newLine$ + CHR$(ltr)                        'form new line
         IF count = wide THEN GOSUB Prnt                        'check length
Skip:
      NEXT
      GOSUB Prnt
   WEND

'*
'* All done, print the findings.
'*
DisplayResults:
   WINDOW 2, , (20, 175) - (400, 335), 4
```

Figure B-4. Counting words and characters *(continued)*

more...

```
    PRINT CHR$(7); "Information about: "; doc$
    PRINT
    PRINT "characters"; TAB(20); "= "; numChar
    PRINT "words"; TAB(20); "= "; numWord
    PRINT "paragraphs"; TAB(20); "= "; numPara
    PRINT "average word length"; TAB(20); "= ";
    PRINT USING "##.##"; numChar / numWord
    IF NOT findFlag THEN GOTO skip2
    PRINT
    PRINT "Number of occurrences of "; CHR$(34);      'use CHR$(34) to print quote mark
    PRINT keyWord$; CHR$(34); " = "; occur
skip2:
    TEXTFACE 1
    PRINT
    PRINT "Click this window to continue";
    TEXTFACE 0
    WHILE MOUSE(0) = 0 : WEND
    CLOSE : CLEAR
    GOTO Begin

'*
'* Display document.
'*
Prnt:

    '*
    '* Determine how much to print.
    '*
    IF count < wide THEN lastBreak = count
    printedLine$ = LEFT$(newLine$, lastBreak)
    numChar = numChar + LEN(printedLine$)                'accumulate character count

    '*
    '* Print letters and count words.
    '*
    FOR location = 1 TO lastBreak
        ltr$ = (MID$(printedLine$, location, 1))
        PRINT ltr$;
        IF ASC(ltr$) = 32 THEN numWord = numWord + 1
    NEXT
    PRINT CHR$(13);                                      'end line with carriage return
    IF findFlag THEN startSearch = 1 : GOSUB Find
    count = count - lastBreak
```

Figure B-4. Counting words and characters *(continued)*

more...

```
    newLine$ = RIGHT$(newLine$, count)          'leave unprinted characters in newLine$
    RETURN

Find:
    printedLine$ = UCASE$(printedLine$)         'searchFor$ and printedLine$ uppercase
    found = INSTR(startSearch, printedLine$, searchFor$)
    IF found <> 0 THEN occur = occur + 1 ELSE RETURN
    startSearch = found + LEN(searchFor$)       'restart search after current word
    IF startSearch <= LEN(printedLine$) - LEN(searchFor$) THEN GOTO Find
    RETURN
```

Figure B-4. Counting words and characters *(continued)*

This program also offers the opportunity to count the number of occurrences of a specified word, in addition to generally counting all words. (Of course, if your program can recognize a specific word to count it, you could also have it delete the word or change it.)

The first section of the program displays a message telling the user what the program does and providing an edit field to enter a word to count. If a word to count is entered, the *Waiting* section retrieves the word, stores it as the variable *keyWord$* (and stores an all-caps version as *searchFor$*), and sets the variable *findFlag* equal to *true*. This flag is checked at several points in the program: after each paragraph is printed, to see if the paragraph should be scanned for the specified word, and at the end of the program, to see if the number of occurrences of the word should be displayed.

Notice that in the *GetFile* routine, I used the optional *prompt-string* parameter with the FILES$ function to limit the files listed to those of type TEXT, WORD, or WDBN. Common file types and the programs that create them are:

File type	Program
TEXT	BASIC
WORD	MacWrite
WDBN	Microsoft Word, formatted

In the *Start* section, which sifts through the characters in each paragraph, the program keeps track of the position of the last occurrence of a space (*ltr*=32) or hyphen (*ltr*=45). It stores this as *lastBreak*, and later uses this information to decide

where to break the line when displaying it on the screen. The maximum line width is specified in *Begin* (the variable *wide* is originally set to 75), but the program will try to break the line between words, rather than exactly at the line length specified.

After each paragraph is printed, *findFlag* is checked. If it is *true*, the offset is set equal to 1 and the program is diverted to the *Find* subroutine, which uses the INSTR function to check each paragraph for the supplied word. The syntax for this is:

INSTR([*I*,]*X$*, *Y$*)

The optional *I* is an offset indicating how far into string *X$* the search for *Y$* should start. The function returns a number equal to the position, in characters from the beginning of *X$*, that *Y$* starts, or a zero if there is no occurrence of *Y$*.

If the paragraph does not contain the word, the program then returns to the *Prnt* subroutine. If the word is found, the counter (*occur*) is incremented, and the offset is increased by the starting position of the word plus the length of the word. The *Find* subroutine is repeated until either the word is not found, or the offset comes within the length of the word from the end of the paragraph.

A program to create DATA statements

While writing the shell-game program for Chapter 14, I needed a way to convert a MacPaint picture to DATA statements, so you wouldn't have to draw your own picture of a walnut shell. I wrote the short utility shown in Figure B-5 to do the job. This program brings a picture in from the Clipboard, displays it on the screen within a specified area, uses the GET statement to store it in an integer array, and then breaks the array down to a series of DATA statements.

There are several variables in this program that you will have to adjust for the size of the picture you are using. The size of the array is determined by the area captured with the GET statement, which is in turn determined by the area the picture is scaled to with the PICTURE statement. The following formula, discussed in Chapter 7, is used to determine the size array required:

$$4+(((y2-y1)+1)*2*\text{INT}(((x2-x1)+16)/16))$$

The x and y variables are the boundaries used in the PICTURE and GET statements.

```
'* Program to create DATA statements
'*

   DEFINT a-z
   DIM pict(125)

'*
'* Tell user what to do and get response.
'*
Start:
   WINDOW 1, , (0, 30) - (512, 342), 3                    'background window
   WINDOW 2, , (100, 50) - (350, 170), 2                   'instruction window
   PRINT "Copy a picture from the Scrapbook"
   PRINT "to the clipboard and then click OK."
   PRINT "Click Quit to return to BASIC."
   BUTTON 1, 1, "Quit", (20, 85) - (80, 105)
   BUTTON 2, 1, "OK", (175, 85) - (235, 105)

   '*
   '* Get response.
   '*
   WHILE DIALOG(0) <> 1                                    ' wait for button-click
   WEND
   butSel = DIALOG(1)
   IF butSel = 1 THEN END
   WINDOW CLOSE 2

   '*
   '* Open clipboard in preparation for bringing in picture
   '* previously placed there.
   '*
   OPEN "clip:picture" FOR INPUT AS #1
   image$ = INPUT$ (LOF(1), 1)                                'bring image in
   CLOSE 1
   IF image$ = "" THEN GOTO Start                            'try again if blank
   PICTURE (50, 50) - (100, 80), image$                      'display the image
   GET (50, 50) - (100, 80), pict                            'store image in array

   '*
   '* Open output file in which to store image.
   '*
   filename$ = FILES$(0, "Store image in file:")
   IF filename$ = "" THEN SYSTEM
```

Figure B-5. Creating DATA statements

more...

```
   OPEN filename$ FOR OUTPUT AS #2
   count = 0                                          'initialize counter
'*
'* Convert array to series of DATA statements.
'*
PrintDataStatement:
   PRINT #2, "DATA ";                                 'DATA statement on each line
   FOR item = 1 TO 7                                  'first 7 items on line
      PRINT #2, pict(count); ", ";
      count = count + 1                               'increment counter
      IF count = 125 THEN CLOSE 2 : END               'check for end
   NEXT item
   PRINT #2, pict(count)                              'last item on line (no comma)
   count = count + 1                                  'increment counter
   GOTO PrintDataStatement                            'do next line
```

Figure B-5. Creating DATA statements *(continued)*

The value returned by this formula is the number of bytes required to store the picture. Since there are two bytes per array element in an integer array, you can divide the value by 2, and enter the result as the variable *limit* at the beginning of the program. This variable is used in both the DIM statement and in the test at the end of the program to see if all array elements have been read.

I wrote this utility to convert one picture. If you are going to convert a bunch of pictures of different sizes, you could modify this program to include the ability to adjust the picture size with the mouse, and then automatically compute the array size and dimension the array.

To use the DATA statements produced by this program in another program, you can either open the sequential file they are stored in and read the data, or use the MERGE command to incorporate the data in the new program. To demonstrate this second technique, type the short program shown in Figure B-6, and then make the Command window active and enter the command *MERGE,* followed by the name of the file in which you stored the DATA statements (enclose the file name in quotation marks). When you press Return, BASIC brings the DATA statements in from the disk file and attaches them to the end of your program. If you now run the program, the picture described by the statements is displayed.

```
ShowNewPict:
  DIM newPict%(125)
  FOR item = 0 TO 125
    READ newPict%(item)
  NEXT
  PUT (100, 100), newPict(0)
```

Figure B-6. Merging DATA statements

A program to compute the dimensions of a picture array

Now that you have suffered through the formula used to compute the dimensions of a picture array, here's a utility that will do the math for you. The essence of this program—the part that actually computes the number of elements in the array—is only a few lines long; the remainder of the program is used to set up and manage the edit fields (shown in Figure B-7) that solicit the boundaries.

The full program is shown in Figure B-8 on the following pages. There should be no unfamiliar commands in this listing, though you may want to note the fact that the VAL function is used in the *RetrieveValues* section to convert the string contents of each edit field to a numeric value.

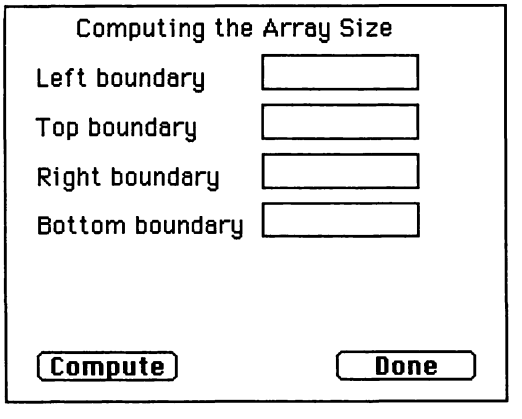


Figure B-7. The array-dimension dialog box

```
'* Program to compute array size needed to hold picture of given dimension
'*

   '*
   '* Open display window.
   '*
   WINDOW 4, , (100, 50) - (350, 250), -3

   '*
   '* Print labels and edit fields.
   '*
   MOVETO 35, 15
   PRINT "Computing the Array Size"
   MOVETO 15, 40
   PRINT "Left boundary"
   EDIT FIELD 1, "", (130, 25) - (205, 40)
   MOVETO 15, 65
   PRINT "Top boundary"
   EDIT FIELD 2, "", (130, 50) - (205, 65)
   MOVETO 15, 90
   PRINT "Right boundary"
   EDIT FIELD 3, "", (130, 75) - (205, 90)
   MOVETO 15, 115
   PRINT "Bottom boundary"
   EDIT FIELD 4, "", (130, 100) - (205, 115)
   ef = 1: EDIT FIELD ef                          'make top edit field active

   '*
   '* Create buttons.
   '*
   BUTTON 1, 1, "Compute", (15, 175) - (85, 190)
   BUTTON 2, 1, "Done", (165, 175) - (235, 190)

'*
'* Wait for something to happen
'*
MainLoop:
   WHILE 1 = 1
      event = DIALOG(0)
      IF event = 1 THEN GOSUB ButtonPressed
      IF event = 2 THEN ef = DIALOG(2)
```

Figure B-8. Computing array dimensions

more...

```
    IF event = 6 THEN GOSUB RetrieveValues
    IF event = 7 THEN ef = ef MOD 4 + 1: EDIT FIELD ef
  WEND

'*
'* Button was pressed.
'*
ButtonPressed:
  IF DIALOG(1) = 2 THEN END

'*
'* Retrieve entries.
'*
RetrieveValues:
  left = VAL(EDIT$(1))
  top = VAL(EDIT$(2))
  right = VAL(EDIT$(3))
  bottom = VAL(EDIT$(4))
  high = bottom - top
  wide = right - left

  '*
  '* Compute size.
  '*
  size = 4 + ((high + 1) * 2 * INT((wide + 16) / 16))
  arrayElements = size / 2

  '*
  '* Print answer.
  '*
  MOVETO 0, 135
  PRINT "An array to hold the pixels within the"
  PRINT "rectangle ("; left; ", "; top; ") - ("; right; ", "; bottom; ")"
  PRINT "requires "; arrayElements; "elements.
  RETURN
```

Figure B-8. Computing array dimensions *(continued)*

This program was written more to convey the idea that you can modify code and apply it to your own needs than as a finished product (it is actually a modification of the program in Section II that transfers an image into BASIC). If you are creating a lot of picture arrays, you may want to rewrite this program as a subprogram that uses the

beginning and ending points of a mouse drag to define the picture boundaries. The number of elements required by the array could be passed to the main program, which would erase the previous array and dimension a new one.

A program to strip comments

The next program removes REM statements and comments. There are two reasons for doing this: size and speed. A heavily commented program can easily be twice as large as the same program without the comments, and runs substantially slower. Size is a factor when disk space is at a premium or when the amount of available RAM in your Macintosh limits how large a program can be loaded. Speed is not always important, but when doing number crunching or animation, it can be significant.

Before having a look at the stripper, let's do a quick experiment to determine the effect of comments on a program's speed. Enter and run the short program shown in Figure B-9. Record the time span displayed. Now add a REM statement between the FOR and NEXT statements and repeat the test. Add an apostrophe and comment to the line *FOR count = 1 TO 25000* and again repeat the test. Remove the REM statement and repeat the test again. You should now have a list of run-times that will give you an idea of the effect of comments and REM statements.

The stripper program is listed in Figure B-10. The comments should pretty well explain the purpose of each section. The program opens the commented file for input and a new file for output, where it will place the stripped lines of the input file. It then

```
'*Speed test
'*

   BEEP
   time1! = TIMER
   FOR count = 1 TO 25000
   NEXT count
   time2! = TIMER
   BEEP
   PRINT time2! - time1!; " seconds"
```

Figure B-9. A speed test

```
'* Program to strip comments
'*

   remark$ = "REM"

'*
'* Open input and output files.
'*
Start:
   inputFile$ = FILES$(1, "TEXT")
   IF inputFile$ = "" THEN END
   outputFile$ = FILES$(0, "Save as")
   IF outputFile$ = "" THEN END
   OPEN inputFile$ FOR INPUT AS #1
   OPEN outputFile$ FOR OUTPUT AS #2

   '*
   '* Process input file.
   '*
   WHILE NOT EOF(1)
      LINE INPUT # 1, lin$
      length = LEN(lin$)
      IF length = 0 THEN GOTO Skip

      '*
      '* Remove leading spaces.
      '*
      FOR char = 1 TO length
         char$ = MID$(lin$, char, 1)
         IF char$ = CHR$(32) THEN GOTO SkipChar
         lin$ = MID$(lin$, char)                    'new line, without leading spaces
         char = length + 1
SkipChar:
      NEXT

      '*
      '* Find first apostrophe in line.
      '*
      comment = INSTR(lin$, CHR$(39))
      IF comment <> 0 THEN lin$ = LEFT$(lin$, comment-1)
      length = LEN(lin$)
      IF length = 0 THEN GOTO Skip
```

more...

Figure B-10. The stripper

```
    '*
    '* Find  first REM in line.
    '*
    comment = INSTR(lin$, remark$)
    IF comment <> 0 THEN lin$ = LEFT$(lin$, comment-1)
    length = LEN(lin$)
    IF length = 0 THEN GOTO Skip

    '*
    '* Get rid of spaces to the right of line.
    '*
    FOR char = length TO 1 STEP -1
       char$ = MID$(lin$, char, 1)
       IF char$ = CHR$(32) THEN GOTO Skip2
       lin$ = LEFT$(lin$, char)
       char = 0
Skip2:
    NEXT

    '*
    '* Drop final colon.
    '*
    IF RIGHT$(lin$,1) = ":" THEN lin$ = LEFT$(lin$, LEN(lin$) - 1)
    PRINT #2, lin$                                    'put line in output file
    PRINT lin$, LEN(lin$)                                   'show results
Skip:
  WEND
  CLOSE
  WHILE MOUSE(0) = 0 : WEND                              'time to read results
  GOTO Start
```

Figure B-10. The stripper *(continued)*

reads the lines of the input file, one-by-one, discards the line if it is blank, and removes any leading spaces. The resulting line is searched for an apostrophe, and if one is found, a new line is formed using all of the old line up to the apostrophe. (Note: Everything after an apostrophe within a PRINT statement will also be stripped. I will leave the solution to this problem as an exercise for you to solve.) The line is next searched for a REM statement, which, if found, is discarded. Any spaces remaining to the right of the line when a comment or REM statement was removed are now discarded, along with the final colon that would have separated a trailing REM statement

from the rest of the line. The stripped line is stored in the output file and displayed on the screen, to prove that the program works.

A program to create a SYLK file

Different application programs, for the Macintosh as well as other computers, store their output files in different formats. Each, naturally, reads its own format, and some have the ability to convert to and from other formats. If you know the format expected by a program, you can have BASIC store its results in that format, so that it can be read directly into the other program. This could be useful, for example, when you have an established BASIC program to accumulate and evaluate numeric information that you would like to chart, using the Microsoft Chart program. Chart will allow you to paste data from the Clipboard and format that into a chart but it would be more efficient to simply read the data file directly.

Chart and most other Microsoft applications can directly read two types of files: their own unique format and SYLK (Symbolic Link) format. SYLK is sort of a general-purpose format that has room for the types of information desired by various applications. Each application can open a SYLK file and read the portions it needs. There is a fairly complete, though not very understandable, explanation of SYLK format in the appendices of the Chart and Multiplan manuals.

The program in Figure B-11 demonstrates how a SYLK file could be created to feed data points to Chart to be plotted. It includes sample data points for the purpose of demonstration: If you were actually using this program, you would probably convert it to a subprogram and pass the data points to it from the main program.

```
'* Program to create a Microsoft SYLK file
'*

' This program could be called from a program that performs calculations
' and produces a set of x and y variables, or it could be tacked onto the
' end of such a program. The variable NumDatPts, and the data points
' themselves should be defined in the main program, and x() and y()
' should be dimensioned at the beginning of the main program. If you
```

more...

Figure B-11. Creating a SYLK file

```
' are creating a Text series, the variable x should be defined as a string
' variable. If multiple files are to be created during one session, make
' sure that the arrays are ERASEd prior to re-dimensioning.

  DEFINT a - w

  '*
  '* Following values are included to demonstrate program.
  '*
  numDatPts = 4                                    'number of datapoints passed
  DIM x$(numDatPts - 1), y$(numDatPts - 1)
  DIM x(numDatPts - 1), y(numDatPts - 1)           'normally done in main program
  x(0) = 9                                         'x() and y() passed from main program
  x(1) = 3
  x(2) = 5
  x(3) = 8
  y(0) = 11
  y(1) = 22
  y(2) = 43
  y(3) = 14
  x$(0) = "one"
  x$(1) = "five"
  x$(2) = "seven"
  x$(3) = "ten"

  '*
  '* Assign default names and provide opportunity to change them.
  '*
  WINDOW 4, , (100, 50) - (400, 200), -2           'create dialog box
  MOVETO 10, 20
  PRINT "Series name:"
  MOVETO 15, 40
  PRINT "category axis:"
  MOVETO 15, 60
  PRINT "value axis:"
  MOVETO 10, 90
  PRINT "Type:"
  BUTTON 1, 2, "Sequence", (15, 100) - (100, 115), 3    'selected by default
  currentButton = 1
  type$ = "S"
  BUTTON 2, 1, "Date", (150, 100) - (250, 115), 3
  BUTTON 3, 1, "Text", (15, 120) - (100, 135), 3
  BUTTON 4, 1, "Number", (150, 120) - (250, 135), 3
```

Figure B-11. Creating a SYLK file *(continued)*

more...

```
   BUTTON 5, 1, "OK", (260, 100) - (290, 135), 1
   vaName$ = "Y"                                        'default value axis name
   EDIT FIELD 3, vaName$, (120, 45) - (290, 60)
   caName$ = "X"                                        'default category axis name
   EDIT FIELD 2, caName$, (120, 25) - (290, 40)
   sName$ = TIME$                                       'default series name
   EDIT FIELD 1, sName$, (120, 5) - (290, 20)
   ef = 1

'*
'* Watch for action in dialog box.
'*
Loop:
   event = DIALOG(0)
   IF event = 1 THEN GOSUB But                          'Type button was clicked
   IF event = 2 THEN ef = DIALOG(2)                     'different edit box was clicked
   IF event = 6 THEN GOTO Done                          'Return key pressed
   IF event = 7 THEN ef = ef MOD 3 + 1 : EDIT FIELD ef  'Tab key pressed
   GOTO Loop

But:
   buttonPushed = DIALOG(1)                             'which button was pressed
   IF buttonPushed = 1 THEN type$ = "S"                 'Sequence button
   IF buttonPushed = 2 THEN type$ = "D"                 'Date button
   IF buttonPushed = 3 THEN type$ = "A"                 'Text button
   IF buttonPushed = 4 THEN type$ = "N"                 'Number button
   IF buttonPushed = 5 THEN GOTO Done
   BUTTON currentButton, 1                              'deselect previously selected button
   BUTTON buttonPushed, 2                               'select new button
   currentButton = buttonPushed                         'store currently selected button
   RETURN

Done:                                                   'get new names
   sName$ = EDIT$(1)

   '*
   '* Have to get rid of  colons (if time retained as name), as this is
   '* file name, and Mac recognizes colon as volume identifier.
   '*
   tempName$ = ""
```

Figure B-11. Creating a SYLK file *(continued)*

more...

```
    FOR count = 1 TO LEN(sName$)
      char$ = MID$(sName$, count, 1)                    'check each character
      IF ASC(char$) = 58 THEN char$ = CHR$(46)          'change : to -
      tempName$ = tempName$ + char$                     're-create name
    NEXT
    sName$ = tempName$
    caName$ = EDIT$(2)
    vaName$ = EDIT$(3)
    WINDOW CLOSE 4

    '*
    '* Produce SYLK file.
    '* See Appendix of Chart documentation for explanation of SYLK files.
    '*
    OPEN sName$ FOR OUTPUT AS #1
    PRINT #1, "ID;PMC"
    PRINT #1, "C;X1;Y1;K"; CHR$(34); "MC;;T"; type$; ";;N"; sName$; ";;S1;;P1"; CHR$(34)
    PRINT #1, "C;Y2;K"; CHR$(34); sName$; "."; caName$; CHR$(34)
    PRINT #1, "C;X2;K"; CHR$(34); sName$; "." vaName$; CHR$(34)

xVal:
    IF type$ = "S" OR type$ = "D" THEN GOTO yVal        'x values computed by Chart
    IF type$ = "A" THEN GOTO aVal                       'have to insert quotation marks
    lin$ = "C;X1;Y4;K" + STR$(x(0))
    GOSUB Strip
    FOR dp = 1 TO numDatPts - 1                         'print rest of x-axis data points
      lin$ = "C;Y" + STR$(dp + 4) + ";K" + STR$(x(dp))
      GOSUB Strip
    NEXT
    GOTO yVal

aVal:
    lin$ = "C;X1;Y4;K" + CHR$(34) + x$(0) + CHR$(34)    'insert quotes
    GOSUB Strip
    FOR dp = 1 TO numDatPts - 1
      lin$ = "C;Y" + STR$(dp + 4) + ";K" + CHR$(34) + x$(dp) + CHR$(34)
      GOSUB Strip
    NEXT

yVal:
    FOR count = 0 TO numDatPts - 1
      y$(count) = STR$(y(count))
    NEXT
```

Figure B-11. Creating a SYLK file *(continued)*

more...

```
    lin$ = "C;X2;Y4;K" + y$(0)                    'specify y-axis and print first data point
    GOSUB Strip
    FOR dp = 1 TO numDatPts - 1                    'print rest of y-axis data points
      lin$ = "C;Y" + STR$(dp + 4) + ";K" + y$(dp)
      GOSUB Strip
      NEXT
    PRINT #1, "E"                                  'indicates end of SYLK file
    CLOSE 1
    END

Strip:
      FOR count = 1 TO LEN(lin$)
        char$ = MID$(lin$, count, 1)
        IF ASC(char$) = 32 THEN GOTO Skip
        nLin$ = nLin$ + char$
Skip:
      NEXT
      PRINT #1, nLin$
      nLin$ = ""
      RETURN
```

Figure B-11. Creating a SYLK file *(continued)*

When you run the program, the dialog box shown in Figure B-12 solicits information about the chart. This information, along with the data points, is used to create a simple column chart. With a little more study of the SYLK format, you could add buttons to select the style of chart: bar, column, line, and so on.

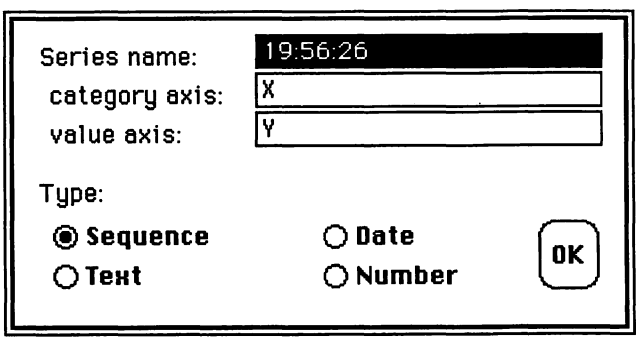


Figure B-12. The information dialog box

The SYLK format is very flexible. The information stored in this file by BASIC can be retrieved by Chart, Multiplan, or File. If you do a lot of number crunching in BASIC and want to pass the results to one of these programs, you should spend some time picking apart the SYLK format, and then modify this program to suit your needs.

A sort program

The sort program that's shown in Figure B-13 is another demonstration program, rather than a finished product. This program allows you to enter a list of words, and then sorts them into alphabetical order. If you were to incorporate this routine into one of your programs, you would probably have it read words from a file, rather than typing them in one at a time. This program is reasonably efficient, but since it does the entire sort in memory, rather than on disk, the number of words that can be sorted is limited by the amount of available memory. (The variable *max*, defined in the *Start* section, limits the number of entries the program will accept from the keyboard; the number that can actually be sorted is limited by memory.)

```
'* Program to sort an array of string variables in memory
'* using Decreasing Increment Sort method
'*

Start:
   DEFINT a - z
   max = 100                                   'maximum number of words to sort
   DIM word$(max)
   WINDOW 1, , (10, 30) - (200, 340), 3                          'output window

   '*
   '* Create data entry window with control buttons.
   '*
   WINDOW 2, , (201, 30) - (400, 100), 3
   PRINT "Type word and press Return"
   EDIT FIELD 1, "enter word", (10, 20) - (180, 35)
   BUTTON 1, 1, "Sort", (10, 45) - (55, 65)
   BUTTON 2, 1, "Again", (65, 45) - (110, 65)
   BUTTON 3, 1, "Quit", (120, 45) - (180, 65)
```

Figure B-13. A sort program

more...

```
'*
'* Wait for something to happen in window.
'*
Loop:
   event = DIALOG(0)
   IF event = 6 THEN GOSUB GetWord                    'Return pressed
   IF event = 1 THEN GOSUB GetButton                  'button pressed
   GOTO Loop

'*
'* Retrieve word from edit field.
'*
GetWord:
   temp$ = EDIT$(1)
   IF temp$ = "" THEN GOTO Invalid
   count = count + 1
   IF count = 101 THEN GOTO Invalid
   word$(count) = temp$
   temp$ = ""
   EDIT FIELD 1, "enter word", (10, 10) - (180, 25)
   RETURN
Invalid:
   BEEP
   RETURN

'*
'* Respond to button pushed.
'*
GetButton:
   butPushed = DIALOG(1)
   IF butPushed = 3 THEN END                          'Quit
   IF butPushed = 2 THEN CLEAR: GOTO Start            'Again
   GOSUB Sort
   GOSUB Look
   RETURN

'*
'* Sort routine.
'*
Sort:
   WINDOW 1
   PRINT
```

Figure B-13. A sort program *(continued)*

more...

```
   PRINT "SORTING..."
   offset = count
Sort1:
   offset = offset \ 2
   IF offset = 0 THEN RETURN
   midPoint = count  - offset
   sortCount = 1
Sort2:
   low = sortCount
Sort3:
   high = low + offset
   IF word$(low) <= word$(high) THEN GOTO Sort4

   '*
   '* Swap words
   '*
   temp$ = word$(low)
   word$(low) = word$(high)
   word$(high) = temp$
   low = low  - offset
   IF low >= 1 THEN GOTO Sort3
Sort4:
   sortCount = sortCount + 1
   IF sortCount > midPoint THEN GOTO Sort1
   GOTO Sort2

'*
'* Display sorted list.
'*
Look:
   CLS
   PRINT
   PRINT "SORTED LIST:"
   PRINT
   FOR temp = 1 TO count
      PRINT "["; temp; "]" ; word$(temp)
   NEXT temp
   WINDOW 2
   RETURN
```

Figure B-13. A sort program *(continued)*

A program to convert a Word file to an unformatted text file

The next utility should require almost no typing on your part, as it is a modification of a previous program (the one that counts words and characters). This utility is useful if you have files formatted by Word stored on a disk, and you would like to telecommunicate them, which usually requires stripping all formatting.

There are no new commands in this program, which is shown in Figure B-14, but there is a little bit of magic in the *GetFile* routine that I am going to leave for you to figure out for yourself.

```
'* Program to convert Word file to unformatted text file
'*

   DEFINT a - z
   WINDOW 1, , (2, 20) - (510, 340), 3                    'the output window

'*
'* Initialize variables.
'*
Begin:
   numChar = 0                                    'number of characters counted
   wide = 75                                      'width of output in characters
   lastBreak = 1                                  'place to break line
   count = 0                                      'number of characters in current line
   newLine$ = ""                                  'blank line
   true = -1                                      'logical true
   false = 0                                      'logical false

'*
'* Get name of file to be searched.
'*
GetFile:
   doc$ = FILES$(1, "WDBN")
   IF doc$ = "" THEN  END
   OPEN doc$ FOR INPUT AS #1
   head$ = INPUT$(128, 1)
   nBytes = 256 * ASC(MID$(head$, 17, 1)) + ASC(MID$(head$, 18, 1)) - 128
```

more...

Figure B-14. Converting a Word file to a text file

```
'*
'* Open output file.
'*
OutputFile:
   newFile$ = FILES$(0, "File in which to save text")
   IF newFile$ = "" THEN END
   OPEN newFile$ FOR OUTPUT AS #2

'*
'* Start display and search.
'*
Start:
   WHILE NOT endFlag
      LINE INPUT #1, para$
      numChar = numChar + 1                                    'count carriage return

      '*
      '* Check each letter in paragraph.
      '*
      FOR position = 1 TO LEN(para$)
         ltr =  ASC(MID$(para$, position, 1))
         count = count + 1

         '*
         '* Check for space, hyphen, or newline.
         '*
         IF ltr = 32 OR ltr = 45 OR ltr = 11 THEN lastBreak = count
         IF ltr = 11 THEN ltr = 13
         newLine$ = newLine$ + CHR$(ltr)                       'form new line
         IF count = wide THEN GOSUB Prnt                       'check length
Skip:
      NEXT
      GOSUB Prnt
   WEND
   CLEAR : GOTO Begin
```

Figure B-14. Converting a Word file to a text file *(continued)*

more...

```
'*
'* Display document.
'*
Prnt:

   '*
   '* Determine how much to print.
   '*
   IF count < wide THEN lastBreak = count
   printedLine$ = LEFT$(newLine$, lastBreak)
   numChar = numChar + LEN(printedLine$)                    'accumulate character count
   IF numChar > = nBytes THEN endFlag = true

   '*
   '*Print line.
   '*
   PRINT printedLine$
   PRINT #2, printedLine$
   count = count - lastBreak
   newLine$ = RIGHT$(newLine$, count)          'leave unprinted characters in newLine$
   RETURN
```

Figure B-14. Converting a Word file to a text file *(continued)*

The text in a Word file is preceded by a 128-byte header that contains information about the file, and is followed by a section of formatting information. Included in the header are two bytes, the 17th and 18th, that indicate the size of the file less the formatting information at the end. The number of characters in the file is computed by multiplying byte 17 by 256 and adding byte 18 to the product. The result includes the 128-byte header, so subtracting 128 from the total gives the exact number of characters in the text portion of the file. By keeping track of the number of characters that are transferred from the Word file to the unformatted file, we will be able to stop the transfer after all the text has been copied to the new file, and avoid transferring any of the formatting information.

If you are using the interpretive version of BASIC, it is just as much of a bother to load BASIC and run this program as it is to load Word and save the file without formatting, but with Microsoft's BASIC compiler, this program should be more convenient.

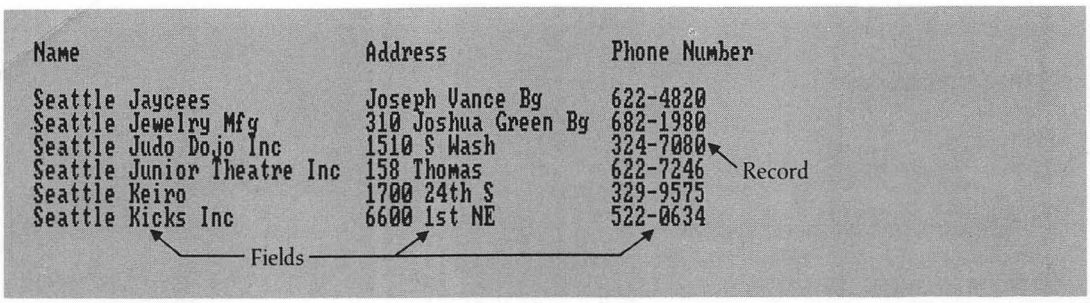


Figure B-15. A section of the telephone book

A program to create a random-access file

The next program deals with random-access files, BASIC's alternative to sequential files. The primary difference between the two file types is pointed out by their names. As you know from working with sequential files, items of information are stored and retrieved one after the other, from the beginning of the file to the end. If you want to read the 20th, 43rd, and 72nd records in a sequential file, you have to read the first 72 records. In a random-access file, on the other hand, you can simply request those three files, without sifting through the rest. Random-access files are slightly (but only slightly) more difficult to work with than sequential, but are substantially faster when you want to retrieve or change specific records in a file.

A more subtle difference is that in a sequential file the records are not necessarily related to each other in any manner, whereas in a random-access file each record contains essentially the same type of information. A familiar object that has many of the characteristics of a random-access file is your telephone book. Figure B-15 shows a chunk of my phone book as it might look if entered into a random-access file. Each record consists of three fields: Name, Address, and Phone Number. Each field is allocated a specific amount of space; if the entry is shorter than the allocated space, it is padded with spaces to make it fit exactly.

Although this is not a utility created to do a specific job, it could easily be developed into a general-purpose data entry routine that could be used by any program requiring the user to repeatedly provide the same kind of information. The entire program is listed in Figure B-16. I'm only going to discuss the parts that are unique to random-access files.

```
'* Random access file program
'*

   '*
   '* Open file and define field sizes.
   '*
   OPEN "nameList" AS #1 LEN = 40
   FIELD #1, 19 AS firstName$, 19 AS lastName$, 2 AS age$
   record = 1
   up = 1
   down = -1
   f$ = "First Name"
   l$ = "Last Name"
   a$ = "Age"

   '*
   '* Create data entry window.
   '*
   WINDOW 1, , (75, 100) - (450, 290), 3
   MOVETO 10, 25
   TEXTFACE 1
   PRINT "Entering data into a random access file."
   TEXTFACE 0
   MOVETO 160, 60
   PRINT "Fill in the fields and  click OK "
   MOVETO 160, 90
   PRINT "or press Return to enter record."
   MOVETO 160, 120
   PRINT "Click Quit to return to BASIC"
   MOVETO 165, 160
   PRINT "Review"
   LINE (95, 145) - (280, 170), , b
   BUTTON 1, 1, "OK", (10, 150) - (70, 165)
   BUTTON 2, 1, "QUIT", (305, 150) - (365, 165)
   BUTTON 3, 1, "Up", (100, 150) - (150, 165)
   BUTTON 4, 1, "Down", (225, 150) - (275, 165)

'*
'* Create edit fields.
'*
CreateEditFields:
   EDIT FIELD 1, f$, (10, 45) - (150, 60)
```

Figure B-16. Creating a random-access file

more...

```
    EDIT FIELD 2, l$, (10, 75) - (150, 90)
    EDIT FIELD 3, a$, (10, 105) - (50, 120)
    ef = 1 : EDIT FIELD 1

'*
'* Wait for event.
'*
Loop:
    event = DIALOG(0)
    IF event = 1 THEN GOSUB GetButton                    'button clicked
    IF event = 2 THEN ef = DIALOG(2)                     'edit field clicked
    IF event = 6 THEN GOSUB EnterRecord                  'Return pressed
    IF event = 7 THEN ef = ef MOD 3 + 1 : EDIT FIELD ef  'Tab pressed
    GOTO Loop

'*
'* Take action on button click.
'*
GetButton:
    butPushed = DIALOG(1)
    IF butPushed = 2 THEN END
    IF butPushed = 3 THEN direction = -1 : GOSUB Review  'move up
    IF butPushed = 4 THEN direction = 1 : GOSUB Review   'move down
    GOSUB EnterRecord
    RETURN

'*
'* Retrieve entry and store in file.
'*
EnterRecord:
    first$ = EDIT$(1)
    last$ = EDIT$(2)
    years$ = EDIT$(3)

    '*
    '* Move data to buffer.
    '*
    LSET lastName$ = last$
    LSET firstName$ = first$
    LSET age$ = years$
```

Figure B-16. Creating a random-access file *(continued)*

more...

```
    '*
    '* Put in record.
    '*
    PUT #1, record
    IF record = lastRec + 1 THEN lastRec = record : record = record + 1
    IF record < lastRec + 1 THEN RETURN
    f$ = "First Name"
    l$ = "Last Name"
    a$ = "Age"
    RETURN CreateEditFields

'*
'* Review records already entered.
'*
Review:
    record = record + direction
    IF record < 1 THEN BEEP : GOTO FileEnd
    IF record > lastRec THEN BEEP : GOTO FileEnd

    '*
    '* Retrieve record from file.
    '*
    GET #1, record
    f$ = firstName$
    l$ = lastName$
    a$ = age$
    RETURN CreateEditFields

'*
'* Review has reached one end of file.
'*
FileEnd:
    record = lastRec + 1
    f$ = "First Name"
    l$ = "Last Name"
    a$ = "Age"
    RETURN CreateEditFields
```

Figure B-16. Creating a random-access file *(continued)*

This program opens a random file, specifies the kind of information that will be stored in each record, creates a data entry window with three edit fields, and then settles into the *Loop* routine to await some kind of action. The available actions are to enter records into the file, review entries, and quit.

You open a random-access file with the same OPEN statement you would use to open a sequential file. Here are the two syntaxes of this statement again:

OPEN *mode*, [#[*filenumber*,*filespec* [,*file-buffer-size*]

OPEN *filespec* [FOR *mode*] AS [#]*filenumber* [LEN = *file-buffer-size*]

To specify a random-access file you use a *mode* of *R* in the first syntax, and omit the *mode* in the second. For both, you specify a *file-buffer-size* equal to the combined size of the fields listed in the *FIELD* statement that follows.

The amount of space assigned to an individual record in a random-access file can be apportioned between multiple fields: In this program there are three fields, holding *firstName$*, *lastName$*, and *age$*. The amount of space allocated to each field is specified by the FIELD statement, which has this syntax:

FIELD [#]*filenumber*, *fieldwidth* AS *string-variable*...

The FIELD statement in this program:

```
FIELD #1, 19 AS firstName$, 19 AS lastName$, 2 AS age$
```

allocates 19 bytes to *firstName$*, 19 bytes to *lastName$*, and 2 bytes to *age$*. This totals 40 bytes, which is the number specified as *file-buffer-size* in the OPEN statement.

There is no practical limit on the number of fields you can have in a record, but there is a limit on how many characters you can have on one line of a program, so multiple FIELD statements are allowed. If you use more than one FIELD statement for a file, the first field of each statement after the initial one must be a dummy field allocating space equal to the sum of all previous fields, as shown in Figure B-17.

```
FIELD #1, 19 AS firstName$, 19 AS lastName$, 20 AS address$
FIELD #1, 58 AS dummy$, 12 AS phone$, 2 AS age$, 5 AS race$
```

Figure B-17. Multiple FIELD statements

The sections of the program that create the data-entry window and edit fields, and wait for an event, should be familiar to you. The next section that has anything new is the *EnterRecord* subroutine (Figure B-18), which retrieves the information from the edit fields and places it in the file.

Retrieving the information, with the EDIT$ function, is routine, but the process of putting each variable in the file is new. Variables aren't placed directly into a random-access file: All the variables you want to place into a particular record are transferred to a buffer, and then the contents of the buffer are placed in the file. Individual variables can be moved into the buffer with either the LSET or the RSET statement. The difference between the two is that LSET left justifies the variable if it is shorter than the field length, and RSET right justifies it. Once all the variables are in the buffer, they are transferred to a specific record of the file with the PUT statement.

```
'*
'* Retrieve entry and store in file.
'*
EnterRecord:
   first$ = EDIT$(1)
   last$ = EDIT$(2)
   years$ = EDIT$(3)

   '*
   '* Move data to buffer.
   '*
   LSET lastName$ = last$
   LSET firstName$ = first$
   LSET age$ = years$

   '*
   '* Put in record.
   '*
   PUT #1, record
   IF record = lastRec + 1 THEN lastRec = record : record = record + 1
   IF record < lastRec + 1 THEN RETURN
   f$ = "First Name"
   l$ = "Last Name"
   a$ = "Age"
   RETURN CreateEditFields
```

Figure B-18. The *EnterRecord* subroutine

Notice in the *EnterRecord* subroutine that each variable name is changed as the variable is passed to the buffer: The variable names used in the FIELD statement can be used only when transferring variables to and from the file. At other places in the program the same variables must be referred to by another name.

Also notice that all the variables stored in the file are string variables; even *age$*, which could conceivably be needed in the form of a numeric variable at some other point in the program. This is another characteristic of random-access files: All variables are stored as string variables. To move a numeric variable into the buffer in preparation for storing it in a record, use the MKI$, MKS$, or MKD$ function to convert an integer, single-, or double-precision variable to a string variable. When retrieving the same variable from the file, the CVI, CVS, or CVD function is used to convert back to an integer, single-, or double-precision variable.

The only item of interest left in the program is in the *Review* subroutine, shown in Figure B-19, which retrieves a record and places the variables in the edit fields for the user to view or modify. This subroutine uses the GET statement to retrieve a specific record from the file. GET places the variables from all fields of a record into the buffer, from which they can be assigned to variables used in the rest of the program.

```
'*
'* Review records already entered.
'*
Review:
   record = record + direction
   IF record < 1 THEN BEEP : GOTO FileEnd
   IF record > lastRec THEN BEEP : GOTO FileEnd

   '*
   '* Retrieve record from file.
   '*
   GET #1, record
   f$ = firstName$
   l$ = lastName$
   a$ = age$
   RETURN CreateEditFields
```

Figure B-19. The *Review* subroutine

In this case they are assigned to *f$*, *l$*, and *a$*, which are the default text strings for the three edit fields.

That's about all that's new in this program. If you enter and run it, you may want to experiment by adding an edit field that displays the record number of the current record. You could then allow the user to specify the record to be viewed, rather than having to scroll through the file as if it were a sequential file. Note the ease with which you can modify existing files as they pass in review.

This appendix presented a variety of utility programs. Most were originally written in a "quick and dirty" fashion to get a job done, and then cleaned up and commented for use in this book. These programs are included more as inspiration than with the idea of providing a ready-made solution to any specific problems. Excellent sources for additional examples of utilities are the Macintosh SIGs (special interest groups) available on information services such as CompuServe and THE SOURCE. So you might want to use the communication program developed in Chapter 12 to fatten your stockpile of BASIC programs.

Building the HBC-1

Appendix C

I realize that this is supposed to be a book about writing computer programs, not assembling electronic projects, but if you bear with me a while you will discover that the two are not all that dissimilar. As you study the integrated circuits that are used to build the Home-Brew Converter (HBC-1), you'll find that many of them are based on the same truth tables that describe the input-output relationship of the logic statements in BASIC: AND, OR, NOT, and so on. It is possible to write computer programs that simulate the action of electronic circuits, and to build electronic circuits that demonstrate the results of your programs.

The explanation that follows and the accompanying schematics should enable a relatively inexperienced electronic hobbyist to build the HBC-1. Additional information and control programs are available on the disk you can order from Microsoft.

The HBC-1 was designed by Gordon Mills, a linear field application engineer for Texas Instruments, to demonstrate the use of various TI integrated circuits. Although TI manufactures all the semiconductors, other than the diodes, needed to build this project, and provides data sheets, application notes, and other information about specific integrated circuits, the project as a whole is not a TI product, and TI probably would not appreciate requests for assistance in putting it together. If you have problems with the HBC-1, or if you find some really fantastic uses for it that you would like to share with others, address them to me, care of Microsoft Press.

Assembling the components

The three methods of assembling electronic projects differ primarily in the types of surface on which the components are mounted: poke-home board, wire-wrap, or printed circuit board. Whichever method you choose, if you have no previous

experience with it, I suggest you pick up a tutorial on the subject from one of the electronic hobby stores — Radio Shack or Heath, for example.

Printed circuit board

The third method listed is the easiest and most permanent — if a printed circuit board has already been created. Assembling the project is merely a matter of inserting the parts in the proper holes and soldering them in place. Unfortunately, there is no printed circuit board currently available, but I am working on this and, should one be developed, more information will be included on the program disk.

Poke-home board

Of the other two methods, the easiest (but not the most permanent) is the poke-home board. These are available in a variety of sizes and configurations, but in general they consist of a gridwork of holes, spaced at the proper interval for the insertion of integrated circuit (IC) pins. The holes — actually sockets — are interconnected in a fashion similar to that shown in Figure C-1. An IC is inserted straddling the center space, and the ground and supply voltages are connected to the side rails and jumpered over as needed. The row of sockets that each pin on the IC plugs into has four additional sockets available for jumpers to connect it to other devices.

Poke-home boards are available through mail-order catalogs and stores such as Radio Shack and Active Electronics. The layout illustrated in this book uses three boards, each 63 sockets long, but it is possible to rearrange the parts so as to use slightly less space.

Wire-wrap

The stages of construction for a wire-wrap model are the same as for the poke-home model, with a few extra parts and possibly one additional step. You will need wire-wrap sockets for the integrated circuits and mounting posts or IC-socket compatible component headers for the discrete components. Unless you are assembling the project on a board that already has substantial ground and V_{CC} grids, you will have

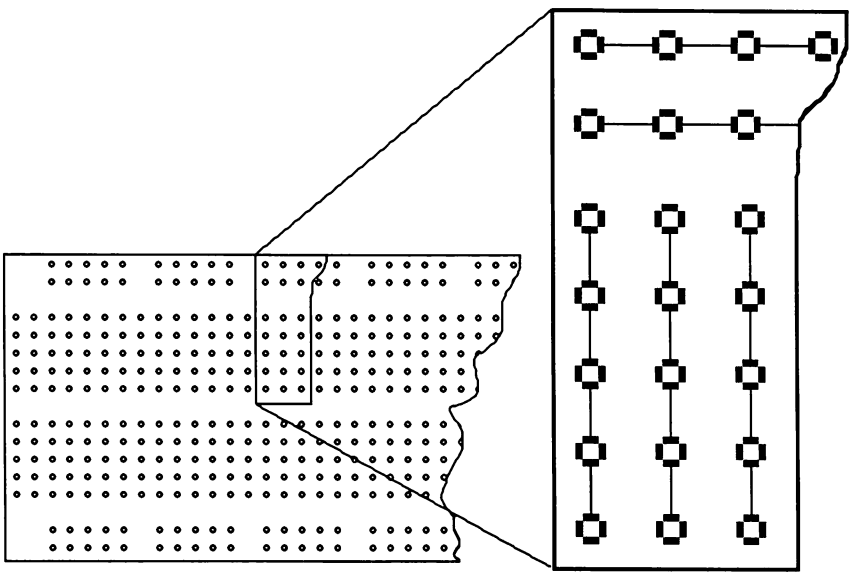


Figure C-1. Interconnection of poke-home sockets

to provide these. The ground and V_{CC} grids serve the same purpose as the side rails of the poke-home board: They distribute noise-free power evenly throughout the board (the lines of each grid must be connected together at three or four points to avoid small voltage differences along the grid). It is extremely important that the grids be constructed of heavy wire (14- or 16-gauge solid conductor) and the two grids be connected with five evenly distributed bypass capacitors of about 0.1μF each to minimize noise problems.

If you have not previously completed a wire-wrap project, a few hours spent reading the instructions and assembling some small project from a hobby store will probably save you more than that much time on this project.

With a project such as this, the placement of the parts can be critical. The placement shown in Figure C-2 on the following page, which can be followed for both the poke-home and the wire-wrap method, works and should not be modified unless you have a thorough understanding of the possible consequences.

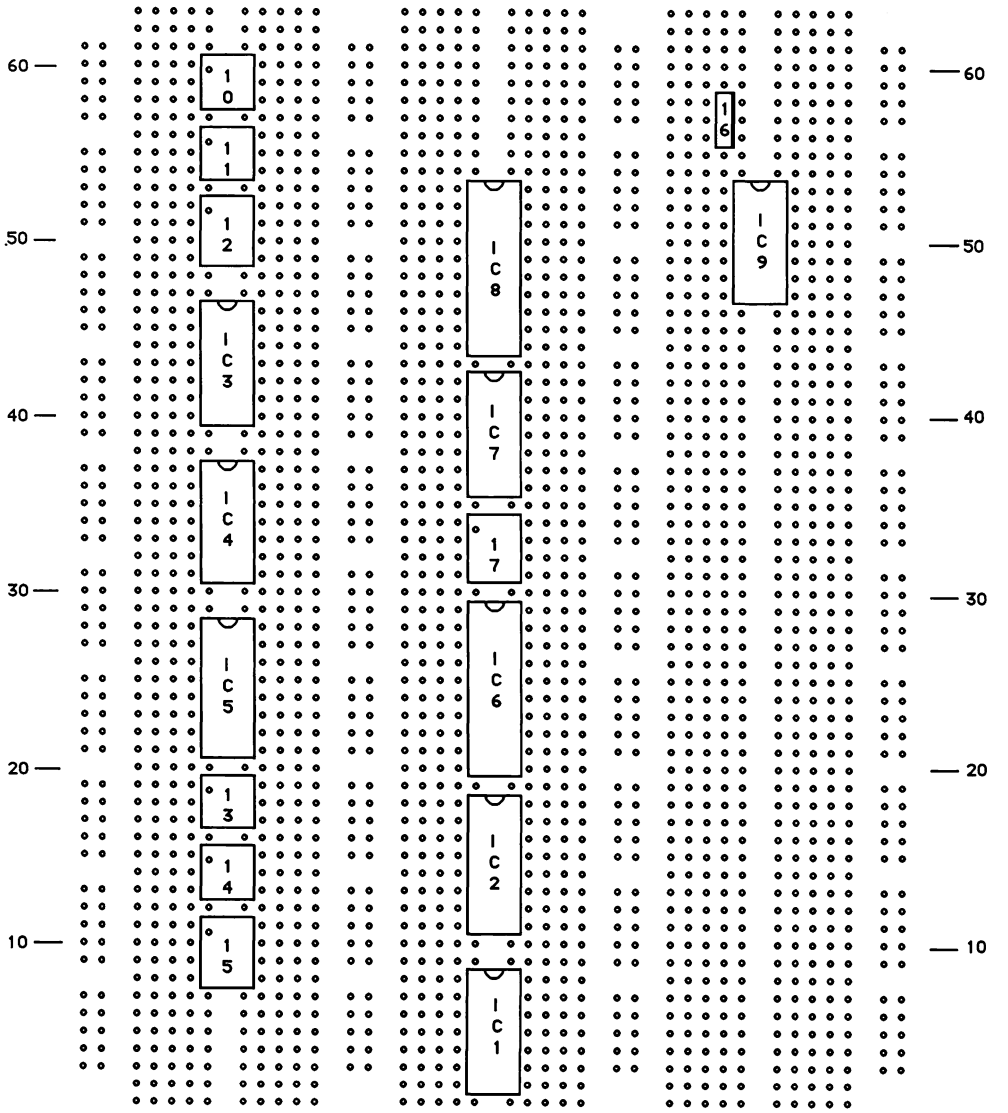


Figure C-2. Pictorial diagram of HBC-1

Parts and pin-outs

The description of each section of the HBC-1 will include a list of the parts used by that section: The following is a full list of the parts for all sections, with the parts for several of the optional input buffers tacked onto the end.

Quantity	Name	Part
1	SN74HCUO4N	IC1
1	SN74HC4020N	IC2
2	SN74HC74N	IC3, IC4
1	SN74HC163N	IC5
1	TLC540IN	IC6
1	SN74HC164N	IC7
1	SN74HC377	IC8
1	ULN2003A	IC9
4	TIL124	IC10, IC11, IC13, IC14
1	TLC272CP	IC12
1	TLC372N	IC15
1	TL780-05CKC	IC16
1	TL431CP	IC17
1	1MΩ resistor	R1
1	6.8KΩ resistor	R2
1	10KΩ resistor	R3
2	3.3KΩ resistors	R4, R12
1	10MΩ resistor	R5
1	2.2KΩ resistor	R6
2	10KΩ resistors	R7, R8
1	4.7KΩ resistor	R9
1	330Ω resistor	R10
1	33KΩ resistor	R11
1	100Ω 1% resistor	R13
1	4.12KΩ 1% resistor	R14
1	10KΩ 1% resistor	R15
1	22pF capacitor	C1
1	47pF capacitor	C2
6	0.1μF capacitors	C3 – C10
1	0.33μF capacitor	C11
1	4.7μF tantalum capacitor	C12
6	1N914 diodes	CR1 – CR6
1	1N4004 diode	CR7
1	2.4576MHz crystal	X1
Voltmeter buffer		
1	TLC27L2ACP	IC1
3	1MΩ, 1% resistors	R1, R2, R6
1	499KΩ, 1% resistor	R3
1	249KΩ, 1% resistor	R4
1	124KΩ, 1% resistor	R5

Quantity	Name	Part
1	332KΩ, 1% resistor	R6
1	143KΩ, 1% resistor	R7
1	.01μF capacitor	C1
Anemometer buffer		
1	TLC27L2ACP	IC1
1	30KΩ resistor	R1
1	100KΩ resistor	R2
2	20KΩ resistors	R3, R4
1	18KΩ, 1% resistor	R5
1	2KΩ, 1% resistor	R6
1	200KΩ, 1% resistor	R7
1	100KΩ, resistor	R8
1	22KΩ, resistor	R9
1	10KΩ potentiometer	VR1
2	0.01μF capacitors	C1, C2

NOTE: Unless specified otherwise, all resistors are 5% tolerance, and all capacitors are monolithic ceramic.

Figure C-3 is a schematic for the major portion of the HBC-1. The discussion of the HBC-1's individual sections in the second half of this appendix includes schematics for each section. If you run these through a copy machine, you should be able to tape them together to form a larger version of the schematic in Figure C-3, which you can use as a road map to keep you visually oriented as you assemble the project. However, you should use the list that follows to actually determine which connection to make next. This list gives the point-to-point wiring for each pin of every IC, plus each lead of the discrete components (resistors, capacitors, diodes, and the crystal). As you make each connection, place a check mark beside it on the list. All connections are listed at least twice (once from each end of the jumper), and some are listed more than twice (where more than two pins are connected to the same point). You obviously have to make each connection only once.

You may notice, in comparing the schematic to the wire-list, that some pins called out on the list—the power and ground for each IC and some unused inputs that must be terminated—are not shown on the schematic. This is simply to avoid cluttering the schematic with standard connections: The wire-list is complete. Notice that I

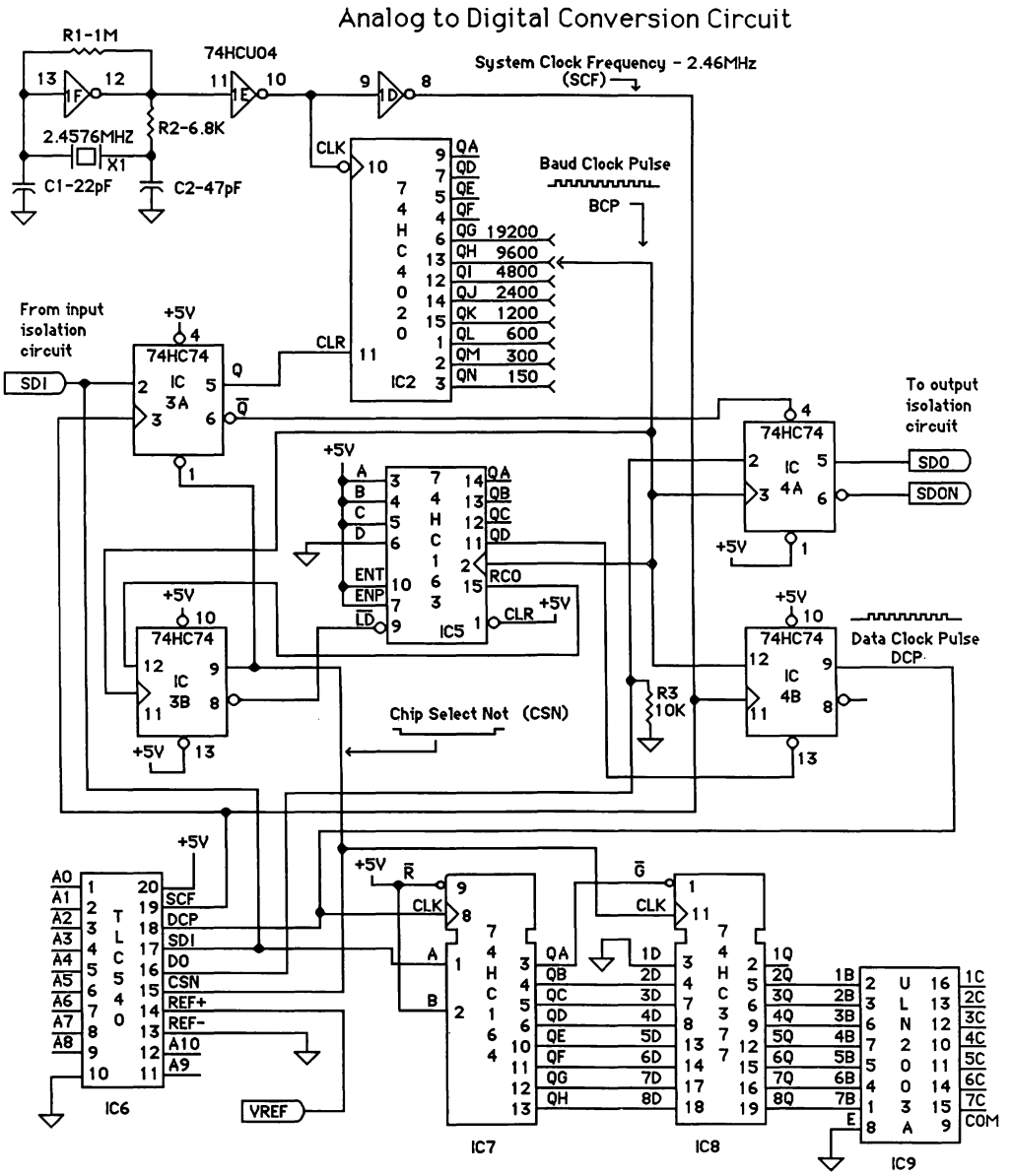


Figure C-3. The system schematic

have labeled the leads of the two-lead discrete components (resistors, capacitors, and diodes) as pin (1) and pin (2), to tell them apart. The numbers are determined by whether the component is drawn horizontally or vertically on the schematic: Pin (1) is the upper or the left lead, pin (2) is the lower or the right lead. The abbreviation *n/c* after an IC pin means that it is not connected.

Master oscillator

IC1: 74HCU04

1 → ground
2 → n/c
3 → ground
4 → n/c
5 → ground
6 → n/c
7 → ground
8 → IC3A(3) → IC4B(11) → IC6(19)
9 → IC1(10) → IC2(10)
10 → IC1(9) → IC2(10)
11 → IC1(12) → R1(2) → R2(1)
12 → IC1(11) → R1(2) → R2(1)
13 → R1(1) → X1(1) → C1(1)
14 → V_{CC}
R1(1) → IC1(13) → X(1) → C1(1)
(2) → IC1(11,12) → R2(1)
R2(1) → IC1(11,12) → R1(2)
(2) → X(2) → C2(1)
C1(1) → R2(2) → X(2)
(2) → ground
C2(1) → IC1(13) → X(1) → R1(1)
(2) → ground
X(1) → IC1(13) → C1(1) → R1(1)
(2) → C2(1) → R2(2)

Baud rate generator

IC2: 74HC4020

1 → baud-n/c
2 → baud-n/c
3 → baud-n/c
4 → baud-n/c
5 → baud-n/c
6 → baud-n/c
7 → baud-n/c
8 → ground
9 → baud-n/c
10 → IC1(9) → IC1(10)
11 → IC3A(5)
12 → baud-n/c
13 → IC3B(11) → IC4A(3) → IC4B(12) → IC5(2)
14 → baud-n/c
15 → baud-n/c
16 → V_{CC}

Data timing and control

IC3: 74HC74

1 → IC3B(9) → IC6(15) → IC8(11)
2 → IC12(1) → R7(2) → IC6(17) → IC7(1)
3 → IC1(8) → IC4B(11) → IC6(19)
4 → V_{CC}
5 → IC2(11)
6 → IC4A(4)
7 → ground
8 → IC5(9)
9 → IC3A(1) → IC6(15) → IC8(11)
10 → V_{CC}
11 → IC2(13) → IC4A(3) → IC4B(12) → IC5(2)

12 → IC5(15)
13 → V_{CC}
14 → V_{CC}

IC4: 74HC74

1 → V_{CC}
2 → R3(1) → IC6(16)
3 → IC2(13) → IC3B(11) → IC4B(12) → IC5(2)
4 → IC3A(6)
5 → R10(1)
6 → IC13(2) → IC14(1)
7 → ground
8 → n/c
9 → IC6(18) → IC7(8)
10 → V_{CC}
11 → IC1(8) → IC3A(3) → IC6(19)
12 → IC2(13) → IC3B(11) → IC4A(3) → IC5(2)
13 → IC5(11)
14 → V_{CC}

IC5: 74HC163

1 → V_{CC}
2 → IC2(13) → IC3B(11) → IC4A(3) → IC4B(12)
3 → V_{CC}
4 → V_{CC}
5 → V_{CC}
6 → ground
7 → V_{CC}
8 → ground
9 → IC3B(8)
10 → V_{CC}
11 → IC4B(13)
12 → n/c
13 → n/c

14 → n/c
15 → IC3B(12)
16 → V_{CC}
R3(1) → IC4(2) → IC6(16)
(2) → ground

Analog-to-digital conversion

IC6: TLC540

1 → A/D input
2 → A/D input
3 → A/D input
4 → A/D input
5 → A/D input
6 → A/D input
7 → A/D input
8 → A/D input
9 → A/D input
10 → ground
11 → A/D input
12 → A/D input
13 → $-V_{REF}$ (ground)
14 → IC17(1) → R13(2) → R14(1) → C12(+)(+V_{REF})
15 → IC3A(1) → IC3B(9) → IC8(11)
16 → R3(1) → IC4A(2)
17 → IC3A(2) → IC7(1) → R7(2) → IC12(1)
18 → IC4B(9) → IC7(8)
19 → IC1(8) → IC3A(3) → IC4B(11)
20 → V_{CC}

Parallel control output

IC7: 74HC164

1 → IC3A(2) → IC6(17) → R7(2) → IC12(1)
2 → V_{CC}
3 → IC8(1)

4 → IC8(4)
5 → IC8(7)
6 → IC8(8)
7 → ground
8 → IC4B(9) → IC6(18)
9 → V_{cc}
10 → IC8(13)
11 → IC8(14)
12 → IC8(17)
13 → IC8(18)
14 → V_{cc}

IC8: 74HC377

1 → IC7(3)
2 → n/c
3 → ground
4 → IC7(4)
5 → IC9(2)
6 → IC9(3)
7 → IC7(5)
8 → IC7(6)
9 → IC9(6)
10 → ground
11 → IC3A(1) → IC3B(9) → IC6(15)
12 → IC9(7)
13 → IC7(10)
14 → IC7(11)
15 → IC9(5)
16 → IC9(4)
17 → IC7(12)
18 → IC7(13)
19 → IC9(1)
20 → V_{cc}

IC9: ULN2003A

1 → IC8(19)
2 → IC8(5)
3 → IC8(6)
4 → IC8(16)
5 → IC8(15)
6 → IC8(9)
7 → IC8(12)
8 → ground
9 → +12 volts
10 → output to LED, relay, etc.
11 → output to LED, relay, etc.
12 → output to LED, relay, etc.
13 → output to LED, relay, etc.
14 → output to LED, relay, etc.
15 → output to LED, relay, etc.
16 → output to LED, relay, etc.

Primary power supply

IC16: TL-780-05

Input → + power adapter → C11(1) → +12 volts
Common → ground → CR6(anode)
Output → +5 volts

C11(1) → +12 volts → IC16 input → + power adapter
(2) → ground
C6(1) → +5 volts
(2) → ground
C7(1) → +5 volts
(2) → ground
C8(1) → +5 volts
(2) → ground
C9(1) → +5 volts
(2) → ground

C10(1) → +5 volts
(2) → ground
CR7(anode) → ground
(cathode) → −power adapter → offset line

Precision voltage reference

IC17: TL431

1 → R13(2) → C12(+) → R14(1) → +3.52 volt line
2 → n/c
3 → n/c
4 → n/c
5 → n/c
6 → ground
7 → n/c
8 → R14(2) → R15(1)

R13(1) → +5 volts
(2) → C12(1) → R14(1) → IC17(1) → +3.52 volt line (+V_{REF})
R14(1) → C12(1) → R13(2) → IC17(1) → +3.52 volt line (+V_{REF})
(2) → R15(1) → IC17(8)
R15(1) → IC17(8) → R14(2)
(2) → ground
C12(+) → IC17(1) → R13(2) → R14(1) → +3.52 volt line
(−) → ground

Input isolation circuit

IC10: TIL124

1 → IC11(2) → +TXD from Mac
2 → IC11(1) → R4(2)
3 → n/c
4 → IC10(6) → IC11(5) → R5(2) → R6(1)
5 → IC11(4,6) → R8(1) → R9(1) → IC12(3,5) → C3(1)
6 → IC10(4) → IC11(5) → R5(2) → R6(1)

IC11: TIL124

1 → IC10(2) → R4(2)
2 → IC10(1) → +TXD from Mac
3 → n/c
4 → R8(1) → R9(1) → IC11(6) → IC10(5) → C3(1) → IC12(3,5)
5 → IC10(4,6) → R5(2) → R6(1)
6 → IC11(4) → R8(1) → R9(1) → IC10(5) → C3(1) → IC12(3,5)

IC12: TLC272

1 → R7(2) → IC3(2) → IC6(17) → IC7(1)
2 → CR1(anode) → CR2(cathode) → IC12(7)
3 → IC12(5) → C3(1) → IC10(5) → IC11(4,6) → R8(1) → R9(1)
4 → ground
5 → IC12(3) → C3(1) → IC10(5) → IC11(4,6) → R8(1) → R9(1)
6 → CR1(cathode) → CR2(anode) → R6(2)
7 → CR1(anode) → CR2(cathode) → IC12(2)
8 → +5 volts

R4(1) → −TXD
(2) → IC10(2) → IC11(1)
R5(1) → +5 volts
(2) → IC10(4,6) → IC11(5) → R6(1)
R6(1) → IC10(4,6) → IC11(5) → R5(2)
(2) → CR1(cathode) → CR2(anode) → IC12(6)
R7(1) → +5 volts
(2) → IC12(1) → IC3(2) → IC6(17) → IC7(1)
R8(1) → IC11(4,6) → IC10(5) → R9(1) → C3(1) → IC12(3,5)
(2) → +5 volts
R9(1) → IC11(4,6) → IC10(5) → R8(1) → C3(1) → IC12(3,5)
(2) → ground
R10(1) → IC4(5)
(2) → IC13(1) → IC14(2)
C3(1) → IC12(3,5) → IC10(5) → IC11(4,6) → R8(1) → R9(1)
(2) → ground

CR1(cathode) → CR2(anode) → IC12(6) → R6(2)
(anode) → CR2(cathode) → IC12(2,7)
CR2(anode) → CR1(cathode) → IC12(6) → R6(2)
(cathode) → CR2(anode) → IC12(2,7)

Output isolation circuit

IC13: TIL124

1 → IC14(2) → R10(2)
2 → IC14(1) → IC4(6)
3 → n/c
4 → IC13(6) → IC14(5) → C4(2) → R11(2) → C5(1) → IC15(6) → + RXD and ground from Mac
5 → IC14(4,6) → IC15(5) → R11(1)
6 → IC13(4) → IC14(5) → C4(2) → R11(2) → C5(1) → IC15(6) → + RXD and ground from Mac

IC14: TIL124

1 → IC13(2) → IC4(6)
2 → IC13(1) → R10(2)
3 → n/c
4 → IC14(6) → IC13(5) → R11(1) → IC15(5)
5 → IC13(4,6) → C4(2) → R11(2) → C5(1) → IC15(6) → + RXD and ground from Mac
6 → IC14(4) → IC13(5) → R11(1) → IC15(5)

IC15: TLC372

1 → n/c
2 → IC15(8)
3 → IC15(4)
4 → CR5(anode) → CR6(anode) → C5(2)
5 → IC13(5) → R11(1) → IC14(4,6)
6 → C4(2) → R11(2) → C5(1) → IC13(4,6) → IC14(5) → + RXD and ground from Mac
7 → R12(2) → − RXD pin 9 on Mac
8 → R12(1) → C4(1) → CR3(cathode) → CR4(cathode)

R10(1) → IC4(5)
(2) → IC13(1) → IC14(2)
R11(1) → IC13(5) → IC14(4,6) → IC15(5)
(2) → C4(2) → IC15(6) → C5(1) → IC13(4,6) → IC14(5) → +RXD and ground from Mac
R12(1) → IC15(8) → C4(1) → CR3(cathode) → CR4(cathode)
(2) → IC15(7) → −RXD pin 9 on Mac
C4(1) → IC15(8) → R12(1) → CR3(cathode) → CR4(cathode)
(2) → IC15(6) → R11(2) → C5(1) → IC13(4,6) → IC14(5) → +RXD and ground from Mac
C5(1) → R11(2) → IC15(6) → C4(2) → IC13(4,6) → C14(5)
(2) → IC15(4) → CR5(anode) → CR6(anode)
CR3(cathode) → CR4(cathode) → R12(1) → C4(1) → IC15(8)
(anode) → CR6(cathode) → −TXD pin 5 on Mac
CR4(cathode) → CR3(cathode) → R12(1) → C4(1) → IC15(8)
(anode) → CR5(cathode) → +TXD pin 4 on Mac
CR5(anode) → IC15(4) → CR6(anode) → C5(2)
(cathode) → CR4(anode) → +TXD pin 4 on Mac
CR6(anode) → IC15(4) → CR5(anode) → C5(2)
(cathode) → CR3(anode) → −TXD pin 5 on Mac

Construction advice

I am not going to provide a wire-by-wire construction account, but I will recommend a few precautions and a preferred order in which to assemble the HBC-1.

It is a recommended construction practice to work on a grounded conductive surface to avoid electrostatic discharge (sparks) that can damage integrated circuits. This can easily be accomplished by laying out one or two sheets of heavy-duty aluminum foil (preferably grounded) on your work area and either resting your forearms on the foil or maintaining contact in some other manner in order to keep everything electrically at the same potential.

When you are handling the crystal, bear in mind that, as its name implies, it is essentially a fragile piece of glass. Treat it gently.

When running the interconnecting wiring on either the poke-home or the wire-wrap board, keep the wires as short as possible to avoid signal cross-coupling, noise pickup, and the effects of stray capacitance.

Connect positive rails to each other and negative rails to each other at three equally spaced points. *Do not connect positive and negative rails directly together.*

Do connect the positive and negative rails with 0.1μF bypass capacitors, placed at five spots evenly distributed about the board.

Assemble the primary power supply section, carefully confirming the placement of each component and wire. The TL780-05 leads, which are slightly flat, can be inserted into sockets in three adjacent rows by carefully putting a 90-degree axial twist in each of the leads so that the flat side of the lead aligns with the row into which it is inserted. This prevents excessive spreading of the socket contacts, which are a little longer than they are wide. You can use a short jumper to connect each lead of the TL780-05 to the appropriate voltage. NOTE: The pin-orientation for the TL780-05 is shown in a drawing adjacent to the primary power supply schematic.

With the exception of capacitor C12 in the precision power supply circuit, none of the capacitors in the HBC-1 are polarized, so it doesn't matter which way the leads go. The diodes, however, must be inserted in the correct direction. The wire-list defines the diode leads as anode and cathode: On the schematic the cathode is the pointed end with the bar across it, and the diode itself has a band at the cathode end.

After assembling the power supply, apply power and use a voltmeter to confirm that the voltage between the rails is 4.95 to 5.05 volts (note this voltage for later comparison) and that the polarity is correct (if an oscilloscope is available, check for ripple; it should not exceed a few millivolts). If you get nothing out of the power supply, make sure you have correctly connected the positive and negative leads from the power adapter: The positive lead should connect to the input pin of the TL780-05. When the power supply voltage has been confirmed, it is safe to continue construction with little fear of damaging the rest of the components.

Turn off the power and plug in the rest of the components. Double-check the orientation of all ICs (there is either a semicircular notch at the pin-1 end, or a recessed dot near pin 1).

Install the short jumpers from each IC to V_{CC} and ground.

Make sure you jumper the unused inputs of the 74HCU04 to the ground rail. The jumper that connects the Baud Clock Pulse (BCP) from the 74HC4020 to the data timing and control circuit should be fairly long and installed last, so that it can be moved to other pins on the 74HC4020 to select different baud rates.

Double-check the ground and V_{CC} jumper to each IC. Measure the resistance between the positive and negative rails. Depending on the make and model of your meter, you should get a reading of several hundred ohms or more. A lower reading indicates an incorrect connection that must be corrected before power is applied.

Connect your voltmeter between the positive and negative rails and plug in the power adapter. If the voltage has dropped more than a few millivolts below the previous reading, indicating a heavy load on the power supply, disconnect immediately and check all components and wiring.

Before actually connecting the Macintosh to the HBC-1, set your ohmmeter to its highest range and verify that there is no continuity between any pin in the cable that you will connect to the Mac and either the positive or the negative rail of the HBC-1. If all checks are correct, you are ready to connect to the Mac.

Connecting to the Macintosh

The HBC-1 communicates with the Macintosh via the modem port. Unlike most modems that can be connected to the Mac, which use only part of the Mac's RS-422 connection to emulate an RS-232 connection, the HBC-1 uses the full RS-422, and therefore requires more than the minimum of three wires typically used in a modem cable. The point-to-point connections for the HBC-1 cable are shown in Figure C-4 on the next page; I will leave to you the decision as to what kind of a connector to use on the HBC-1 end, but the Macintosh end requires a male DB-9 connector like the one for your modem or printer.

Try to keep this cable reasonably short to avoid picking up interference from signals radiated by the power line and other electrical devices, and to maintain reliable communication. I had no problems using it with a 50-foot cable, but 25 feet should probably be the maximum for normal operation.

Macintosh DB-9 connector		HBC-1
Ground	1	Not connected
+5 volts	2	Not connected
Ground	3	IC13(6) on output isolation circuit
+TXD	4	CR4/CR5 on output isolation circuit IC10(1), IC11(2) on input isolation circuit
−TXD	5	CR3/CR6 on output isolation circuit R4 on input isolation circuit
+12 volts	6	Not connected
Handshake	7	Not connected
+RXD	8	Connect to pin 3 of Mac, at either end
−RXD	9	IC15(7),R12 on output isolation circuit

Figure C-4. Macintosh to HBC-1 cable connections

Tracing problems

It is difficult, though undoubtedly not impossible, to assemble the HBC-1 in such a manner that it damages itself. If it doesn't work after you make the final connection and apply power, the problem will probably be traced to a wire you forgot to connect, or connected incorrectly. Troubleshooting is substantially easier if you have access to an oscilloscope, but rudimentary checks can be made with a voltmeter or even a digital logic probe, and most problems will yield to a careful visual scrutiny.

The first thing to do if the system doesn't work is confirm that there is no problem with the program that is sending channel requests to the HBC-1 and attempting to retrieve the responses. If you are using the voltmeter program from Chapter 18 to test the system, you should modify it so that it doesn't wait for a response after each request is sent: If there is no response, the program will lock up at this point. (And make sure that *simFlag* is set to *false.*) An alternative to modifying the voltmeter program is to enter the short program shown in Figure C-5.

```
'*
'*Program to send a character out the communication port.
'*

'*
'* Open communication port.
'*
   OPEN "COM1:9600, n, 8, 2" AS #1
   sentData = &H55
SndChnl:
   PRINT "Sending "; sentData
   PRINT #1, CHR$(sentData);

'*
'* Get a response.
'*
Reply:
   IF LOC(1) = 0 THEN GOTO SndChnl
   returnedData = ASC(INPUT$(LOC(1), 1))
   PRINT "Returned data is "; returnedData
   GOTO SndChnl
```

Figure C-5. A short test program

You can test everything up to the HBC-1 by jumpering a few of the wires in the cable while running this program with the HBC-1 disconnected. (NOTE: Turn off power to the Macintosh and the HBC-1 before disconnecting or connecting wires.) With no wires jumpered, just an indication of the character being sent is printed on the screen while the program runs. Jumpering −TXD to −RXD (the wire from pin 4 on the Mac to the wire from pin 9) will loop the output back to the input, and the send-message and the receive-message should both be printed on the screen. You can disconnect this jumper and jumper +TXD to +RXD (pin 5 to pin 8; disconnect the jumper between pin 8 and ground before conducting this test, and reconnect it after all tests are complete) with the same result. Finally, connect both jumpers, and you should still see both messages printed on the screen.

If the computer and cable check out, you will have to look for a problem in the HBC-1. This should be fairly easy if you have an oscilloscope, as you can compare the waveforms shown later in Figure C-14 and on the individual schematics to what is actually happening. Critical sections to test first are the oscillator section that develops

the System Clock Frequency (SCF), and the input isolation section, which passes the data from the Mac through to the data timing and control section. (Remember, when looking at the data, that it has been converted to backward binary: The 1 that you send from the Mac comes in as 128, 2 as 64, 3 as 192, 4 as 32, and so on.) If both of these sections look good, check the Baud Clock Pulse (BCP) output of the 74HC4020 (shown later in Figures C-10 and C-14). If you use the negative-going edge of Chip Select Not (CSN) as a trigger, you should be able to count the nine pulses plus a blip. Moving on to Data Clock Pulse (DCP), you should see eight pulses. A request to convert the test channel (channel 11; the Mac sends a 208) should produce a response of 124 to 130 from the HBC-1, and a request for any invalid channel (12 through 15) should produce a response of all zeroes.

If you don't have access to an oscilloscope, here are a few common trouble spots to look into:

- Make sure your program's baud rate (set with the OPEN statement) matches the baud rate selected on the 74HC4020.
- Make sure that all unused inputs to the 74HCU04 are connected to ground (actually, they can all go to ground or to V_{CC}, as long as they are all connected).
- Confirm that you have the five bypass capacitors installed between the positive and negative rails.
- Use the wiring list and a check-mark system to reconfirm every connection.
- Use a voltmeter or digital probe to check the quiescent (no data into the HBC-1) levels of the ICs. Between the waveform tables and the text, most of these values are included in this appendix.

If all else fails, make friends with someone who owns an oscilloscope.

The HBC-1 components

I have divided the HBC-1 into nine functional sections, based on the functions of the HBC-1 circuitry, and will explain the purpose and theory of operation of each. Some of these sections are optional or easily modified to operate in some other manner; the following explanation outlines one combination of circuits that work well

together. It is not necessary to read or understand this explanation in order to use the HBC-1, but if you run into problems the extra information may prove useful. The nine sections are:

- Power supply
- Master oscillator
- Baud rate generator
- Serial data input isolation
- Data timing and control
- Analog-to-digital conversion
- Serial data output isolation
- Parallel control output
- Analog signal conditioning

The schematic in Figure C-4 showed the major elements of the HBC-1—everything except the power supply circuits and the input and output isolation circuits. This drawing is primarily to show you how the sections fit together, since it is a little small to actually work from. Larger-scale schematics are included with the explanation of each section.

Power supply

The HBC-1 requires a +5 volt power supply capable of providing about 30 milliamps, but the entire monitoring system can require a variety of voltages, depending on the peripheral devices connected to it. The transducers that provide analog inputs often require a reference voltage, and, of course, the outputs that drive relays, lamps, motors, and so on can require additional power.

The power supply shown in Figure C-6 on the next page consists of two sections, labeled *Primary Power Supply* and *Precision Voltage Reference.* The primary power supply gets its power from a plug-in AC adapter that provides an unregulated 12 volts DC at up to 500 milliamps. This unregulated voltage is used by the TL780-05 to generate a regulated +5 volts DC, which powers the integrated circuits. This section also

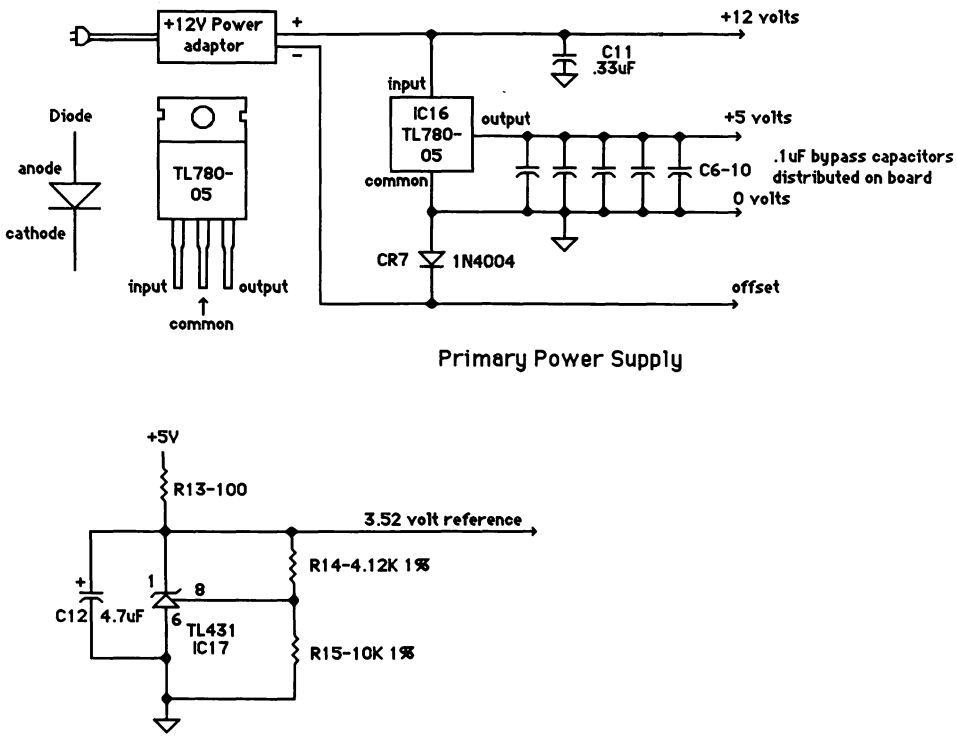


Figure C-6. The power supply schematic

which powers the integrated circuits. This section also passes along the unregulated 12 volts, which can be used to power relays and lamps. (NOTE: If you want to power peripheral 5-volt devices from this power supply, you will have to add a heat sink to the TL780-05.)

The diode in the return line provides polarity protection, preventing smoke if the power-adapter wires are accidentally reversed. A secondary effect of this diode is to make the 0-volt reference for system power float slightly above the power adapter's return line. This very slight difference in potential can be used to generate a negative offset-compensation voltage of a few millivolts, which can in turn be used to zero-set the high-gain amplifiers for signal-conditioning circuits, such as the one used to measure wind speed with an anemometer.

The regulated voltage provided by the primary power supply section (2% regulation) is adequate if you are satisfied with monitoring ratiometric devices such as

joysticks, photocells, and keypads. In this case the 5-volt supply is used as the system reference voltage, in addition to powering the integrated circuits. To monitor devices that have a high output impedance, or that otherwise require op-amp buffering, such as anemometers and thermocouples, the precision voltage reference section is used to provide a stable 3.52-volt reference. There are several advantages to using this:

- It has time and temperature stability of instrumentation quality, providing an excellent source of excitation voltage for strain gauges, thermistors, and anemometers.
- It provides improved short-circuit protection for the power supply, shutting down if the current exceeds about 50 milliamps, which is well within the capability of the power adapter to handle.
- It allows you to use a simple op-amp configuration as a buffer in the signal-conditioning circuit.

Parts used

Quantity	Description	Part
Primary power supply		
1	AC power adapter	Radio Shack #273-1652, 12 volt 500 ma DC output
1	5-volt regulator	TL780-05KC (for 4% regulation a UA7805KC can be used)
1	diode	1N4004 (polarity protection)
1	capacitor	0.33μF (provides stability)
5	capacitors	0.1μF (bypass)
Precision voltage reference		
1	adjustable precision shunt regulator	TL431
1	capacitor	4.7μF
1	resistor	100Ω
1	resistor	4.12KΩ, 1%
1	resistor	10KΩ, 1%

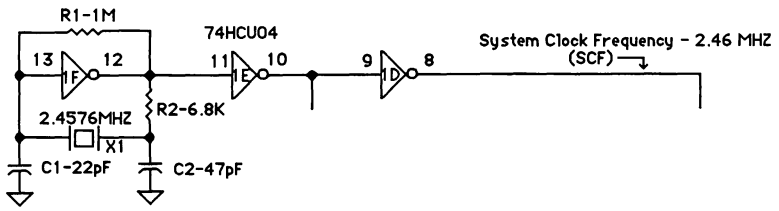


Figure C-7. Master oscillator circuit

Master oscillator

The master oscillator circuit, shown in Figure C-7, develops the System Clock Frequency (SCF) used by the remainder of the circuits—either directly or after it is divided down—for timing.

The 74HCU04 was chosen as the inverter for this circuit due to its relatively low gain of about 10: high enough to guarantee oscillation at 2.4576 MHz, but low enough to suppress spurious oscillations at harmonic frequencies, which could be a problem. The two inverters that buffer the oscillator's output sharpen the waveform's edges, providing clean rise and fall times. The output taken from between the inverters goes only to the baud rate generator, which is negative-edge-triggered. The System Clock Frequency is inverted once more and is used by a variety of positive-edge-triggered devices.

Parts used

Quantity	Description	Part
1	crystal	2.4576 MHz
3	inverters	Half of a 74HCU04 hex inverter chip
1	resistor	1MΩ
1	resistor	6.8KΩ
1	capacitor	22pF
1	capacitor	47pF

Baud rate generator

This circuit is composed of a single chip: the 74HC4020 binary ripple counter. The counter has 14 stages, each of which divides the output frequency of the previous stage by 2. Of the 14 frequencies developed, the first and the fourth through the fourteenth are available as outputs. With the clock frequency of 2.4576 MHz provided by the master oscillator circuit, the eight lowest frequencies match standard baud rates available in the Macintosh and most other computers (Figure C-8).

The master oscillator frequency we are using provides most of the Macintosh baud rates; most of the rest—1800, 3600, 7200, and 57600—could be matched by changing the crystal in the master oscillator to one with a frequency 1.5 times as high. This would not be a standard crystal, and there is little point in actually doing this unless you have a specific need for one of these baud rates (I have, however, run the

Stage frequency	Output	Pin	Mac baud rates
1228800	QA	9	
614400	n/a	n/a	
307200	n/a	n/a	
153600	QD	7	
76800	QE	5	
			57600
38400	QF	4	
19200	QG	6	19200
9600	QH	13	9600
			7200
4800	QI	12	4800
			3600
2400	QJ	14	2400
			1800
1200	QK	15	1200
600	QL	1	600
300	QM	2	300
150	QN	3	150
			110

Figure C-8. 74HC4020 outputs and Macintosh baud rates

HBC-1 up to 57600 to prove that it will work). The only remaining baud rate—110—could also be matched, but I can think of no practical reason for running the HBC-1 at this low speed.

Parts used

Quantity	Description	Part
1	Asynchronous 14-bit binary counter	74HC4020

The 74HC4020

The 74HC4020 is an asynchronous 14-bit binary ripple counter—a pretty fancy name for an integrated circuit that is the functional equivalent of a bunch of nested FOR...NEXT loops. Figure C-9 shows the logic diagram for the 4020.

The input on pin 11 disables or enables the counter: A high at this input disables the counter and resets all outputs to zero; a low on pin 11 enables the counter, allowing

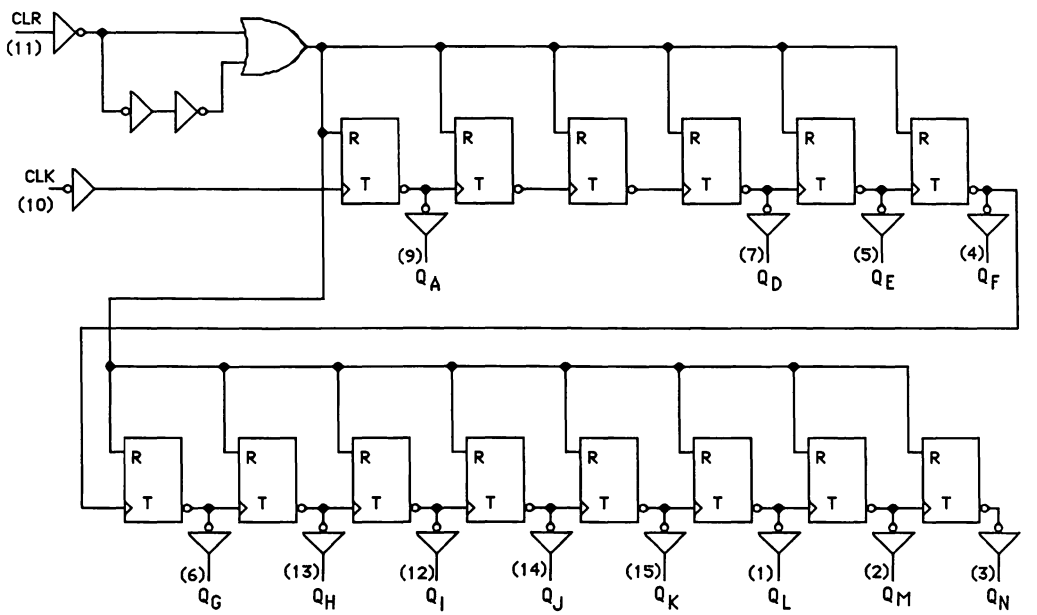


Figure C-9. Logic diagram of the 74HC4020

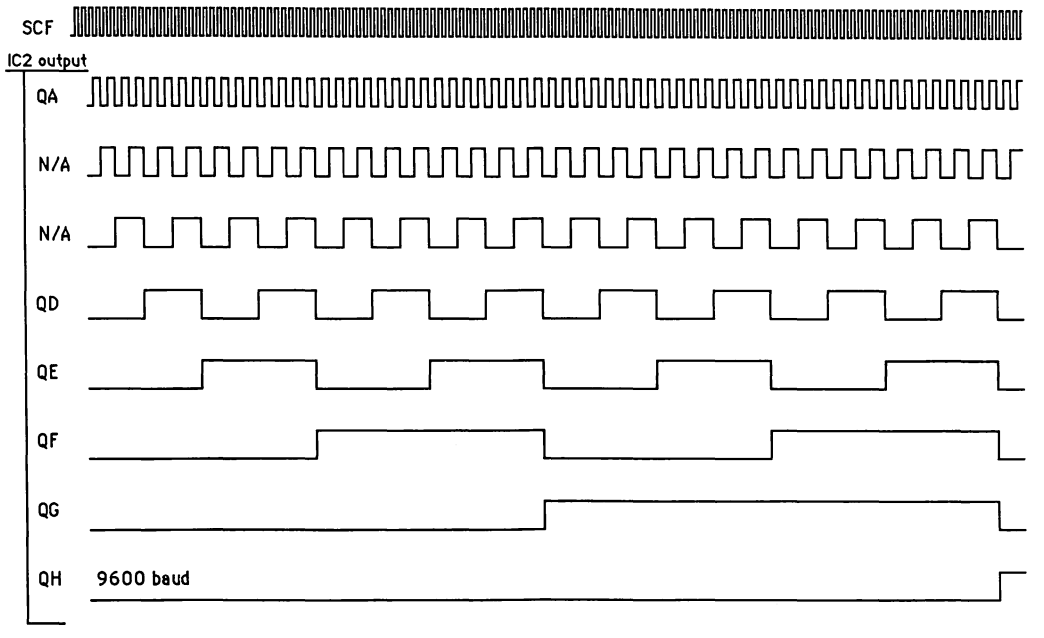


Figure C-10. Ripple-counter waveforms

the system clock frequency (SCF) to ripple through the counter (the word "ripple" is used because the ouput of one stage is the input to the next).

Each stage of the counter is triggered by the negative-going edge of its input, which happens once in each input cycle, so each stage divides its input frequency in half. The primary clock frequency and the waveforms after each of the first eight stages are shown in Figure C-10. You can imagine how the rest of the stages would continue to divide the clock frequency.

There are not enough pins on the integrated circuit to make the outputs of all 14 stages available. As you saw in the logic diagram, the second and third stages of the counter are skipped. The 12 stages that are available provide common baud rates that are binary multiples of 150. Figure C-11 on the next page shows an enlarged view of the baud rate generation portion of the system schematic.

Only one of the outputs—usually one between 1200 and 19200—is used at any one time: It must be matched to the baud rate specified in the OPEN statement used to open the modem port for communication with the HBC-1. Whichever Baud Clock Pulse (BCP) is selected, it will be used by the data timing and control circuit to create

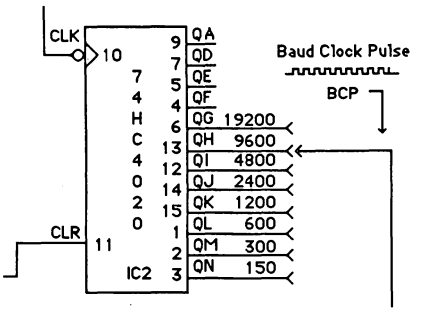


Figure C-11. Baud rate generator circuit

pulses and levels that step the requests and commands from the Macintosh into the HBC-1, and the responses back out to the Mac.

Serial data input isolation

The primary purpose of this circuit is to provide signal isolation between the HBC-1 and your Macintosh, ensuring that no voltages present in the Macintosh can harm the HBC-1. The output from the HBC-1 is similarly isolated to protect the Macintosh. In a very general way, this isolation is accomplished in much the same manner as the historical passing of messages between ships at sea: with flashing lights. The TIL124 used in the circuit shown in Figure C-12 converts the voltage signal arriving

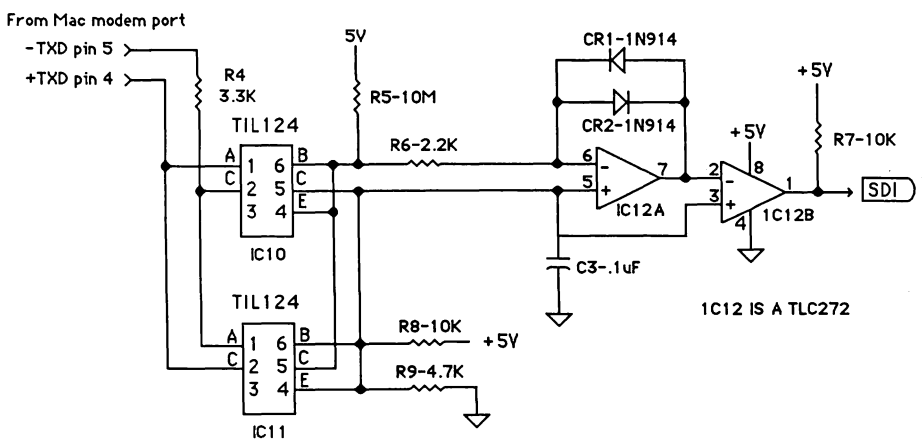


Figure C-12. Input isolation circuit

from the Macintosh to light, beams the light across a small gap, and then converts it back to a voltage signal. There is no hard-wired electrical connection between the two machines, and the voltage range of the signal inside the HBC-1 is independent of the voltage range of the signal from the Mac.

Parts used

Quantity	Description	Part
2	optical isolators	TIL124
1	dual operational amplifier	TLC272
2	diodes	1N914
1	resistor	2.2KΩ
1	resistor	3.3KΩ
1	resistor	4.7KΩ
2	resistors	10KΩ

Theory of operation

The signal from the Macintosh is applied to the inputs of the two TIL124 optical isolators. Notice that the positive input goes to the anode of one optical isolator and to the cathode of the other, and that the negative return is connected to the other input of each optical isolator. This ensures that one TIL124 or the other, but never both, will always be conducting. Whichever TIL124 is conducting generates an output current at its base collector outputs. The outputs of the TIL124s are tied together so that as the input voltage reverses polarity, the output follows. This output is applied to the inputs of the TLC272 op-amp, which is configured as a current-summing amplifier. The 1.6-volt bias level produced by R8 and R9 ensures that the op-amp, with back-to-back diodes in its current feedback path, will switch sharply when the input voltage from the Mac changes state. Resistor R6 is included to improve the stability of the circuit. The output of the first stage of the TLC272 is fed into the second stage, which is operating as a comparator. The output of this stage, in conjunction with the 10K pull-up resistor to V_{CC}, produces a CMOS-compatible output that is stable with inputs to the circuit ranging from 1 milliamp to over 10 milliamps.

Resistor R6 was added to eliminate the possibility that random noise could trigger the parallel digital outputs if the HBC-1 were disconnected from the Macintosh but left turned on.

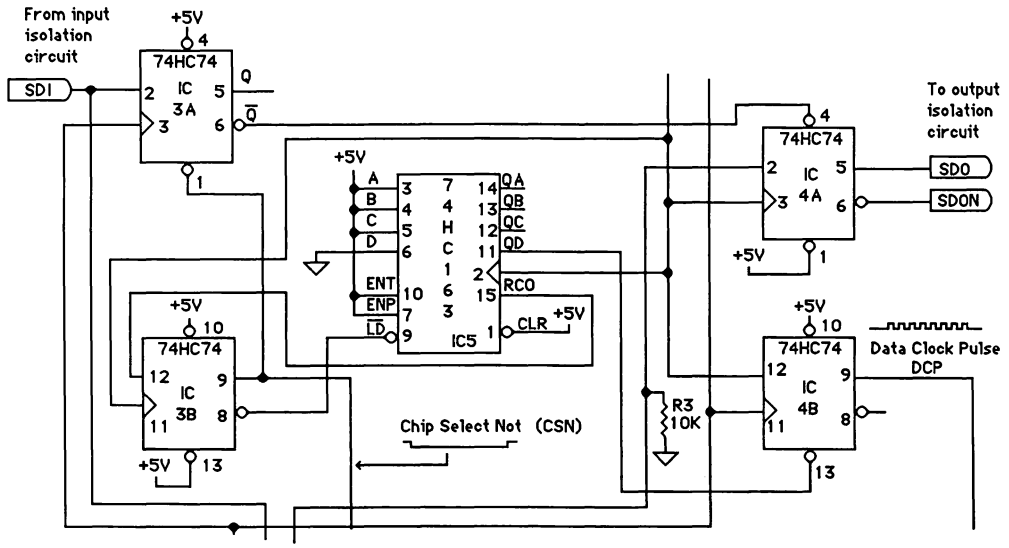


Figure C-13. Data timing and control schematic

Data timing and control

The data timing and control circuit, shown in Figure C-13, is probably the most difficult circuit in the HBC-1 to explain. The reason for the difficulty is that multiple events occur simultaneously, triggering other events; but the explanation must be linear, covering one event at a time.

The waveform diagram in Figure C-14, which shows the relationship of the various clock pulses and enable levels created or affected by this circuit, may provide a perspective on my linear explanation. Notice that significant times in the diagram are marked by vertical lines dropping from the top.

The data timing and control circuit responds to serial data from the computer by enabling the baud rate generator and then using the resulting Baud Clock Pulses (BCP) to create eight Data Clock Pulses (DCP) that load the channel request into the analog-to-digital conversion circuit and transmit the conversion result (of the previous request) back out. Let's look at the conditions in this circuit while it is waiting for a binary character to come from the computer, and then we will follow the character through the HBC-1.

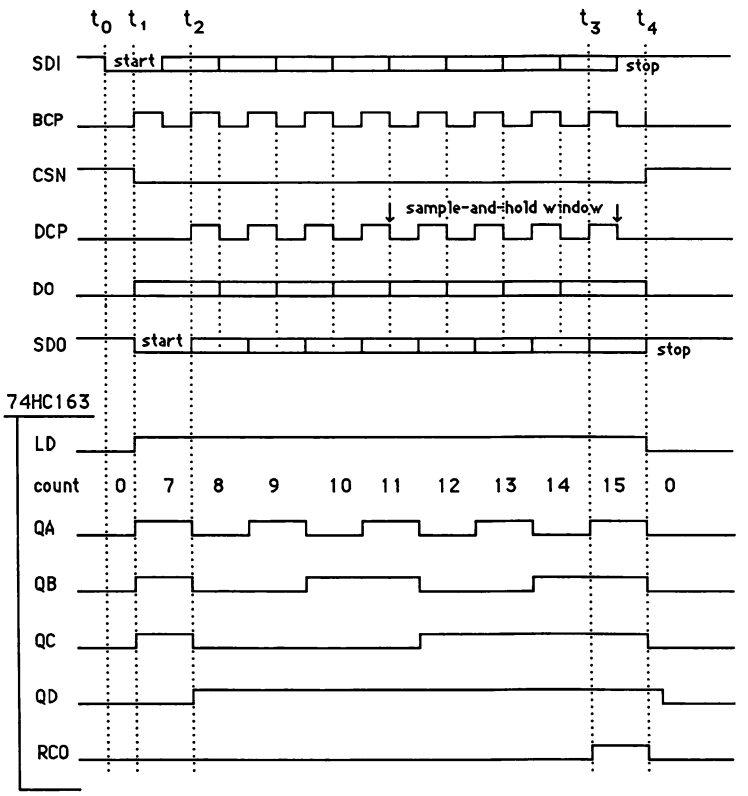


Figure C-14. Timing waveforms

While awaiting data

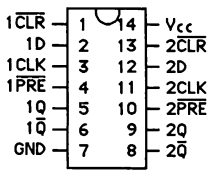
While awaiting data, the Clear (CLR) and the Preset (PRE) inputs to IC3A are high (PRE is tied to +5 volts, CLR is held high by Chip Select Not (CSN), the Q output of IC3B on the next page). This means that the level on the data input to IC3A will be latched through to the Q output by each positive-edge of the SCF. Between Macintosh output bytes, the Serial Data In (SDI) is high, so a high appears on the output of IC3A (see the truth table for D flip-flops in Figure C-15 on the next page).

The high from IC3A Q output is applied to the CLR input (pin 11) of the 74HC4020 asynchronous binary counter, disabling that chip and holding all its outputs low, so there is no BCP.

Function Table

INPUTS				OUTPUTS	
$\overline{PRE}$	$\overline{CLR}$	CLK	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H†	H†
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	Q_0

†This configuration is nonstable; that is, it will not persist when Preset or Clear returns to its inactive (high) level.



Logic Diagram (each flip-flop)

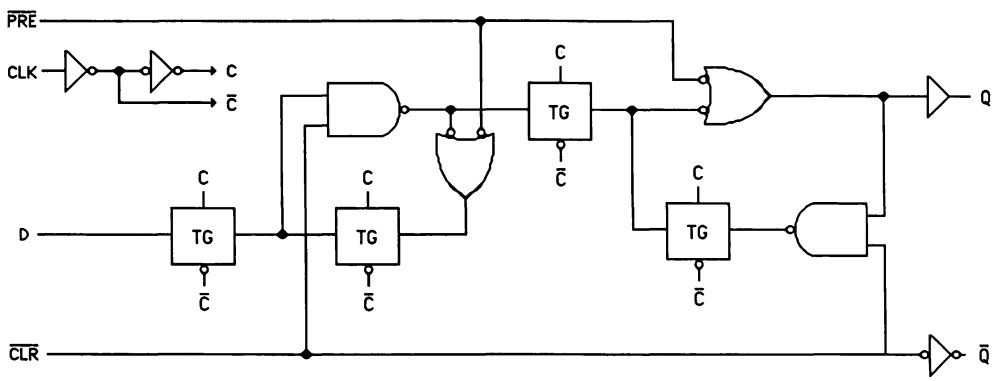


Figure C-15. Logic diagram, function table, and pinout for D flip-flops

The four preset inputs of the 74HC163 synchronous binary counter are hard-wired to 1, 1, 1, and 0, ready to set the outputs to a count of 7. The low level on the 163's Load input (LD—pin 9), provided by IC3B not-Q output, holds the 163 in a state of readiness to have the preset inputs passed through to the outputs. The positive edge of the first BCP (t_1) into the 163 will set the output to 7, but until that happens the four outputs are each at 0.

The low from the QD output of the 163 is applied to the CLR pin of IC4B, holding its output low.

Data arrives

Serial Data Input (SDI) from the Macintosh, after passing through the isolation circuit, appears on the data-input pin of IC3A. The character from the Macintosh consists of a start bit, which is low, eight data bits, and a stop bit, which is high (the input remains high until the next character is received).

When SDI changes to a low state at the data input to IC3A, indicating the start of a character (time t_0 on the waveform table), the next SCF (remember that this is a high-frequency, 0.4 μSec clock, so the next SCF occurs almost instantly) latches the Q output low, putting a low level on pin 11 of the 74HC4020 and allowing it to start counting. (The not-Q output of IC3A is applied to the PRE input of IC4A.) It will take one half of the start-bit width for the BCP to appear at the selected output: This delay is put to good use. It is possible that noise or some other glitch could appear on the input to IC3A and start the timer. This is pretty much avoided because the timer will be reset if the level on pin 11 returns high prior to t_1, which would happen if a low glitch came and went away. So pin 11 is held low for the length of the start bit. But we will need a total of ten BCP out of the counter (as you can see on the timing diagram, nine full BCP plus the leading edge of the tenth, which stops the entire process), so the counter will have to remain enabled after the start bit ends and a data bit (possibly a high) appears at IC3A. For this to happen, the output of IC3A must be held low by some means.

One output from the 74HC4020 counter has been selected (hard-wired) to be used as the BCP to provide timing for the data timing and control circuit. As the circuit is presently drawn, with a 2.4576 MHz crystal providing the clock input to the 4020, the output of 9600 Hz appearing on pin 13 is used to provide baud rate timing (the computer should be set for 9600 baud, no parity, eight data bits, one stop bit). This signal is applied to the data input of IC4B, and the clock inputs of IC3B, IC4A, and the 74HC163 synchronous binary counter. Let's look at the effect of the first BCP, at time t_1 on the timing diagram.

Time t_1

The first positive-going transition of BCP (t_1 on the waveform diagram in Figure C-15) clocks the low on IC3B's input through, latching the Q output (CSN) low, and the not-Q output high. CSN is applied to IC3A's CLR input, disabling the flip-flop and

holding its Q output low (which maintains the needed enable level on the 74HC4020). You can see on the waveform diagram that CSN stays low long enough to bring one character in and send one out. CSN is also applied to pin 15 of the TLC540 analog-to-digital converter, enabling it.

With the first BCP edge, the HBC-1 is committed to the fact that this is a valid start bit, so it starts sending out the response to the previous conversion request. Data Out (DO) is clocked out through IC4A, which has been enabled ever since a few SCF cycles after the SDI went low at t_0 (the high from IC3A not-Q output was applied to IC4A PRE). The data input to IC4A is the DO from the TLC540 (you'll have to look at the main schematic to confirm this). The output pin of the 540, however, was at a high impedance until a few SCF cycles after the first BCP. This high impedance state allows the 10K resistor between IC4A's data input and ground to define a low state until just after the first BCP, when the output of the 540 presents a low impedance output, allowing the TLC540 DO to define the most significant bit (MSB) of the response data. This initial low causes the data input to IC4A to be low when the BCP at t_1 latches the output of IC4A low. This low is the start pulse for the Serial Data Output (SDO), which is the response going back to the Macintosh.

The initial outputs of the 74HC163 are all 0s, as that is where the 163 stopped counting at the end of the previous character. The outputs following t_1 are determined by the preset levels on the A, B, C, and D inputs, which are 1, 1, 1, and 0. With these inputs, the chip can load a count of seven. The first positive-going BCP (time t_1) applied to the clock input of the 74HC163 causes its outputs to go to the preset state, and the counter to start counting.

Time t_2

At time t_2, which corresponds to the leading edge of the second BCP, the QD output of the 163 goes high. This level is applied to the CLR input to IC4B, enabling it and allowing SCF to gate BCP through, creating eight Data Clock Pulses (DCP) from the BCP signal. DCP is applied to pin 18 of the TLC540 as I/O clock.

From t_2 to t_4

The positive-going edge of each DCP performs two functions: It samples the SDI bit and loads it into the TLC540—the first four bits are used to define the address

of the next channel to be converted—and it clocks the DO bit that is currently at the input of IC4A out as SDO. When QA goes high on count fifteen of the 163, the Ripple Carry Output (RCO) also goes high (time t_3). RCO is applied to the data input of IC3B, and the final short BCP (time t_4) triggers the flip-flop, latching Q high (resetting CSN) and not-Q low. Resetting CSN allows IC3A to again latch SDI through. The data input at this time should be the stop bit, so the next SCF positive edge will latch it through, thus stopping BCP by resetting the 74HC4020. The not-Q output from IC3A goes low, forcing the output of IC4A high, thereby creating the outgoing stop bit, which is high.

Parts used

Quantity	Description	Part
2	dual D-type flip-flops with clear and preset	74HC74
1	synchronous 4-bit binary counter	74HC163
1	resistor	10KΩ

The 74HC74

Flip-flops get their name from the action they perform. Typically, a flip-flop has two outputs, one of which—at any given moment—is high and the other low. Upon the occurrence of the proper input condition, the outputs flip—the high one going low and the low one high. The 74HC74 contains two independent flip-flops, each with a Preset (PRE), a Clear (CLR), a Clock (CLK), and a Data (D) input, and two outputs (Q and not-Q). PRE and CLR are used to force the outputs to specific conditions: A low on PRE forces Q high and not-Q low—called setting the flip-flop—while a low on CLR forces the opposite conditions—called resetting the flip-flop. A low on either of these inputs overrides the CLK and D inputs. The table in Figure C-15 showed these input/output relationships.

If both PRE and CLR are high, then the Q output follows the condition of the D input at each positive-going clock pulse (the not-Q output is always the opposite of the Q output).

The 74HC163

This counter is essentially composed of four gated flip-flops that are interconnected. The fact that this is a synchronous counter means that the outputs can change only on the positive edge of the clock pulse, regardless of when any control input

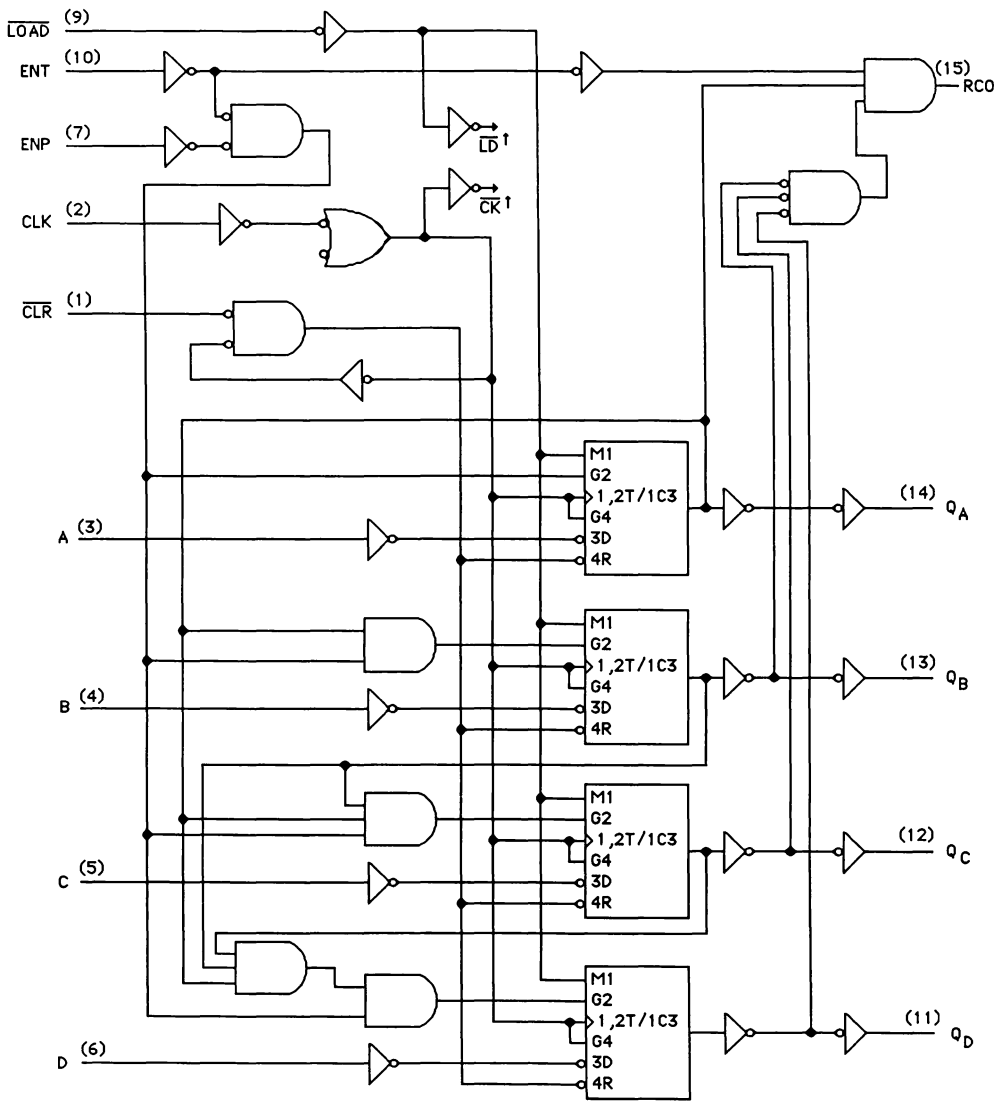


Figure C-16. Logic diagram for the 74HC163

change is made. An important feature of this counter that makes it useful in our application is the ability to preset the output to a specific value. The first BCP that occurs after the Load (LD) input goes low passes the preset value to the output. Proper enabling and removal of the LD input allows the counter to start counting, incrementing the output by 1 with each BCP. When the output reaches 15, it "rolls over" to zero at the next count. In our circuit, we stop the clock pulses after the return to zero, so the counter stops. Figure C-16 shows the logic diagram for the 74HC163.

The data inputs (A, B, C, and D), which are used to preset the outputs, and the outputs (QA, QB, QC, and QD—see Figure C-17) use low- and high-voltage levels to represent binary numbers from 0000 to 1111 (0 to 15 decimal). In our application we determine the load value of the counter to be 7 by connecting A, B, and C to +5 volts

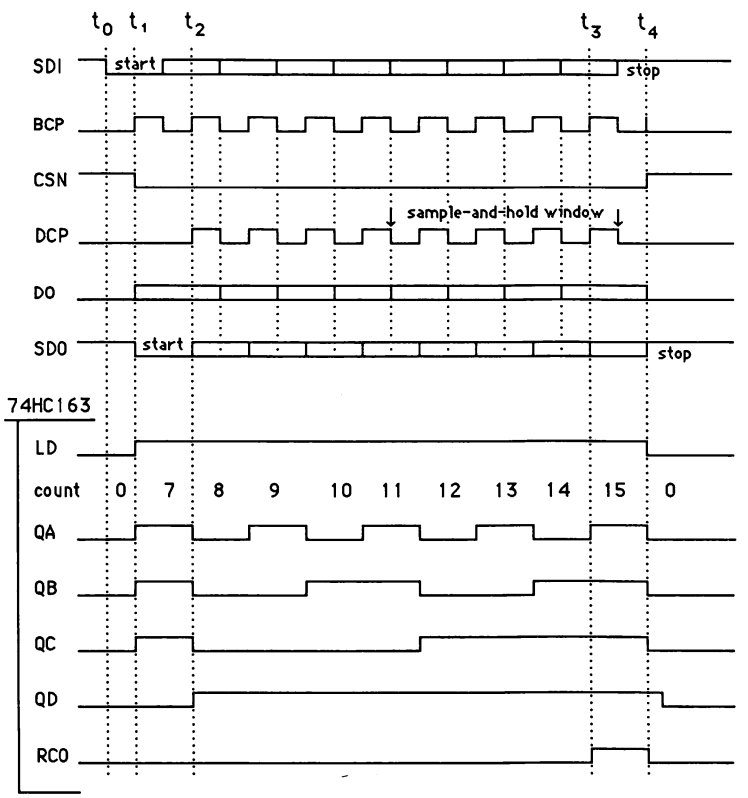


Figure C-17. 74HC163 waveforms

(a high), and D to ground (0 volts, a low). Typically, this counter is enabled and disabled by the Enable inputs (ENT and ENP), but in this application we have tied both to +5 volts, so the counter is always enabled and loads the preset count and starts counting on the first positive-going clock pulse after the LD input goes low. Normal up-counting occurs after LD goes high. The waveforms in Figure C-17 show one full count-cycle, from 7 through 15.

The two outputs of the 74HC163 that are used by the data timing and control circuit of the HBC-1 are the QD output and the RCO output.

Analog-to-digital conversion

The TLC540, shown in Figure C-18, is the heart of the HBC-1. A single integrated circuit, it has a built-in channel selector (multiplexer) enabling it to monitor 11 input channels (plus a self-test channel). It also has circuitry to convert the analog levels measured to digital values, decode the channel-number requests, and make the converted values available at its data output pin.

Several other important features of the TLC540 are its short data-conversion cycle time of 13 microseconds, and the fact that it is capable of 0.5 LSB accuracy.

Parts used

Quantity	Description	Part
1	8-bit analog-to-digital converter with serial control and 11 inputs	TLC540

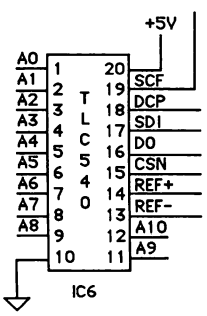


Figure C-18. The analog-to-digital converter schematic

Theory of operation

As explained in the data timing and control section, the DCP clocks eight data bits into the TLC540. The first four data bits are recognized as the number of the channel to read. This channel is sampled from the end of the fourth DCP through to the end of the eighth DCP (the sample-and-hold window). At the falling edge of the eighth DCP, the analog value of the selected channel is held and the conversion process, which requires 36 SCF cycles to complete, is started.

The result of this conversion will be available when the next channel address is clocked in. During the same time period, DCP also clocks the response to the previous conversion request out of the TLC540 to IC4A as DO.

To read the value present at the input of a channel, two bytes from the computer are required. The first specifies the channel and converts the data; the second retrieves the result. If you are repeatedly reading one or more channels, the address byte for the second channel can be used to retrieve the result of the first request.

Serial data output isolation

The output isolation circuit shown in Figure C-19 protects the computer from the voltages measured or generated in the HBC-1.

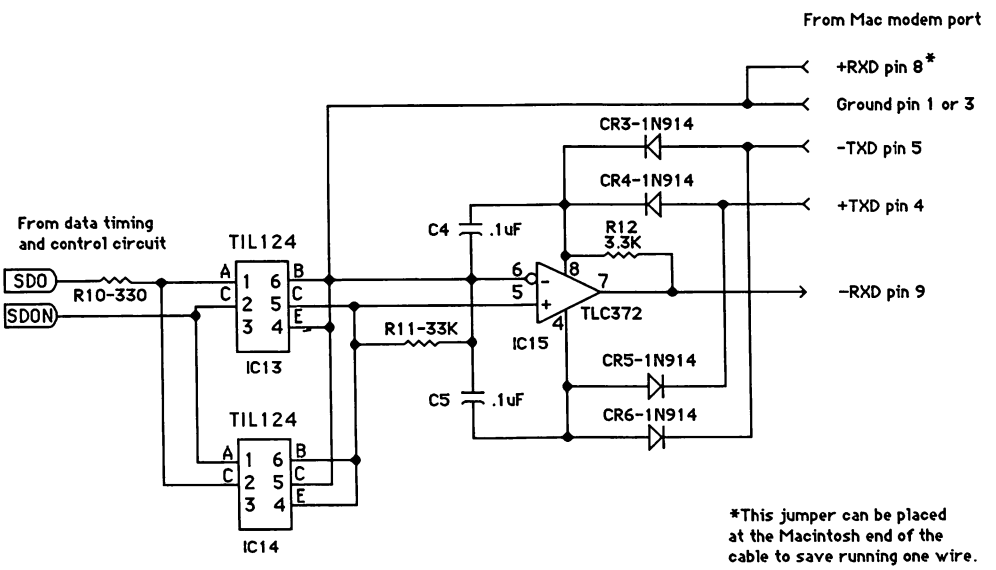


Figure C-19. Output isolation circuit

Parts used

Quantity	Description	Part
2	optical isolators	TIL124
1	differential comparator	Half a TLC372N (an LM393N can be substituted)
4	diodes	1N914
1	resistor	330Ω
1	resistor	3.3KΩ
1	resistor	33KΩ
2	capacitors	.1μF

Theory of operation

The two outputs from the data timing and control circuit (SDO and its inverse, SDON) are applied to the opto-isolators via a 330Ω resistor, creating a 10 milliamp current. One opto-isolator or the other will always be conducting. The opto-isolator photo diode (base to collector) output current across the 33K load resistor creates an input voltage exceeding plus or minus 100 millivolts to the TLC372 comparator. The comparator's output, with the aid of the 3.3K pull-up resistor, provides the output from the HBC-1 to the Macintosh (−RXD on pin 9 of the Macintosh). The four 1N914s are connected as a diode bridge to provide a self-powered interface; that is, it is powered only by the voltage difference between the −TXD and the +TXD's signal from the Mac. This avoids having to supply 5 or 12 volts from the Mac or from an external power supply.

Parallel control output

The section of the HBC-1 shown in Figure C-20 controls output voltage levels, which can be used to activate relays, LEDs, or other low-current devices. The number of outputs can be expanded, practically infinitely, in groups of seven.

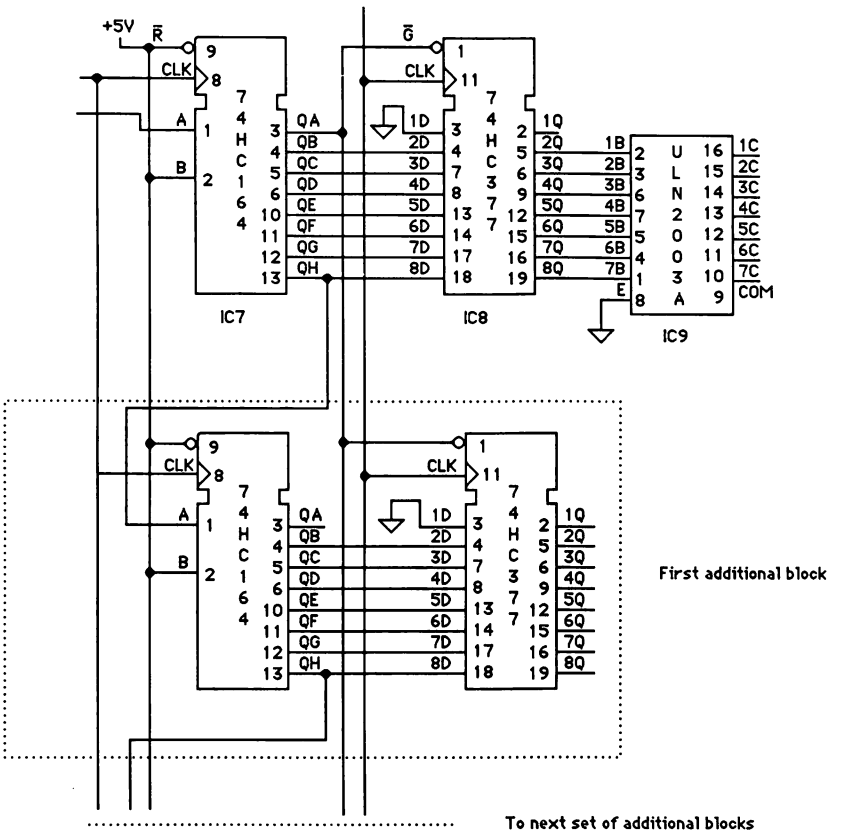


Figure C-20. Parallel control output circuit

Parts used

For each group of seven outputs, the following parts are used:

Quantity	Description	Part
1	8-bit parallel-out serial shift register	74HC164
1	octal D-type flip-flops with clock enable	74HC377
1	Darlington transistor array	ULN2003A

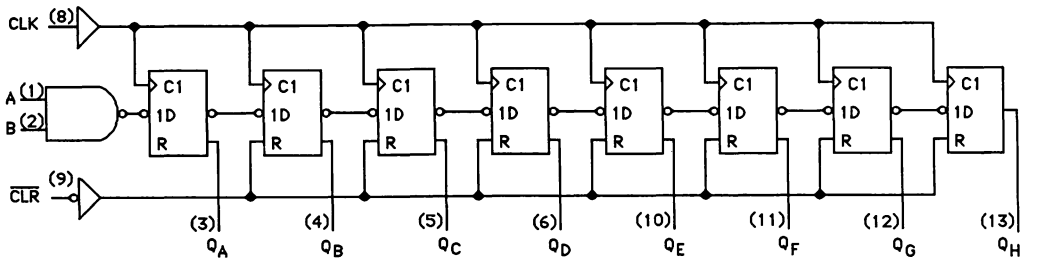


Figure C-21. Logic diagram for the 74HC164

The 74HC164

A shift register is essentially a small storage area; this particular shift register holds 8 bits, which is the length of one character sent to the HBC-1. As its logic diagram in Figure C-21 indicates, the 74HC164 consists of eight flip-flops, each after the first getting its input from the previous one.

The bits come into the shift register as serial data—that is one after the other—and are gated through by clock pulses (DCP in this case). After eight clock pulses, the channel request/digital output data from the computer is fully loaded into the shift register. There are eight outputs from the 74HC164, and at any point the bits that have been loaded can be read out as a parallel byte.

The 74HC377

The 74HC377 is another set of eight flip-flops, this time with parallel input and parallel output. Though the individual flip-flops are not connected to each other, they do share common clock and enable signals, as shown in Figure C-22.

If the enable (not-G) is low when a positive-going clock pulse occurs, then whatever levels are on the eight inputs (1D through 8D) are clocked into the flip-flops and appear on the eight outputs (1Q through 8Q).

Theory of operation

The serial data coming from the computer is fed into the shift register and appears in parallel on its outputs, most of which are also the inputs to the 74HC377. The exception is the last bit in, which has a special function: It is used as the enable for the 74HC377. If this bit is low, the next positive-going clock pulse (the positive-going edge

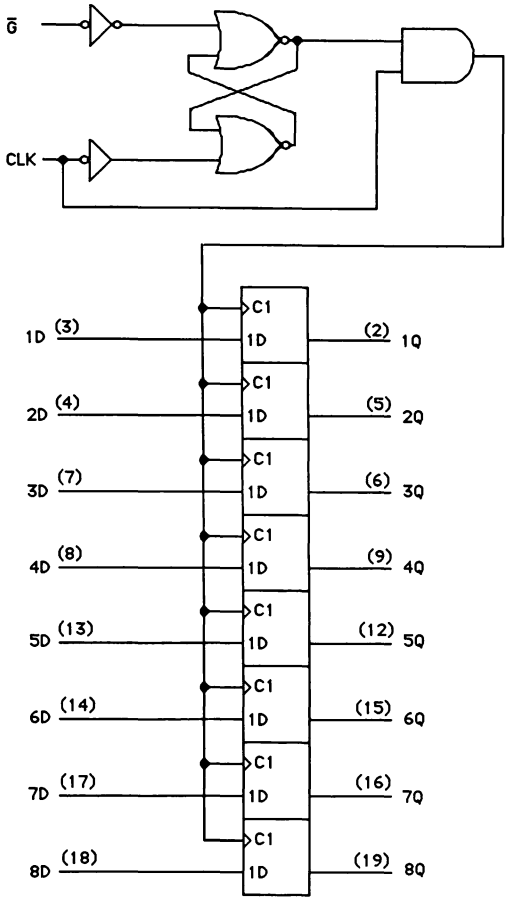


Figure C-22. Logic diagram for the 74HC377

of CSN) will gate the other seven bits into the 74HC377, and they will appear on its outputs. If this bit is high, the 74HC377 does not gate the data to its outputs, leaving the previous data there. Either way, the TLC540 is processing the same data stream and considers the first four bits to be an address request, which it attempts to process.

What all this means is that if you want to set the parallel output, you send an ASCII character to the HBC-1 consisting of seven bits that represent the seven desired outputs, and an eighth bit that is low. Since each character sent to the HBC-1 results in a response from the TLC540, the program must read the input after each output command and discard it, to keep everything straight (requests for an invalid channel return a value of 0).

Analog signal conditioning

Each of the analog inputs to the TLC540 is limited to a voltage range of 0 to 5 volts. If you want to measure a voltage that matches this range, you can use the 5-volt power supply as V_{REF} and apply the voltage to be measured directly to the TLC540 input channel with no signal conditioning required. You can also measure ratiometric devices based on variations in resistance, such as joysticks, potentiometers, thermistors, and photoresistors using the simple configurations shown in Figure C-23.

These configurations will provide accurate measurements as long as the transducer source resistance is 10K or less. The variable resistor in the first circuit represents one of the two potentiometers in a joystick. You could use two channels of the HBC-1 to monitor both potentiometers to control a pointer on the Macintosh screen

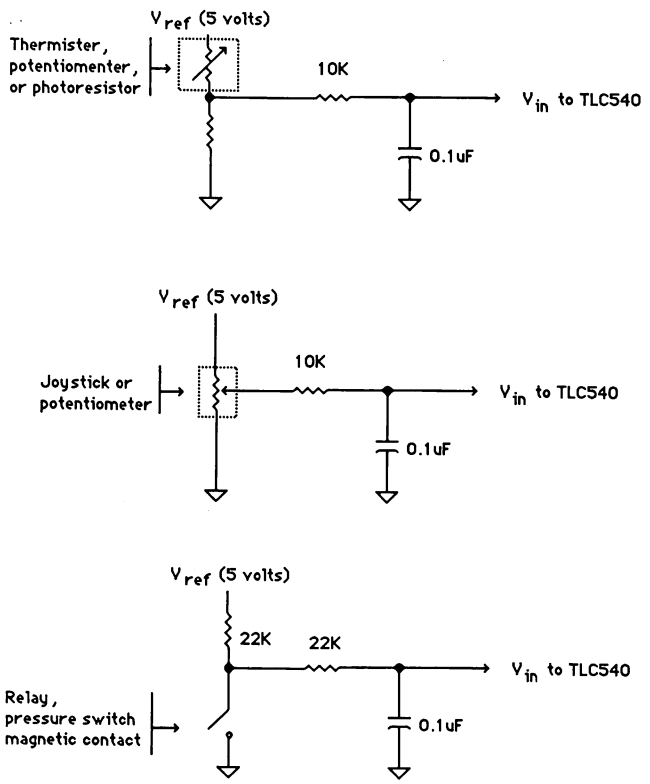


Figure C-23. Configuration for ratiometric input circuits

just as the mouse does, or to perform some other function. The variable resistor in the second circuit could represent a thermistor, a potentiometer, or a photoresistor. A thermistor varies in resistance as the temperature changes, and a photoresistor varies in resistance as the amount of light changes. There are inexpensive versions of each of these available at different sensitivities and response speeds. By matching the response speed of the photoresistor to the expected speed of light-level change, you can use them to measure light levels that vary from as slowly as the sunlight beaming through your living room window to as rapidly as a light beam broken by a car as it enters your driveway.

To accurately measure voltages that are higher, or substantially lower, than 5 volts, or to accurately measure the output of a transducer with a source resistance that is much greater than 10K, some sort of buffer amplifier is required. This buffer amplifier provides additional protection for the TLC540 inputs against the accidental application of excessive voltage, and matches the specific transducer output to the 0 to V_{REF} range. This is where the optional precision voltage reference described in the power supply section comes in. The schematic in Figure C-24 on the next page shows how you could buffer the output from an anemometer to accurately measure wind speeds from near 0 to approximately 100 miles per hour (the mechanical anemometer itself is not very accurate below 2 mph, and I haven't actually wandered out into a 100-mph wind to test the upper end).

By changing the values of R1 and R2 in this circuit you can vary the gain of the TLC27L2, which determines the full-scale input range of the circuit. The gain is computed by the formula:

$$\text{Gain} = 1 + (R2 / R1)$$

So in the configuration shown, the gain is equal to 101. If you decide to change the gain, keep R2 in the 200K to 400K range in order to maintain a reasonable response time and keep the offset circuit operating properly. The maximum full-scale input is determined by the formula:

$$\text{Full-scale input voltage} = V_{REF} / \text{Gain}$$

which yields 34.9 millivolts in this case.

Another simple circuit that can be easily modified to cover a variety of applications is the buffer shown in Figure C-25 on page 529, which is used in conjunction with a resistance network to form a four-range, bipolar digital voltmeter.

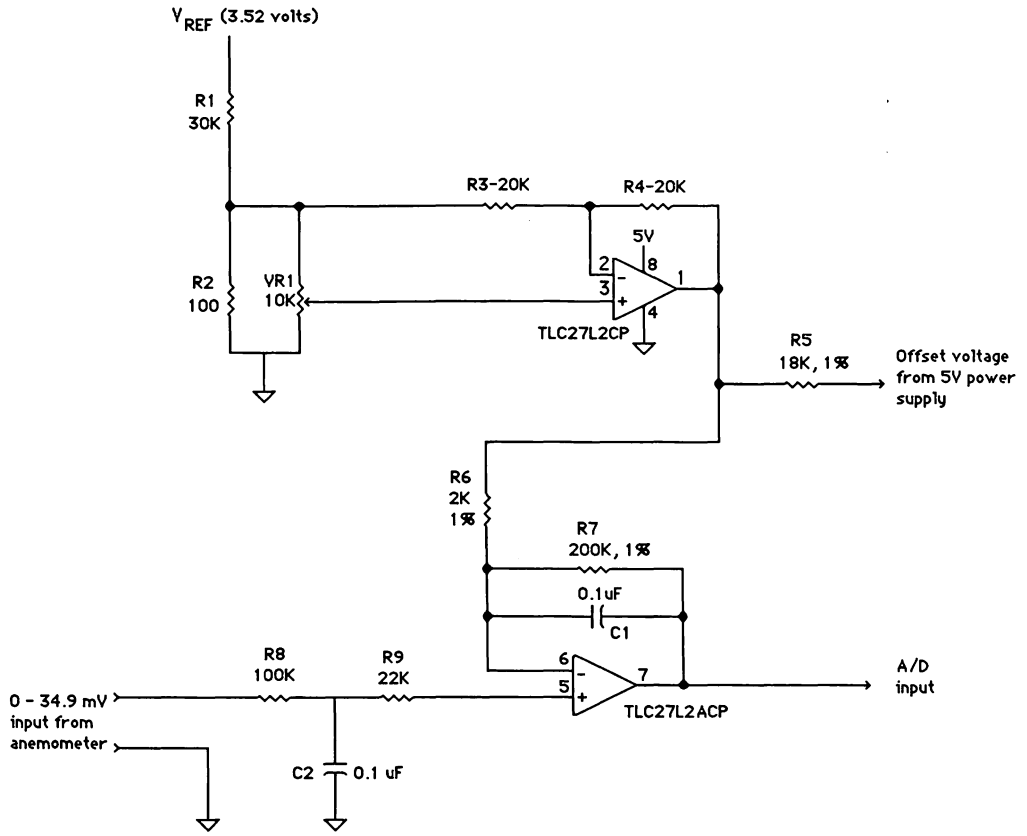


Figure C-24. Monitoring an anemometer

The four ranges are determined by the ratio of resistor R1 to each set of level-translation resistors (R5 and R8, R4 and R7, R3 and R6, and R2). With the values shown, the four ranges are:

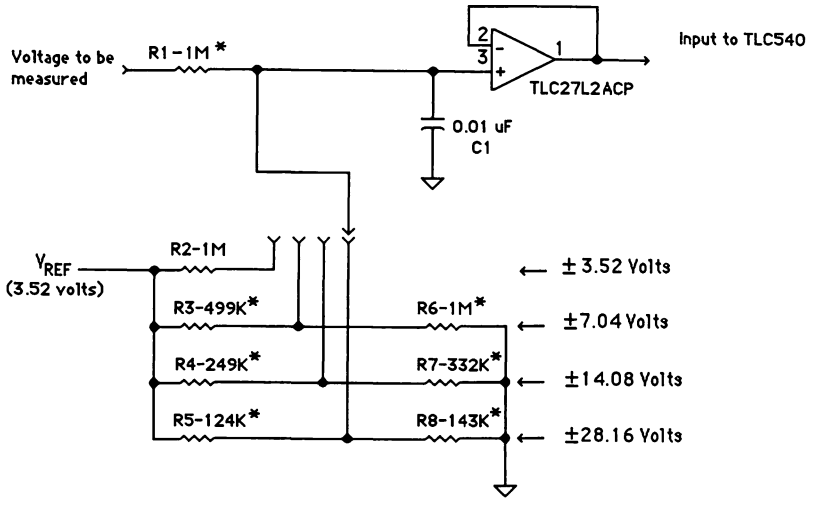
−3.52 to +3.52

−7.04 to +7.04

−14.08 to +14.08

−28.16 to +28.16

In this example each range is neatly symmetrical with respect to zero volts, but you can set the high and low point of each range independently of each other and of the



*Note: all resistors are 1%

Figure C-25. A digital voltmeter

other ranges. For example, one range could be from −5 to +10 volts and the next could be from −20 to +7. The only restrictions on the range are that the most positive limit of the negative end of the range is minus the value of V_{REF}, −3.52 in this case, and the most negative limit of the positive end of the range is V_{REF}. The resistors labeled R2 through R5 in Figure C-25 control the negative extent of the range, and resistors R6 through R7 the positive. The value of each resistor in any particular range is determined by a formula. The formula that determines the value of the resistor for the negative extent is:

$$R = R1 \times V_{REF} / 1 - V_{NEGLIM} 1$$

and the formula that determines the value of the resistor for the positive extent is:

$$R = R1 \times V_{REF} / (+V_{POSLIM} - V_{REF})$$

It is important to use precision resistors (1%) in this circuit, so after using the formulas to compute the desired resistance, you will have to substitute the value of the nearest available precision resistor. This will provide an exact reading of the voltage over a range very close to the one you want. You can, of course, add more ranges to the meter by adding more resistors.

One point to bear in mind when establishing the measurement range is that the resolution is inversely related to the range: If you double the range, you cut the resolution in half. (Resolution is equal to the range divided by the number of output steps from the A/D converter—256 for the HBC-1.) With the configuration shown in the voltmeter schematic, the resolution at the lowest range is 0.0275 volts per step; at the highest range it is 0.22 volts per step. You could easily double the resolution of one (or all) of the inputs by removing the low-extent resistor for that input (R2, 3, 4, or 5). You would sacrifice the bipolarity of that input but in so doing the resolution would be doubled (the range cut in half). If this is the only TLC540 channel being monitored, you could reverse the voltmeter leads to read a negative voltage. If more than one channel is being monitored, reversing the leads on one would mess up the others.

In conclusion

As I said in Chapter 18, the HBC-1 was conceived as a method to demonstrate a simple BASIC program: From that point of view the project probably got a little out of hand. We ended up with a full-blown analog-to-digital conversion system that is comparable to or better than most commercially available units costing 10 to 20 times as much. Building the HBC-1 was a lot of fun, but using it is even more fun. With a little imagination, you will come up with a lot of ways to put it to use. For example, one person has already suggested a method of building a do-it-yourself Thunderscanner, and another is creating a navigation system for his boat. If you come up with a particularly interesting idea, or if you write a program that does something new and different, and you would like to share it with others, send it to me care of Microsoft Press, and I will try to find a way to spread the word.

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Steve Lambert

Steve Lambert, a native of Seattle, Washington, has worn many hats, including those of high-rigger, house painter, locksmith, and journeyman electrician. An interest in Seattle's early architecture led him to publish a biography of designer/builder Fred Anhalt.

Steve's fascination with personal computers has led him to investigate many of their practical uses. He has written about computers for *High Technology* and *Computing for Business/Interface Age* magazines, and is a contributor to *Macworld, Time-Life Access,* and *PC World* magazines. He is also the author of *Presentation Graphics on the Apple Macintosh,* and *Online: A Guide to America's Leading Information Services,* published by Microsoft Press.

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were processed and formatted using Microsoft Word.

Cover design by Ted Mader and Associates. Interior text design by John D. Berry. The high-resolution screen displays were created on the Apple Macintosh and printed on the Hewlett-Packard LaserJet.

Text composition by Microsoft Press in New Caledonia with display in Helvetica, using the CCI composition system and the Mergenthaler Linotron 202 digital phototypesetter.

OTHER TITLES FROM MICROSOFT PRESS

THE APPLE MACINTOSH BOOK, *2nd edition* *Cary Lu* *$19.95*

EXCEL IN BUSINESS *Douglas Cobb and the Cobb Group* *$22.95*
Number-Crunching Power on the Apple Macintosh

THE PRINTED WORD *David A. Kater and Richard L. Kater* *$17.95*
Professional Word Processing with Microsoft Word on the Apple Macintosh

MACWORK MACPLAY *Lon Poole* *$18.95*
Creative Ideas for Fun and Profit on Your Apple Macintosh

PRESENTATION GRAPHICS ON THE APPLE MACINTOSH *Steve Lambert* *$18.95*
How to Use Microsoft Chart to Create Dazzling Graphics for Professional and Corporate Applications

MICROSOFT MACINATIONS *The Waite Group, Mitchell Waite, Robert Lafore, and Ira Lansing* *$19.95*
An Introduction to Microsoft BASIC for the Apple Macintosh

MACINTOSH MIDNIGHT MADNESS *The Waite Group, Mitchell Waite, Dan Putterman, Don Urquhart, and Chuck Blanchard* *$18.95*
Utilities, Games, and Other Grand Diversions in Microsoft BASIC for the Apple Macintosh

MICROSOFT MULTIPLAN: OF MICE AND MENUS *The Waite Group, Bill Bono, and Ken Kalkis* *$16.95*
Models for Managing Your Business with the Apple Macintosh

COMMAND PERFORMANCE: MULTIPLAN ON THE APPLE MACINTOSH *Eddie Adamis* *$19.95*
The Microsoft Desktop Dictionary and Cross-Reference Guide

INSIDE MACPAINT *Jeffrey S. Young* *Introduction by Bill Atkinson, creator of MacPaint* *$18.95*
Sailing Through the Sea of FatBits on a Single-Pixel Raft

ONLINE *Steve Lambert* *$19.95*
A Guide to America's Leading Information Services

SILICON VALLEY GUIDE TO FINANCIAL SUCCESS IN SOFTWARE *Daniel Remer, Paul Remer, and Robert Dunaway* *$19.95*

OUT OF THE INNER CIRCLE *"The Cracker" (Bill Landreth)* *$9.95 softcover* *$19.95 hardcover*
A Hacker's Guide to Computer Security

A MUCH, MUCH BETTER WORLD *Eldon Dedini* *$6.95*

Available wherever fine books are sold. For a catalog listing all our titles write to:

Marketing Department • Microsoft Press • 10700 Northup Way • Box 97200 • Bellevue, WA 98009

CREATIVE PROGRAMMING IN MICROSOFT® BASIC joins the growing Microsoft Press library of high-quality, critically-acclaimed books on the Macintosh.

Creative Programming in

MICROSOFT® BASIC

BY STEVE LAMBERT

Here—for intermediate programmers—is a superior selection of original programs that explores the incredible possibilities of programming with Microsoft BASIC on the Macintosh. If you're already familiar with BASIC on another machine, you'll discover all the unique characteristics of the Macintosh that are accessible through BASIC. Lambert's example-driven approach includes 17 BASIC utilities and games: a feature-rich communications program, a program that transfers images from BASIC to another program, and a unique and inventive analog to digital converter—worth over $600—that can be built for under $75. Steve Lambert is the author of the popular Microsoft Press titles *Presentation Graphics on the Apple Macintosh* and *Online*.

USA	**$18.95**
U.K.	£15.95
AUST.	$28.95 (recommended)
CAN.	$28.95

MICROSOFT PRESS

ISBN 0-914845-57-