MACINTOSH MIDNIGHT MADNESS
Utilities, Games & Other Grand Diversions in Microsoft BASIC for the Apple Macintosh

THE WAITE GROUP: Mitchell Waite • Dan Putterman • Don Urquhart • Chuck Blanchard
Utilities, Games & Other Grand Diversions in Microsoft BASIC for the Apple Macintosh
Contents

Acknowledgments ix
Introduction xi

I. UTILITIES
MacInterface 3
MacGrid 31
MacGraph 51
MacCursor 87
MacConvert 119
MacAnimate 149

II. GAMES
MacDart 191
MacBandit 209
MacMouse 233
MacBall 249
MacGurkha 273
MacSpace 297

III. DIVERSIONS
MacBuild 325
MacOrgan 345
MacCompose 359
MacFinance 381
MacMeasure 403
ACKNOWLEDGMENTS

When microcomputers first came on the scene, one of the first things that they were programmed to do was flash a patterned sequence of lights. Early demonstration programs, in fact, were games that were toggled into the machine through a series of switches. We owe our thanks to the early pioneers—to people like Bill Gates and Paul Allen who successfully got BASIC to run on an Altair, and the myriad nameless programmers, from electrical engineers to piano players—for the tradition that this book follows.

Creating games and utilities—sharing them, tearing them apart, and putting them back together—was how many of us first learned to program with BASIC. The development of this book followed that same spirit. Mitchell Waite, Chuck Blanchard, Harry Henderson, Dan Putterman, and Don Urquhart of The Waite Group development team, and Chris Mattheus, Barry Preppernau, and David Laraway of Microsoft Press spent many hours devising, designing, refining, programming, testing, and describing the games, utilities, and other diversions you will find in this book.

We would like to acknowledge the help and encouragement that we received from so many people throughout this project. We would like to thank in particular Sally Oberlin of Microsoft Press for seeing this project through from the beginning; Lisa Yount, for her critique of the text through the rewrite phases; Mary Johnson of The Waite Group for coordinating the flow of materials among the participants; Bob Johnson for his cursor designs; and the unsung heroes and heroines in the realm of production—the people who turned the words, programs, and images into this book. Thank you all.
Introduction

With Midnight Madness we invite you to participate in a programming tour de force. The Microsoft BASIC version 2 series for the Macintosh makes it easier than ever before to create powerful yet flexible programs that take full advantage of the Mac’s unique user interface and graphics capabilities. Running and studying the 17 programs in this book will not only
enhance your skills as a programmer, it will also give you hours of pro-
gramming pleasure.

As the book's subtitle suggests, the programs in *Midnight Madness*
are arranged in these three categories: utilities, games, and “other grand
diversions.”

*Utilities* are tools that help you do a job easier or faster. Some of
our utilities provide resources for your programming; others help orga-
nize information. When you learn how a tool works, you can modify it to
fit your needs or to add new capabilities.

*Games* are toys. They’re fun to play with and great to share with
friends of all ages. There’s more to them than that, though: Programming
tricks found in these games can be used for many other sorts of programs
as well. You can also make the games harder or more sophisticated, and
add strategic as well as arcade features.

*Diversions* combine the usefulness of utilities with the playful ex-
ploration of games. They are “edutainment”—a blend of education and
entertainment—and represent the growing variety of creative and unique
applications that Microsoft BASIC and the Macintosh make possible.
We’re betting you’ll find many others!

**Presenting the programs**

Now it’s time to introduce our all-star cast of programs. We’ll briefly de-
scribe what each of the programs in the three categories does. The moons
next to the program names indicate the general level of difficulty of the
programming concepts involved:

- is rather easy—the program code is short and straightforward,
  and the algorithms involved are not too complex;
- is intermediate—you will find longer code and more involved
  structures, and the algorithms and techniques will help you learn
  some advanced concepts you will need in building programs of your own;
- is somewhat difficult—but don’t let this scare you—these pro-
  grams will become the stars in your collection, and will teach you
  tricks and techniques you may have thought weren’t possible before.
Utilities

*MacInterface*, the first program in the book, lets you move graphics back and forth between BASIC and applications like MacPaint via the Macintosh Clipboard. You can use the application of your choice to create the images, and then animate them with BASIC, or let BASIC create complex shapes to use with other applications. You can paste images to the Scrapbook and read them into other programs.

*MacGrid* makes it a snap to create any kind of graph paper you want, varying the line intervals, heavy lines, and the overall size of the page. Great for math and science projects!

*MacGraph* turns your data into bar, line, or pie charts, quickly and easily, with no complex formats or commands to learn. You will be able to plot business trends, or the results from your lab experiments, or your success with your diet.

*MacCursor* is a powerful utility that lets you create custom mouse pointers, complete with “mask” and “hot spot.” Now your mouse can wield a pencil or control a starship!

*MacConvert* automates the conversion of programs from Microsoft Macintosh BASIC 1.0 to the new version 2. It also aids conversion from other dialects of Microsoft BASIC. You will be able to have your old favorite programs running on the Mac in no time.

*MacAnimate* explores the world of animation. It provides you with an editor for creating and modifying a series of images much like the frames of a strip of movie film, and also includes an animator that lets you control their motion. Now Mickey won’t be the only mouse who can move across a screen.

Games

*MacDart* has a dart and target balloon that move at different speeds each time you throw. What’s more, it even includes the effect of gravity on your throw! You will be able to tune up your hand-eye coordination, and you don’t have to worry about hitting passing strangers. Timing is everything with this game.
MacBandit provides a Las Vegas-style “one-armed bandit.” You drop your coins in the slot, the wheels spin, and with a bit of luck the Macintosh is stacking your winnings in neat little piles. This one is great for parties!

MacMouse features an animated mouse race complete with music. It’s a good way to introduce young children to the Mac.

MacBall is a “break-out” game where placing a shot just right will reward you with a “chain reaction” of scoring. The algorithms show you how to re-create the rebound of the ball off the sides of the screen—a technique that will have you devising your own variations in no time at all.

MacGurkha is named for the intrepid paratroopers from Nepal. The challenge of this game is to guide your trooper past the barrage balloons into the jump zone. Not only will you learn techniques for moving objects, but you will also learn how to make animated objects interact with their environment.

MacSpace brings real arcade excitement to the Mac. This game integrates many of the features of the utility programs. It’s all here: animation, a custom pointer, and complex graphics. Just about the only thing missing is that pointy-eared executive officer. Can you destroy the dreaded Klaxxon star cruisers before they get too close?

MacBuild is a “drawing construction set.” With it, you can assemble designs from our “prefab” shapes, and also design your own custom shapes. You will be able to design all sorts of interesting figures once you’ve entered this program. And if you don’t like what you’ve built, you can always try building another.

MacOrgan explores the Mac’s musical capabilities and turns your keyboard into an organ with two octaves and three chords. In no time at all, you will be entertaining your friends with tuneful little ditties that show off the voices of the mighty Macintosh.

MacCompose lets you express your musical ideas by placing notes on a staff and playing the resulting melody. While MacCompose is a more complex program than MacOrgan, it will give you hours of enjoyment as you explore the fundamentals of music composition.
Companion disk to *Macintosh Midnight Madness*

To save you hours of typing in the programs—and the additional hours of finding those inevitable typos—Microsoft Press has created a special program disk for *Macintosh Midnight Madness*. The disk contains all 17 projects included in the book, plus additional programs containing animation sequences and cursor patterns.

This disk is only available directly from Microsoft Press. To order, use the special bind-in card at the back of the book. If the card has already been used, send $19.95 (plus $1.00 per disk for domestic postage and handling charges, $2.00 for foreign orders) to: Microsoft Press, Box 97200, 10700 Northup Way, Bellevue, WA 98009. Payment must be in U.S. funds, California residents must add 6.5% sales tax, and Washington residents 8.1%. You may pay by check or money order (payable to Microsoft Press) or by American Express, VISA or MasterCard; please include both your credit card number and expiration date. Allow 4 weeks for delivery.
MacFinance answers "questions of interest" about your money—whether you're borrowing or investing it. It shows you some of the really practical things that you can do with the Macintosh. You might want to incorporate some of the algorithms that you learn here in other number-oriented programs.

MacMeasure answers a different kind of question: Is it a long way to Tipperary...or to Cleveland? After calibrating your mouse against a map's scale, you can then use the mouse to trace between any two points and learn the distance.

How to use this book

Each chapter in Midnight Madness starts with a description of how to use the program, including examples of actual screens. You can type in the program from the source code at the end of the chapter first and then learn how to use it from the description, or you can wait until you've worked through the chapter before entering and running the program—whichever you prefer.

Following the "how to use it" discussion is the "how the program works" section. Here's where the programming concepts and tricks are revealed in all their glory. This section starts with a brief overview of the program's operation and the concepts it features. While most programs embody several concepts, there is usually a major theme for a given chapter: MacAnimate focuses on moving objects on the screen, MacFinance features window-refreshing techniques, MacOrgan and MacCompose employ the Mac's sound and music features, and so on.

Next comes a complete pseudocode description of the program. What's pseudocode? It's like an outline you might make before doing a class paper. Pseudocode provides a high-level, English-like description of program operations in such a way that you can study the methods (algorithms) that are used by the program and see its structure, without getting bogged down in detail. For example, you might run into something like this:

```
WHILE there are images in the graphics file
    read an image into the graphics array
    add one to the image count
WEND
```

This tells you what the program is doing without your having to worry about the names of the variables used or the exact syntax of the statements that perform the operations.
Following the pseudocode outline is the in-depth, “blow-by-blow” description of how the program works. This description is keyed to the program-code listing at the end of the chapter, and uses the names of the subroutines plus some special colored labels we’ve added to the code for easy reference. These labels serve to break up long sections of code (such as the initialization part of the program) into logically related pieces that can be discussed one at a time. (Unlike the subroutine and other labels followed by colons, these special labels aren't part of the code and shouldn't be typed in with the program itself.)

In the discussion you will sometimes find references to other chapters in Midnight Madness where a given technique is discussed at greater length. You'll also sometimes find references to the companion book Microsoft Macinations (Microsoft Press, 1985). This book complements Midnight Madness by providing complete discussions and additional examples of most of the techniques used in our programs. In fact, if you are new to Microsoft BASIC for the Macintosh, or new to programming in general, we'd like to recommend that you read Macinations first or in conjunction with this book. Advanced programmers will also benefit from its comprehensive treatment of BASIC features for the Macintosh.

Finally, each chapter concludes with a discussion of possible modifications for the program and suggestions on how to go about implementing them. This is your chance to add fiendish features to the games, extend the functions of your favorite utility, or make a diversion a bit more diverting. In addition to extending the programs in this book, the tools and the concepts you'll learn will enable you to professionally polish your own programming projects. We think you'll end up agreeing with us that with Microsoft BASIC, the marvelous Mac, and this book, programming has never been so much fun!
I. UTILITIES

- MacInterface
- MacGrid
- MacGraph
- MacCursor
- MacConvert
- MacAnimate
Integrated software combining two or more application programs is all the rage these days, and for good reason: In theory, integrated software is supposed to make it easy to move information from one application to another. For example, you might create a business graph with a statistical program and then insert the graph into a document that you’re writing on your
word processor. Programs that claim such integration are widely available, but many of them have limitations. Some can't handle text and graphics together, while others can move information in one direction but not in another. If a particular function isn't available in a certain integrated software package, you may find it hard to set up communication between that program and another one that can do the job. In any case, you're limited to whatever integration the author of a particular software package chooses to provide.

The Macintosh, on the other hand, provides system-level integration. This means that the operating system is designed so that any application can easily access a standard set of tools such as the Clipboard and the desk accessories (Note Pad, Scrapbook, Calculator, and so on). Thus, instead of being "in" an application such as a word processor and having to use special commands to send information to another application, you're seated in front of a "desk" as you do your work. The desk contains a variety of applications that you use as needed, just as you might type a letter on a typewriter and then reach for a set of colored pencils to draw an illustration to go with the letter. The desk is always there to serve as your "home base" and to provide standard facilities as you move from one application to another.

You have access to many of these handy Macintosh desk facilities when you're developing programs in Microsoft BASIC, but if you deal much with graphics, there's another capability that you probably wish you had. As you know if you've studied the code for our games, BASIC is great for animating pictures and manipulating them in other ways; a lot of nice things can be done with a few powerful commands. On the other hand, BASIC isn't very good for creating pictures. If you want to draw a realistic-looking spaceship, for example, you have to get some graph paper and try to draw the image as you want it to appear on the screen. Then you have to figure out what BASIC statements are needed to produce this image, and you probably have to go through several cycles of debugging before you get it to come out right.

Your Macintosh probably came with an excellent tool for creating pictures in a natural way, without a lick of programming: MacPaint. Of course, to animate pictures or make them interact with others, you do need to program, which is where BASIC comes in. Clearly it would be nice to use something like MacPaint to create pictures and a language like BASIC to manipulate them. Alas, this is easier said than done.
MacPaint lets you create a picture and put it on the Clipboard for temporary storage, and from there into the Scrapbook, if you wish to keep it as a more permanent file. Unfortunately, BASIC doesn’t come with any built-in way of making a picture on the Clipboard accessible to your programs. Don’t despair, though: MacInterface will come to your rescue! This utility lets you take pictures from the Clipboard that were created with MacPaint or some other application and save them in files that can be used by your BASIC programs. Many of the games in this book, as well as programs using similar code that you write yourself, can read files created with MacInterface and use the pictures they contain. Designing games and other programs that use graphics becomes much easier and more fun once you have this utility, since you can build up graphics library files and extract pictures from them later to use anywhere you wish. MacInterface provides a powerful new link between Microsoft BASIC and the rest of the Macintosh world.

How to use MacInterface

When you run MacInterface, a new menu appears to the right of the BASIC Windows menu:

If you choose Instructions from the menu, you’ll be given the window shown in Figure 1 on the following page, which briefly summarizes MacInterface’s features.

Let’s step through an example to show a practical use of MacInterface. We’ll create two new star-cruiser pictures with MacPaint and then use MacInterface to make them available for use in our MacSpace game. One is a new enemy ship, the Robiton, which represents a mechanical intelligence even more implacable than the vicious Klaxxons who terrorize
With MacInterface you can transfer pictures from the Scrapbook to a file, and move a picture file into the Scrapbook. Pictures are moved between file and Scrapbook through the Clipboard in a two-step process. Use the normal Edit menu to move pictures between the Clipboard and Scrapbook, and these menu commands to move between Clipboard and file:

- **Save Clip As:** Asks for a file name and puts the contents of the Clipboard into that file.
- **Append Clip:** Asks for a file name and appends the picture in the Clipboard to the end of that file.
- **Append Clip To:** Appends a picture in the Clipboard to the file currently opened by MacInterface.
- **Read from File:** Loads a selected picture file into memory. Use the Next button to leaf through the pictures in the file and the Copy button to copy individual pictures into the Clipboard.

**FIGURE 1.** The instruction window available from the MacInterface menu.

the galaxy in MacSpace. The other picture is a new player ship for the peace-loving defenders of truth and justice in that program. (If you haven't played MacSpace or looked over that chapter, you might want to do so now so you'll know how our pictures might be used.)

If you've studied the code for MacSpace or some of the other games, you'll remember that these programs are set up to read a graphics file if it is present on the disk. The name of the graphics file consists of the program's name followed by a period and the word *graphics*, so the graphics file we'll create for MacSpace will be called *MacSpace.graphics*.

Before we get started, there's one unfortunate problem with moving files between MacPaint, MacInterface, and BASIC programs that we'd better warn you about: There usually isn't room for everything on one disk, which means that you will have to do a fair amount of disk-swapping if you have only one drive. One Macintosh feature not advertised by Apple is that it's a great exercise machine for the arms; there always seems to be one more disk swap to make, right down to the Mac's demand that you reinsert the disk with the smear of peanut butter on it that you haven't seen since last Tuesday. This is a cross that those of us with single-drive systems simply have to bear.
This problem can be eased somewhat by copying MacPaint, Micro-
soft BASIC, MacInterface, and a couple of the games or other programs
you're going to be experimenting with to one disk. Since copying the Sys-
tem files to this same disk to make it a startup disk wouldn't leave much
space, however, you probably will have to start up with another disk and
then switch to your MacInterface "working disk." That will still leave more
disk-swapping than you'd like, but it should be fairly tolerable. If you're
lucky enough to have two disk drives, life is much easier. Or, you could use
the Font/DA Mover, which comes with the Finder 4.1 system disk, to re-
duce the size of the system file by removing fonts and desk accessories.

In the unlikely event that you haven't yet used MacPaint, we rec-
ommend that you take some time to play with it and familiarize yourself
with its capabilities now. Since games usually require small but detailed
pictures, you should probably use MacPaint's FatBits option (chosen from
the Goodies menu) when creating these pictures. We used FatBits to cre-
ate the star cruisers shown in Figures 2 and 3.

FIGURE 2. A FatBits view in MacPaint of the Robiton Star Cruiser to be used in
MacSpace
We created the Robiton star cruiser first because MacSpace loads the “enemy” cruiser picture before it loads the picture of the player’s starship. After creating the Robiton picture, we then copied it to the Clipboard. This is done by selecting the picture with the Marquee or Lasso and then choosing Copy from the MacPaint Edit menu.

Figure 4 shows how our Macintosh desktop looked after we copied the Robiton picture to the Clipboard and quit from MacPaint. We obtained the display of the Clipboard by choosing Show Clipboard from the Macintosh Edit menu—a useful way to check the contents of the Clipboard. If you’re working with several pictures, you can save them in the Scrapbook and copy them to and from the Clipboard using the Edit menu’s Copy and Paste functions.
Next, we ran MacInterface, which presented us with this dialog box:

![Macintosh desktop showing MacPaint and MacSpace]

**FIGURE 4.** *The Macintosh desktop after creating the pictures in MacPaint that will be used in the MacSpace game*

We clicked the Yes button, which opened the window shown in Figure 5 on the next page displaying our Robiton picture. Then we pulled down the MacInterface menu, and chose the Save Clip As option. We called the file *MacSpace.graphics*, and the picture was saved in it.
We followed the same steps with the picture of the player ship, except that, after we clicked the Yes button in the dialog box, we appended it to the already-created MacSpace.graphics file by choosing the Append Clip option from the MacInterface menu.

After both pictures were placed in MacSpace.graphics, we verified that everything had taken place as expected by choosing Read from File from the MacInterface menu, which resulted in the screen display shown in Figure 6. First, the Robiton picture was displayed, then we clicked the Next button to see the picture of the player's starship.

The final test is running MacSpace with the new pictures. Assuming that the MacSpace and MacSpace.graphics files are on the same disk (MacSpace won't ask for another disk if it doesn't find the graphics file), the new pictures should appear on the title screen as shown in Figure 7. If you remove the MacSpace.graphics file from the disk containing MacSpace, the game will run with its own built-in pictures.

We recommend that you practice using MacPaint and MacInterface together by creating two ship pictures of your own for MacSpace. Then try it with another game. Any program that has code that reads a file with the extension .graphics will read a file created by MacInterface. Make
FIGURE 6. The two MacInterface windows

FIGURE 7. MacSpace title screen displaying pictures saved in a MacInterface file.
sure the file you create has the exact name specified in the code and that it's on the same disk as the program that will use it.

Of course, you can do much more with MacInterface than create new pictures for your games. Your own BASIC programs can also read and use picture files created with MacPaint and MacInterface. And you aren't limited to MacPaint as a source of pictures: The output of any application that can place something on the Clipboard can be used by MacInterface. For example, you can take pictures from any of the commercially available libraries of pictures for the Mac. Just select a picture from the library and put a copy on the Clipboard, then use MacInterface to file the picture for use with your BASIC programs.

You can also send information in the other direction— from BASIC to MacPaint, for example. You could write a BASIC program to generate a pie chart and place the chart on the Clipboard. You could then run MacPaint and paste the picture from the Clipboard into a MacPaint document so you could use those neat little brushes and pens to embellish the graph. Finally, you could copy the enhanced graph back to the Clipboard and then use MacWrite to copy it into a document.

Using the Scrapbook with MacInterface

It's probably occurred to you that we had to go back and forth between MacPaint and BASIC twice in order to get both of our spaceships into MacInterface. We did this so the necessary steps could be clearly explained. However, when you want to bring two or more images from MacPaint or another application into MacInterface, you can bypass several steps by saving them in the Scrapbook as you create them if both the application and BASIC are on the same disk. For example, here's what you would do with the two spaceships in MacPaint if both MacPaint and BASIC were on the same disk:

1. Select the Robiton ship with the Marquee or Lasso.
2. Choose Copy from the Edit menu.
3. Choose Scrapbook from the Apple menu.
4. Choose Paste from the Edit menu.
5. Close the Scrapbook.
6. Select the player's ship with the Marquee or Lasso.
7. Choose Copy from the Edit menu, open the Scrapbook, and choose Paste from the Edit menu.

At this point both of the ships would be stored in the Scrapbook, and you would quit MacPaint and start up MacInterface. To get the ships into a
MacInterface file, you would:

1. Open the Scrapbook and scroll to the Robiton image.
2. Choose Copy from the BASIC Edit menu.
3. Close the Scrapbook.
4. Choose Save Clip As from the MacInterface menu and enter a file name in the dialog box.
5. Open the Scrapbook and scroll to the player's ship.
6. Choose Copy from the BASIC Edit menu.
7. Close the Scrapbook.
8. Choose Append Clip from the MacInterface file menu.

By using the Scrapbook like this you would achieve the same results, but by staying in one application at a time you would spend much less time looking at the little wristwatch while the Macintosh quit one application and loaded the other. Figure 8 summarizes the interaction of MacInterface, MacPaint, the Scrapbook and Clipboard, and BASIC programs.

FIGURE 8. The movement of images using the Clipboard and MacInterface
How MacInterface works

How can BASIC programs use the Clipboard? The brief answer is that the Macintosh treats the Clipboard just about like any other file, so the BASIC statements used for creating or reading disk files will also work with the Clipboard. You'll learn a more detailed answer as we study the MacInterface program outline shown in Figure 9.

Initialization
- initialize variables and arrays
- define functions for picture-string and file-name handling
- install MacInterface menu and menu event-trapping
- ask if a picture is in the Clipboard (ClipHere)
  - enable dialog event-trapping and create Output window

Main Program
- establish endless loop:
  - interrupted by menu and dialog events,
  - and ended by clicking in close box of Clipboard (CloseBox)

DialogEvents
- DialogEvent: find number of event, item affected,
  and branch to appropriate subroutine:
  - BtnClick: a button was clicked
    - get number of button and pertinent window
    - if picture file window:
      - either display next image in file, (DisplayNext)
      - or copy image to Clipboard (CopyToClip)
  - ActiveWindow: button was clicked in inactive window
    - make window active
  - CloseBox: close box clicked in active window
    - if Clipboard window, end program
    - if other, enable menu items
  - RegenWindow: window needs refreshing
    - depending on window, regenerate contents of:
      - Clipboard (ShowClip)
      - image file "pseudo-scrapbook" (RefreshRead)
      - instruction window (PrintInst)

Menu Events
- MenuEvent: disable menu, branch to subroutine:
  - SaveClip: save clipboard to a new file
    - if clipboard empty, display message (ClipEmpty)
    - get filename and save clipboard to file (ClipToFile)

FIGURE 9. Program outline
AppClip: append Clipboard to existing file
if clipboard empty, display message (ClipEmpty)
get filename and append clipboard to file (AppToFile)

AppToFile: append Clipboard to file currently opened
save Clipboard contents to file (ClipToFile)

ReadPics: read images from file
get filename of graphics file
if no filename entered, exit
open file
WHILE there are more images
read each image into picture array
WEND
create window for "pseudo-scrapbook"
create buttons in window
for advancing to next picture
for copying picture to Clipboard

Inst: show instructions
create Output window for instructions; opening window
generates refresh dialog event (RegenWindow)

BasQuit: quit to BASIC
reset menu, close all windows (CleanUp)

DeskQuit: quit to Mac Desktop
reset menu, close all windows (CleanUp)

Subroutines
ClipHere:
open window
ask user if Clipboard has picture
if no, clear Clipboard for use
close window

CleanUp: reset menu, close all windows

ShowClip:
open Clipboard; if empty, return
read image and store in picture string
display in Clip window (DisplayPict)

ClipToFile:
open Clipboard; if empty, show message (ClipEmpty)
read image into picture string
send string to opened file
close files and update menu with filename

ClipEmpty: show "empty Clipboard" message

Figure 9. Program outline (continued)
CopyToClip:
- disable buttons in "pseudo-scrapbook" while copying
- open Clipboard for output and store current image
- switch to Clip window and display image (ShowClip)

DisplayNext:
- increment current picture counter
- display image (RefreshRead)

RefreshRead:
- display image (DisplayPict)
- update image number in window

PrintInst: print instructions

Subprograms
DisplayPict:
- get current window dimensions
- calculate current picture dimensions
- display image in center of window if possible, else just top left corner

Figure 9. Program outline (continued)

Constants and initialization

The constants defined in [Init 1] simply define the Output window. We'll discuss the function defined in [Init 2] later, when it's used. The GOSUB CleanUp [SR 2] statement gets rid of any Output windows that might be left over from a previously run program and frees up the memory that had been used to maintain them. This is an example of what could be called "defensive programming." We would like to assume that all programs have the courtesy to leave the Macintosh environment the way they found it, but it's safer not to make that assumption. Since memory is tight on a 128K Mac, it's a good idea to have a program reclaim memory that may be tied up in windows or other facilities left over from previous programs.

The MacInterface menu is created in [Init 3]. Since the program is menu-driven, the seven main functions are accessed via a pull-down menu. (The program produces dialog boxes with buttons when it needs more information to carry out a particular function.) You'll note that the program line for the sixth menu item has "-" for a title; this causes a line to be drawn, dividing the menu selections into two parts. The first five items call subroutines that actually do the main work, while the last two contain the quit functions. Breaking a menu into groups of related functions like
this gives the user a little more help in seeing how things are organized, and it looks nicer, too.

Next, the ON MENU GOSUB statement enables menu event-handling. When a menu event occurs, MenuEvent [ME 0] gets the number of the item selected from the MENU(1) function and dims the MacInterface menu with MENU 6,0,0. This shows the user that a function is in progress and that further menu selections can't be made until the currently selected function is completed. The ON item GOSUB statement sends control to the subroutine in charge of the function selected. (Note the extra comma after Inst in the ON item GOSUB statement; it's needed for the sixth menu event, the "dummy" event that was used to print the line across the menu. If someone tries selecting this event, no branching takes place, since that menu item was never enabled.

In [Init 4], the GOSUB statement that calls ClipHere [SR 1] first creates a window, then sets up two buttons, Yes and No, for answering the question "Does the Clipboard have a picture in it?" If the answer is no, we will not be using the Clipboard for input, so the subroutine then opens the Clipboard for output.

Next, dialog events are enabled with the ON DIALOG GOSUB statement. Following this statement, the window to display the Clipboard is set up and displayed.

When the window is opened, a dialog event 5 is generated. Since all dialog events are handled in DialogEvent [DE 0], control temporarily goes to RegenWindow [DE 4], where ShowClip [SR 3] is encountered. ShowClip opens the Clipboard, reads its contents, and uses the subprogram DisplayPict [SP 1] to put the image on the Clipboard, if there is one in Window 1. The main program [Main] is an endless loop that is interrupted when a menu or dialog event occurs and resumes when processing is completed. The program is exited not by the loop terminating, but by a particular menu event: a Quit selection, which exits via the BasQuit [ME 6] or DeskQuit [ME 7] subroutines. Both of these subroutines call the CleanUp subroutine [SR 2]. The loop and therefore the program also ends if the close box of the Clipboard window is clicked.

When a dialog event occurs, DialogEvent [DE 0] gets the number of the item selected from the DIALOG(0) function and sets the variable dilog equal to it. The subroutine then makes choices based on the value of dilog. If dilog is not equal to zero, then the variable dinfo is set equal to DIALOG(dilog). On the next line, control is branched to a subroutine based on the number of the dialog event that happened—1, 3, or 4 (this program doesn't use edit fields, so a dialog event 2 can't be generated).
If a button was clicked, *dilog* is equal to 1, and control transfers to 
*BtnClick* [DE 1]. Here, some checks are made to make sure the buttons 
clicked were in Window 2, where images from our graphics file are displayed. 
Depending on the button clicked, we either want to see the next 
image (*DisplayNext* [SR 7]) or copy the current image to the Clipboard 
(*CopyToClip* [SR 6]).

A dialog event 3 occurs when the mouse button is clicked with the 
pointer in an inactive window, so *ActiveWindow* [DE 2] changes the cur-
cent window to the desired one. If *dilog* equals 4, then the close box of the 
active window was clicked. If the close box is that of the Clipboard win-
dow (2), then *finished* is set to *true* and the main loop terminates, ending 
the program. Otherwise, the user clicked on the other window and the two 
pertinent menu items are enabled.

Finally, if a window is resized, *dilog* equals 5 and *RegenWindow* 
[DE 4] is called, which recreates the contents of the window specified by 
*dlnfo*, using *ShowClip* [SR 3], *RefreshRead* [SR 8], or *PrintInst* [SR 9].

As we look at each of the functions that *MacInterface* performs, 
we will show you how this whole dialog event area works.

### The Clipboard item functions

While there are seven main functions in *MacInterface*, they can be consid-
ered in groups. There are those functions that relate to saving Clipboard 
items either to new files or appending them to existing files, a function 
which allows you to “leaf” through a file image by image, a function which 
lets you read the instructions, and then the standard exit functions to 
*BASIC* or to the Macintosh desktop. Let’s take a closer look at the Clip-
board functions now.

The first three menu selections basically show you the current 
contents of the Clipboard and ask if they are to be saved as a new file or 
appended to an existing one.

#### Save Clip As

The first Clipboard menu function, *SaveClip* [ME 1] saves the 
contents of the Clipboard to a new file. It does this by first opening the file 
“CLIP:PICTURE” for input. The Clipboard, or “CLIP:”, is a “predefined” 
device, analogous to a disk file. The “PICTURE” tells *BASIC* that the infor-
mation on the Clipboard is encoded graphics information rather than text.
Of course, there may be nothing on the Clipboard to read. Therefore, the length of the Clipboard file is checked with the LOF function: If the length of the file is zero bytes, there's nothing on the Clipboard. In that case, control passes to ClipEmpty [SR 5], where DIALOG is set to OFF and the message "Sorry, the Clipboard is empty" is displayed. An OK button is drawn, and the program sits in a WHILE...WEND loop waiting for the user to press the button. There isn't really too much you can do with an empty Clipboard.

If the Clipboard isn't empty, LOF is greater than zero and the program requests a name for the file that will save the Clipboard image. The FILE$ function is used to display a standard file-selection dialog box, and the parameter 0 indicates that this is to be a new file and that a file name needs to be obtained. The prompt "Save as" is displayed, an insertion point and the usual editing facilities are provided, and an option to eject the disk and insert another is also automatically provided by FILE$. You can see that this dialog box saves the programmer a lot of work and gives the user something familiar from many other Macintosh applications.

After a name is entered via the dialog box, the name is saved in the variable appFile$. This variable is the name of the file to be used in the append subroutines. After the file in appFile$ is opened for output, control passes to ClipToFile [SR 4]. This subroutine first opens "CLIP:PICTURE" for input, then checks to see if there is anything on the Clipboard. If there isn't, the control jumps down to the label EmptyClip, which then closes the "files," and goes to ClipEmpty [SR 5], which we described earlier. If there is something on the Clipboard, the Clipboard contents are read. The actual data is read in by the INPUT$ function using the length of the file returned by LOF and assigned to the string clip$. For reading a chunk of data that doesn't need to be divided into fields, the combination of the LOF and INPUT$ functions gives Microsoft BASIC a real advantage over BASICS that lack these functions and have to use a loop to read the data.

Once the Clipboard picture has been read and stored in clip$, control jumps to the section labeled AClipEnd and gives the third item in the MacInterface menu the name of the file stored in the variable appFile$ for later use. Control then returns first to SaveClip [ME 1], and then back to the endless loop in the main program [Main].
Append Clip

The next menu choice is Append Clip; its purpose is to append the contents of the Clipboard to an existing file. This is handled in AppClip [ME 2]. In it, “CLIP:PICTURE” is opened, and like SaveClip [ME 1], the Clipboard is checked to see if it contains something. If it doesn’t, control passes to ClipEmpty [SR 5], which we described earlier.

In AppClip, the FILES$ function has the parameter 1, meaning that an existing file will be used. The parameter TEXT specifies that we will be concerned only with text-type files. Although our files contain encoded pictures, remember that these are strings and thus text as far as the operating system is concerned. The FILES$ statement creates a dialog box similar to that produced in SaveClip, except that an inner window with the names of the text files on the current disk is created. (Using the TEXT parameter prevents the names of non-picture files from being displayed.) The user then double-clicks on the name of the file to which the picture is to be appended. Everything else works the same way as in SaveClip except, of course, that the file is opened for APPEND rather than for OUTPUT.

If the Clipboard does contain something, appFile$ is set equal to the name selected in the dialog box. If a name is not entered, the Cancel button must be pressed and the third item in the menu is reset to show no active file name. If a name is entered, control passes to AppToFile [ME 3]. This subroutine opens the file named in appFile$ for appending the picture and transfers control to ClipToFile [SR 4] which was described above.

Append Clip To

The third menu option is Append Clip To. The corresponding subroutine AppToFile [ME 3], appends the Clipboard’s contents to an already opened file. The name of the file is the last file used in the Append Clip option described above and is stored in appFile$. Control then passes to ClipToFile [SR 4] before returning to the main program.

Reading picture files

Now let’s look at MacInterface’s other principal activity: reading files of pictures and saving selected pictures to the Clipboard. ReadPics [ME 4] manages this function. First, it uses FILES$ to obtain the name of the file to be read, just as was done in AppClip. But if no file name is obtained,
ReadPics returns to MenuEvent without doing anything. Otherwise, the selected file is opened and a WHILE…WEND loop reads the pictures into the string-array *pict*$. This code is similar to that used in many of the games for reading images; the only puzzler might be the statement starting with *dummyByte*$. This is needed to read past the line-feed character that the PRINT statement tacked onto each picture string as it was appended to the file.

Next, a window is set up via the next few lines of code. The chosen file name is used as its title. The buttons give the choices of Next (display the next picture in the file), or Copy (copy the picture to the Clipboard). The fact that Window 2 is opened generates a dialog event 5. Since we are trapping dialog events, program control takes a quick sidestep to RegenWindow [DE 4]. The current window is number 2, so RefreshRead [SR 8] is called, which in turn calls the DisplayPict subprogram [SP 1] to display the current picture. Defining this routine as a subprogram rather than a subroutine allows it to handle any picture string; the STATIC qualifier guarantees that the contents of all the internal variables will not be disturbed between calls. The constants for window width and window height are accessed (using the WINDOW function) so that the subprogram will “know” where the Output window is located regardless of whether it was resized or not.

Remember the function definition IntVal in [Init 2] that we passed over when discussing the constants at the beginning of the program? Let’s take a look at this function now, because it’s used in DisplayPict. The function simply converts two characters, starting at a specified position, into an integer. The ASC function gets the integer ASCII value of the characters. Since integer values have two bytes, with the high-valued byte coming second, the second byte’s value is multiplied by 256 and added to the first to get the full integer represented by the two characters.

We have to take the long way around here because the picture string contains characters in a specific format that encodes the information and graphics instructions necessary for creating the picture. These must be converted to integer values before they can be used for displaying the picture. In particular, the height of the picture must be obtained by finding and subtracting the values of the y coordinates of the top (bytes 7 and 8 in the picture string) and bottom (bytes 3 and 4). The picture’s width is similarly obtained from its right and left x coordinates: bytes 9 and 10, and bytes 5 and 6, respectively.

The two IF statements that follow the code using IntVal take care of positioning the picture within the Output window. If the picture fits in
the window, it is centered; otherwise the x and/or y coordinates are changed to 10 to display as much of the picture as will fit, starting at the upper left. Finally, the PICTURE statement displays the picture.

After the current picture is displayed, control returns to ReadPics, where two buttons are created, one to advance from the currently displayed image to the next one in the file, and the other to copy the current image to the Clipboard. After this is done, ReadPic returns through MenuEvent to the main idle loop.

If the first button is clicked, the user wants to see the next picture. When the first button is clicked, a dialog event 1 occurs and the program flows through a DialogEvent [DE 0] and BtnClick [DE 1] to DisplayNext [SR 7]. DisplayNext increments current, wrapping back to image 1 if we are already at the last image in the file, and calls RefreshRead [SR 8]. RefreshRead displays the current image and prints its number in the window.

If the second button is clicked, the user wants to copy the currently displayed picture to the Clipboard, so CopyToClip [SR 6] is called. This deactivates the buttons, opens the “CLIP:PICTURE” file, and saves the current picture string from the array (pict$(current)) to the Clipboard. After the file is closed, the buttons are reactivated for the next selection. The window is closed, and program control then ends up back in the main program loop.

Auxiliary functions

We have now covered all but the Instruction and Quit choices from MacInterface’s pull-down menu. The Quit menu options [ME 6] and [ME 7] are just our old standbys. In this case they both call CleanUp [SR 2], which resets the menu and closes the windows before exiting. If the Instructions menu item is selected, MenuEvent calls Inst [ME 5]. This opens a window entitled Instructions, with no buttons, and returns to the main program. The opening of the window generates a dialog event 5 which is trapped by RegenWindow [DE 4] in DialogEvents. Here, dInfo is set so that PrintInst [SR 9] is called. After the instructions are printed, the main loop waits until another menu or dialog event occurs.

The main thing to note here is that the instructions are available whenever a menu selection can be made—not just at the beginning of the program. This means that the user can review the instructions without restarting the program any time he or she needs help. Since the help is provided as a menu event, it’s not necessary to anticipate all the places where
the user might get confused so that the program can test repeatedly for the entry of a help command. This may not be too important in a program like MacInterface that has only a few options, but help (perhaps even several levels of help) provided in this way can become invaluable in more complicated programs.

Suggestions for MacInterface

Rather than suggesting modifications for MacInterface, we recommend that you adapt some of its subroutines to your own programs. The most generally useful ones are those that read from and copy to the Clipboard, those that create or append to files of pictures, and the method presented here for maintaining two levels of event-trapping simultaneously. If you write a program that produces output you might want to use with MacPaint, MacWrite, or other applications, giving your program the option of sending output to the Clipboard would be very useful — and very easy to do, once you understand how MacInterface works. Long after you’ve gotten tired of a favorite game, a reliable utility or general-purpose subroutine or subprogram will continue to pay dividends in your work.
MacInterface program listing

MacInterface

Interface between files and the Clipboard

INIT

DEFINT A-Z
DIM pict$(20)

false = 0 : true = NOT false
wTop = 40 : wLeft = 5 : wBot = 320 : wRight = 250
wWidth = wRight - wLeft
wHeight = wBot - wTop
appFile$ = ""

'IntVal extracts a two byte integer value from s$ at position n
DEF FN IntVal(s$,n) = ASC(MID$(s$, n, 1)) * 256 + ASC(MID$(s$, n + 1, 1))

'FileName$ returns just the filename from a 'Volume:Filename' string
DEF FN FileName$(f$) = RIGHT$(f$, LEN(f$) - INSTR(f$,":"))

GOSUB Cleanup

MENU 6, 0, 1, "MacInterface"
MENU 6, 1, 1, "Save Clip As"
MENU 6, 2, 1, "Append Clip"
MENU 6, 3, 0, "Append Clip To:"
MENU 6, 4, 1, "Read from File"
MENU 6, 5, 1, "Instructions"
MENU 6, 6, 0, "-"
MENU 6, 7, 1, "Quit to BASIC"
MENU 6, 8, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON

GOSUB ClipHere

ON DIALOG GOSUB DialogEvent : DIALOG ON
WINDOW 1, "Clipboard", (wRight + 5, wTop) - (510, wBot), 1

MAIN

finished = false
WHILE NOT finished : WEND
GOSUB Cleanup
END

DE

DialogEvent:

dilog = DIALOG(0)
IF dilog <> 0 THEN dInfo = DIALOG(dilog)
ON dilog GOSUB BtnClick, , ActiveWindow, CloseBox, RegenWindow
RETURN
MAC INTERFACE

BtnClick:  
if 'Next' or 'Copy' buttons clicked  
btnn = dlnfo : wind = WINDOW(0)  
IF wind = 2 THEN ON btnn GOSUB DisplayNext, CopyToClip  
RETURN

ActiveWindow:  
make inactive window active  
WINDOW dlnfo  
RETURN

CloseBox:  
respond to click in Close Box  
IF dlnfo = 1 THEN finished = true  
IF dlnfo >= 2 THEN MENU 6, 2 + dlnfo, 1  
RETURN

RegenWindow:  
if window needs refreshing, regenerate contents  
currWind = WINDOW(1)  
WINDOW OUTPUT dlnfo  
RegenWindow  
RETURN

MenuEvent:  
item = MENU(1)  
MENU 6,0,0  
ON item GOSUB SaveClip, AppClip, AppToFile, ReadPics, Inst, , BasQuit, DeskQuit  
RETURN

SaveClip:  
save Clipboard to a new file  
OPEN "CLIP:PICTURE" FOR INPUT AS #2  
length = LOF(2) : CLOSE #2  
IF length = 0 THEN GOSUB ClipEmpty: RETURN  
file$ = FILE$(0, "Save as?")  
IF file$ = "" THEN SNoFile  
appFile$ = file$  
OPEN appFile$ FOR OUTPUT AS #1  
GOSUB ClipToFile  
SNoFile:  
RETURN

AppClip:  
append Clipboard to an existing file  
OPEN "CLIP:PICTURE" FOR INPUT AS #2  
length = LOF(2) : CLOSE#2  
IF length = 0 THEN GOSUB ClipEmpty: RETURN  
appFile$ = FILE$(1, "TEXT")  
IF appFile$ = "" THEN MENU 6,3,0,"Append Clip to:"  
IF appFile$ <> "" THEN GOSUB AppToFile  
RETURN
AppToFile:

    ' append Clipboard to appFile$
OPEN appFile$ FOR APPEND AS #1
GOSUB ClipToFile
RETURN

ReadPics:

    ' read pictures from file
file$ = FILE$(1, "TEXT")
IF file$ = "" THEN RNoAction
MENU 6,4,0
OPEN file$ FOR INPUT AS #1
pictures = 0
WHILE NOT EOF(1)
    INPUT #1, bytes
    pictures = pictures + 1
    pict$(pictures) = INPUT$(bytes,1)
dummyByte$ = INPUT$(1,1)
WEND
CLOSE #1
current = 1
title$ = FN FileName$(file$)
WINDOW 2, title$, (wLeft,wTop) - (wRight,wBot), 1
BUTTON 1, 1, "Next", (10,10) - (50,30)
BUTTON 2, 1, "Copy", (60,10) - (100,30)
RNoAction:
    RETURN

Inst:

    ' display instructions
WINDOW 3, "Instructions", (8,47) - (502,328), 1
RETURN

BasQuit:

    ' quit to BASIC
GOSUB Cleanup
END

DeskQuit:

    ' quit to Mac Desktop
GOSUB Cleanup
SYSTEM

------------------------------- SUBROUTINES

ClipHere:

    ' ask user if Clipboard has a picture
WINDOW 3, "", (100,100) - (400,200), -2
TEXTFONT(0) : LOCATE 2,3
PRINT "Does the Clipboard have a picture in it?";
BUTTON 1, 1, "Yes", (50,60) - (130,80)
BUTTON 2, 1, "No", (170,60) - (250,80)
WHILE DIALOG(0) <> 1 : WEND
IF DIALOG(1) = 1 THEN HasPict
OPEN "CLIP:PICTURE" FOR OUTPUT AS #1
CLOSE #1
HasPict:
    WINDOW CLOSE 3
RETURN
MAC INTERFACE

Cleanup:  ' -------------------------------- clean up desktop to quit program
MENU RESET
FOR n = 1 TO 3 : WINDOW CLOSE n : NEXT
RETURN

ShowClip:  ' -------------------------------- show contents of Clipboard
CLS
OPEN "CLIP:PICTURE" FOR INPUT AS #10
IF LOF(10) = 0 THEN NoShowClip
sBytes = LOF(10)
sClip$ = INPUT$(sBytes,10)
CALL DisplayPict(sClip$)
NoShowClip:
CLOSE #10
RETURN

ClipToFile:  ' -------------------------------- write Clipboard to file #1 (already open)
FOR t = 1 TO 1000 : NEXT ' give DialogEvent time to refresh Clipboard
OPEN "CLIP:PICTURE" FOR INPUT AS #2
bytes = LOF(2) ' get length of Clipboard in bytes
IF bytes = 0 THEN EmptyClip
clip$ = INPUT$(bytes,2)
PRINT #1, bytes : PRINT #1, clip$ ' append to disk file
CLOSE #1 : CLOSE#2
GOTO ACiipEnd

EmptyClip:
CLOSE #1: CLOSE #2
GOSUB ClipEmpty

ACiipEnd:
MENU 6, 3, 1, "Append Clip to:" + FN FileName$(appFile$)
RETURN

ClipEmpty:  ' -------------------------------- tell user Clipboard is empty
DIALOG OFF
WINDOW 4, ",", (100,100) - (400,200), -2
TEXTFONT(0) : LOCATE 2,5
PRINT "Sorry, the Clipboard is empty.";
BUTTON 1, 1, "OK", (230,60) - (280,80)
WHILE DIALOG(0) <> 1 : WEND
DIALOG ON
WINDOW CLOSE 4
RETURN

CopyToClip:  ' -------------------------------- copy current picture in file to Clipboard
BUTTON 1,0 : BUTTON 2,0
OPEN "CLIP:PICTURE" FOR OUTPUT AS #2
PRINT #2, pict$(current)
CLOSE #2
currWind = WINDOW(1) : WINDOW OUTPUT 1
GOSUB ShowClip
WINDOW OUTPUT currWind
BUTTON 1,1 : BUTTON 2,1
RETURN

DisplayNext:                      display next picture in pict$
  current = (current MOD pictures) + 1                'increment, reset to 1 if last image
GOSUB RefreshRead
RETURN

RefreshRead:                     refresh 'Read from file' window
CALL DisplayPict(pict$(current))
CALL MOVETO(130,25) : PRINT "#"; current;
RETURN

PrintInst:                      print instructions
CLS
PRINT
PRINT " With MacInterface you can transfer pictures from the Scrapbook to a"
PRINT " file, and move a picture file into the Scrapbook. Pictures are moved"
PRINT " between file and Scrapbook through the Clipboard in a two-step process."
PRINT " Use the normal Edit menu to move pictures between the Clipboard and"
PRINT " Scrapbook, and these menu commands to move between Clipboard and file:"
PRINT
PRINT " Save Clip As: Asks for a file name and puts the contents of the"
PRINT " Clipboard into that file."
PRINT " Append Clip: Asks for a file name and appends the picture in the"
PRINT " Clipboard to the end of that file."
PRINT " Append Clip To: Appends a picture in the Clipboard to the file currently"
PRINT " opened by MacInterface."
PRINT " Read from File: Loads a selected picture file into memory. Use the Next"
PRINT " button to leaf through the pictures in the file and the"
PRINT " Copy button to copy individual pictures into the"
PRINT " Clipboard. ";
RETURN

SUB DisplayPict(p$) STATIC               display picture in p$
  winW = WINDOW(2) : winH = WINDOW(3)          'get window height and width
  IF LEN(p$) < 10 THEN BadPic
  picLen = FN IntVal(p$,7) - FN IntVal(p$,3)   'find picture height
  picHi = FN IntVal(p$,9) - FN IntVal(p$,5)    'and width. (bytes 3-10)

  IF picLen > winW THEN x = INT((winW - picLen) / 2)
  IF picHi > winH THEN y = INT((winH - picHi) / 2)
  CLS : PICTURE(x,y), p$                      'clear screen and display picture
BadPic:
END SUB
For many people, gridlocks have become a part of everyday life. The flight to the suburbs means that millions of us now spend sizable chunks of our lives locked in one traffic jam or another. We can’t promise you hassle-free commutes, but there is a kind of gridlock our MacGrid utility can banish forever. We’re talking about graph paper, the prosaic but useful tool
that makes it possible to plot the results of school lab experiments, lay out maps and plans, design computer screen displays, and accomplish many other applications where scale or relationship between objects must be shown precisely.

The "gridlock" comes from the fact that only a few sizes of commercially produced graph paper are readily available. If the size of the squares is too small, it's hard to plot the data and show the reference points. If the squares are too big, the plot is too coarse or can't fit all the data. Until now, you've had to compromise.

MacGrid allows you to create and print customized graph paper by specifying the margins, total height and width, vertical and horizontal spacing, and intervals for heavy reference lines. If you need just a few sheets, you can print them on your printer, or print just one and use it as a master for photocopying. Once you've seen how fast and easy it is to create customized graph paper, you may even discover some new uses for it.

Before we begin our discussion of the program, we should first mention that this program is set up to work with the ImageWriter, since most of you have this printer.

How to use MacGrid

When the program starts running, you first see the title screen shown in Figure 1. A sample grid is shown in the bottom half of the screen, and you are informed that the MacGrid menu is now active. This menu has two main options: Grid Setup and Print.

To specify the characteristics of a grid, choose Grid Setup from the menu. You then see a screen like that in Figure 2, with edit-field boxes for entering the numbers needed. A default value is given for each characteristic, and the first field is inverted, indicating that the insertion point is there. You can edit these values using the standard Macintosh editing functions, including Cut, Paste, and Copy. You can move from field to field using the Tab or Return keys, or you can click the mouse in a field to move the insertion point there. When you're satisfied with the values, click the OK button in the lower right corner of the screen. If you decide you don't want to change the specifications, click the Cancel button instead.

Grid specifications

The program instructions, obtainable by choosing Instructions from the MacGrid menu, briefly summarize the dimensions and other specifications used to create a grid. We will elaborate on them a bit here.
MacGrid

Design your own graph paper.

The MacGrid menu is now active.

FIGURE 1. The MacGrid title screen

Edit the parameters to produce the grid you want. Enter fractions as either their decimal or fraction value. All values are in inches except the heavy line values which are an increment of the corresponding grid spacing.

Top Margin: 1
Left Margin: 1/2
Vertical Spacing: 1/4
Horizontal Spacing: 1/6
Total Height: 8
Total Width: 7
Heavy Line Interval: 8
Heavy Line Interval: 6

FIGURE 2. Entering grid parameters
Top Margin is the distance in inches from the top of the print head down to the line at which printing of the grid will begin. Because the reference point used is the print head (from its starting position on the left side of the printer), make sure you have it aligned with the top perforation of the paper before printing a piece of graph paper. Our example grid has a top margin of 1 inch.

Left Margin sets the distance in inches from the left edge of the print head (in its starting position) to the first vertical line of the grid. Our sample grid has a left margin of \( \frac{1}{2} \) inch.

Vertical Spacing is the spacing between horizontal lines as you move vertically down the grid. Since the inch is the basic unit of measurement used, vertical spacing is expressed in inches and fractions of an inch. It would usually be small fractions of an inch, unless you want a grid with very wide spaces. You can express these and other values as either decimal numbers or fractions. We find that using fractions actually makes it clearer: For example, our specification in Figure 2 of a vertical spacing of \( \frac{1}{4} \) inch also translates to “four horizontal lines per inch,” with the denominator of the fraction being equal to the number of lines per inch. You could say “0.25,” but then you would have to divide it into 1.00 to get the number of lines per inch.

Horizontal Spacing establishes the spacing between vertical lines as you move across the grid (that is, horizontally). As a fraction, the denominator expresses the number of vertical lines per inch. (Our example grid has a horizontal spacing of \( \frac{1}{6} \) inch, or six vertical lines per inch.) It follows that if you want the grid to consist of little squares, you should specify equal horizontal and vertical spacing. You would usually want equal vertical and horizontal spacing to plot equations, for example. For other uses where you have a much finer scale in one direction than in the other, you might want unequal spacing, as we have shown in Figure 2. For example, if you wanted to plot the hourly temperature as measured by a weather station, you might want the grid to have only 24 vertical lines across, with a horizontal spacing of \( \frac{1}{4} \) inch (four vertical lines per inch) for a total page width of 6 inches. You might want a vertical spacing of \( \frac{1}{10} \) inch (10 horizontal lines per inch) for a total page height of ten inches, allowing you to plot a range of up to 100 degrees.

Total Height is the total height of the grid, not of the sheet of paper. The printer will move to the top of the next sheet automatically after the
grid has been printed (assuming that you have normal 8½-by-11-inch paper and the normal top-of-form setting). In our example in Figure 3, on the next page, the grid height is 8 inches.

*Total Width* again refers to the *grid*, not the sheet of paper as a whole. In our example, *Total Width* is set to 7 inches.

*Heavy Line Interval* is simply the interval at which a heavy, or emphasized, line will be printed. You can see in Figure 2 that there are two specifications for heavy lines, one vertical and one horizontal. We have a heavy line every eighth line vertically (that is, every eighth horizontal line is heavy), and a heavy line every sixth line horizontally (every sixth vertical line is heavy). If you look at the line spacing and corresponding heavy line specifications in Figure 2, then look at our example printout in Figure 3 on the next page, you'll see the grid is in effect subdivided by the heavy lines into sectors 1 inch across by 2 inches down.

You will want to choose the heavy line intervals according to meaningful divisions of the units your grid will represent. For measuring temperature, for example, you might want a heavy line every tenth line vertically to indicate a division of 10 degrees. In general, you might find it useful to sketch what the grid is intended to represent or plot to help you determine the specifications needed. If you don't want any heavy lines in a particular direction, simply enter 0 in the appropriate field.

Once you’ve entered the specifications and clicked the OK button, you’ll return to a screen displaying the menu bar and the upper left corner of the grid you’ve just specified. Although you won’t be able to tell how all your margins will look, you can use this display to check your line spacing and heavy line intervals before printing.

**Printing the grid**

To print the grid you’ve specified, pull down the MacGrid menu and choose Print. (Make sure that your paper is lined up properly and that the printer is on and has the “select” indicator lit.) Printing will take a while, so this might be a good time to get up and stretch a bit.

That’s all there is to using MacGrid to create customized graph paper. We’ll suggest some additional features you might want to add to the program after we’ve discussed its operation.
How MacGrid works

MacGrid has our standard menu event-driven structure with an endless WHILE...WEND loop waiting for events. The main interest in the program is not in its flow of control, but instead in the way the grid lines are laid out using the specifications given. A program outline for MacGrid is shown in Figure 4.
### Initialization
- Initialize array references, window dimensions, other constants
- Open Output window
- Display title screen
- Install MacGrid menu

### Main Program
- Endless loop interrupted by menu events

### Menu Events
- MenuEvent: branch to appropriate subroutine
  - After routine, draw graph (DrawGraph)
- GridSetup: set up grid parameters
  - Open secondary window (OpenWindow)
  - Print instruction text
  - Set up edit fields (SetEdit)
  - Set up OK and Cancel buttons
  - While not quitting
    - If dialog event is a button, check if OK or Cancel (XitBtn)
    - If event is clicking in new edit field, change to that field
    - If event is Tab or Return key, change to next field
  - WEND
  - Close secondary window
- Prinltt: print grid
  - Save ID number of current window
  - Send window output to printer
  - Draw grid (DrawGraph)
  - Close output
  - Restore output to saved window ID

### Subroutines
- SetEdit:
  - Read edit-field labels from DATA statements
  - Calculate edit-field label coordinates and print label
  - Convert edit-field parameters into string-fraction form (Numberize)
  - Display fields
- XitBtn:
  - If OK button, read edit fields (ReadEdit)
- ReadEdit:
  - For each field, read constants of field
  - Convert into numerator and denominator form
  - Calculate numeric parameter

(continued)
Numberize:
  turn parameter into numerator/denominator string form

DrawGraph:
  get horizontal and vertical spacing of grid lines; heavy line intervals
  get values for margins
  calculate total lines across and down
  draw vertical lines at specified spacing
    if a line should be heavy, use larger pen width
  draw horizontal lines at specified spacing
    if a line should be heavy, use larger pen width

TitleScreen:
  display opening title
  draw sample grid

OpenWindow: open secondary window

Figure 4. Program outline (continued)

Initialization

The constants top, height, left, and wide are set to the values 0 through 3, respectively. These will be used to refer to the first four elements of the params array, which will contain the grid dimensions. Using names to refer to the array elements is a useful technique because it means that as we write code, we can use the name rather than having to look up the number; that is, when we want the left margin we can refer to params(left), rather than having to go back and look up the number for that array element. (Pascal programmers will recognize this technique as being similar to the use of enumerated types.)

The other constants initialized in this segment are the boundaries for the secondary window (to be used for dialog boxes), and the number of pixels-per-inch horizontally and vertically. The latter two numbers are used in converting grid specifications in inches to the actual number of pixels (and thus the screen coordinates) needed to enable the ImageWriter to print the grid.

In [Init 2], the title screen is drawn by calling TitleScreen [SR 6]. This subroutine starts by reading the parameters for the default grid from the data statements following the label TitleParams [Dat 1]. (If most of your grids are going to have the same margins, you can of course substitute your own values and save time on data entry.) The title text is displayed, and five sample grids are drawn in successively finer gradations by a FOR...NEXT loop. Each grid is superimposed on the preceding one.
We will discuss how a grid is drawn later, after we've looked at how the specifications are obtained. For now, note that the top margin is 2 inches, enough to avoid overprinting the title text. As each of the superimposed grids is drawn, a FOR...NEXT delay loop nested within the main loop slows things down enough so that you can see each grid laid on top of the preceding one. Note how the values for vertical and horizontal spacing in the params array are successively divided by 2 to make each graph twice as "dense" as the preceding one. Finally, the user is reminded that the MacGrid menu is available for selection.

After the program returns from drawing the title screen, the MacGrid menu is created and installed. Menu event-trapping is enabled: Menu events will be handled by MenuEvent [ME 0]. Note that after a menu choice is processed, MenuEvent calls DrawGraph to draw whatever grid is currently specified. This is handy because it means that the user, after entering new grid specifications, can see what the new grid will look like. The Instructions, Quit to BASIC, and Quit to Desktop options are standard for our programs.

The program now enters the usual endless loop [Main] to wait for menu events. There are two subroutines that do the real work of the program: GridSetup, where the user's specifications for a grid are obtained, and Print, where the grid is drawn in the Output window and is then sent to the printer.

**Grid setup**

GridSetup [ME 1] handles the Grid Setup choice from the MacGrid menu. First, OpenWindow [SR 7] is called to open the secondary window. Note that the window type is 2, which is a bordered dialog box that can't be moved or resized. After returning from OpenWindow, some instructions are displayed.

Next, SetEdit [SR 1] is used to set up the edit fields, where the user enters the specifications. The constants meX and meY mark the (x,y) coordinates for the start of the edit field area. A FOR...NEXT loop is used to display and activate each of the eight fields.

Each field on the screen is preceded by a label describing the specification. These labels are read into the string m$ from EdData [Dat 2]. Next, the position for displaying the string is calculated. Note how the x coordinate will alternate between just meX = 10 and meX + 240, or 250. This is because of the expression (n MOD 2), which will be 0 when the loop index n is 0, 1 when it is 1, 0 again when it is 2, and so on. Since we want the
y coordinate to stay the same for each horizontal pair of fields, then increase for the next pair, and so on, the expression INT(n / 2) is used. Since INT(n / 2) equals 0 for both n = 0 and n = 1, the y coordinate for the first pair of fields will be just meY. For both n = 2 and n = 3, the expression will equal 1, so the y coordinate will increase by 30, putting these fields 30 pixels farther down the screen. For the next pair, the expression will equal 2 and thus add another multiple of 30, and so on. You might make a note of this use of MOD and INT expressions, because it's a handy way to format any tabular arrangement of strings or other items.

After the label string for a field has been displayed, the default value for the field, which is the corresponding element from the params array, is assigned to the variable num. Numberize [SR 4] has the job of converting a number num to a string containing the number with the appropriate fraction in numerator/denominator form. First, the easy part: The variable n1 is set to INT(num), which takes care of the whole-number part of num. The fractional part is num − n1, which is assigned to n2. The string n$, which will contain the string form of the complete number, is initialized as empty, or null (""").

The next step is to convert the whole-number part to a string. If n1 <> 0, there is a whole-number part, since n1 was obtained as the INT part of num. If num = 0, we consider the 0 to be the whole-number part. The BASIC functions RIGHT$, LEN, and STR$ can then be used to obtain the string. The RIGHT$ of 1 character less than the length of STR$(n1) is taken so that the leading blank that STR$ automatically supplies can be stripped off. At this point, if n2 = 0, the user entered only a whole number, and the subroutine returns.

If a fraction remains, we will try to convert it from a decimal fraction to a numerator/denominator-type fraction. A FOR...NEXT loop is used to test possible fractions to see if they equal the decimal value n2. The denominator (n4) is increased by 1 each time through the loop, starting at 2. The first statement in the body of the loop divides 1 by the current denominator and then divides the result into the decimal fraction we're trying to match (n2). The next statement tests to see if the result of this operation is a whole number; that is, if n3 = INT(n3). If it is a whole number, the denominator divided evenly without any remainder, which means that we have found the right denominator. In this case control goes to the label Fraction. The numerator of our fraction is the number of times the fraction 1 / n4 goes into n2, which is assigned to n3. The denominator is the value n4 had when we left the loop. The string functions are then used to format n3 / n4 as a string.
If we go through all the values of \( n_4 \) and exit the loop without finding the correct denominator, it means either that the decimal fraction is too small for our possible denominators, or that it is one of those decimals that can’t be converted to a fraction with a denominator under 16; for example, 0.33. Numbers like these work perfectly well for producing the grid; they just can’t be converted to simple fractions. None of our default values causes this problem, but the user might enter such a value. In the case of such “intractable” fractions, we give up, convert the number \( \text{num} \) to a string without converting it to an ordinary fraction, and return.

After the “numberization” of the default specification, the \texttt{EDIT FIELD} statement in \texttt{SetEdit [SR 1]} is used to set up the edit field, complete with the default value in \( n_4 \$ \) returned by \texttt{Numberize [SR 4]} and the rectangle in which data will be entered. We cover the meaning of the various parameters used in edit fields, and the use of the \texttt{EDIT$} function (which appears later in this program), in the chapter on MacGraph, so we will refer you to that chapter for details. When the \texttt{FOR...NEXT} loop is finished, \texttt{SetEdit} returns to \texttt{GridSetup [ME 1]}.

The first edit field is then made the current one, and another \texttt{EDIT FIELD} statement places the insertion point there and inverts the field so the user knows where to enter data. Next, the OK and Cancel buttons are created, and the \( \text{exit} \) (exit) flag, which will indicate the user has clicked one of the buttons, is set to \( \text{false} \).

A \texttt{WHILE...WEND} loop then takes care of responding to the user’s activity. There are basically two things the user may be doing: moving between edit fields, or clicking buttons. The \texttt{DIALOG(0)} function provides information on which of these activities is being performed, and the code it returns is assigned to the variable \( \text{diag} \).

If \( \text{diag} \) is 1, a button was clicked. The ID of the button clicked is returned by \texttt{DIALOG(1)}. \texttt{XitBtn (“exit button”)} [SR 2] is called. The first thing it does is set \( \text{exit} \) to \( \text{true} \). If the button clicked was number 2 (Cancel), we don’t need to do anything else: When the subroutine returns, the \texttt{WHILE...WEND} will be exited because \( \text{exit} \) is true, and \texttt{GridSetup} will return without changing any parameters, since the user wanted to cancel the operation. If the button was number 1 (OK), the user has finished entering values and we need to update the \texttt{params} array with these values. \texttt{Read-Edit [SR 3]} is called for this purpose.

A \texttt{FOR...NEXT} loop in this subroutine is used to obtain the values for each of the eight edit fields. First, the string in the edit field is
obtained using BASIC's EDIT$ function. Since edit field identifiers start
with 1, and while the elements of params start with 0, 1 is added to the
loop index \( n \) to get the edit field corresponding to the array element.

The two WHILE...WEND loops that follow are used to strip off
any leading and trailing blanks from the string. BASIC's VAL function is
then used to get the first part of the value of the string and assign it to the
variable \( m \). The string may be an integer plus fraction (for example, \( 4\frac{1}{2} \)),
or it may not have an integer part. The VAL function stops converting
when it runs into a slash (/). Thus if the string has only an integer, the in­
teger will be found and converted. If it has only a fraction, the numerator
of the fraction will be obtained. If it has both an integer and a fraction, the
integer plus the numerator of the fraction will be obtained.

The INSTR function is used to check for an internal space in the
string, which would indicate that there is both an integer and a fraction,
since the space would separate them. If the space is found, the portion
of the string up to the space is given to VAL and its value is assigned to \( m1 \).
The remaining part of the string is assigned to \( m\$ \), and its value is put into
\( m \). If this turns out to be a fraction, VAL will convert only up to the slash
dividing the numerator and denominator, and the numerator of the frac­
tion will have been converted to \( m \).

If the space isn't found, we have only a fraction, so control goes to
the label NoInt, and the location of the slash dividing the numerator and
denominator is searched for. If it is found, we know that \( m \) has the numer­
ator, obtained in the original conversion following the WHILE...WEND
loops, and we use RIGHT$ to extract the portion of the string to the right
of the slash; that is, the denominator. We then simply divide the de­
nominator into the numerator to get the decimal value of the fraction.
This is then added to \( m1 \) (the integer value, if any) to get the total value of
the string, which is assigned to the appropriate element of params.

If the slash isn't found there's no fraction, so \( m \) remains at its origi­
inal value (the integer part), and is put into the array element. When the
main loop (FOR...NEXT) is done, ReadEdit has converted all the param­
eters in the edit fields to their proper numeric values and stored them in
the params array. Control then returns to XitBtn, and since xit was set to
true, the program returns to GridSetup and finishes processing it.

If the user hasn't clicked either button, we check next for a value
of \( diag = 2 \). This means the user has clicked on a new edit field. In this case,
the current edit field is set to the new field. If \( diag = 5 \), the Return or Tab
keys were used to move the insertion point to the next edit field. The MOD
expression is used to wrap around to the first edit field if necessary.
Assuming the user has entered data and finally clicked the OK button, the program is now ready to draw and print the grid when the Print option on the MacGrid menu is chosen.

**Drawing and printing the grid**

PrintIt [ME 2] handles the Print option. It first saves the identifier of the current Output window (returned by the WINDOW(0) function) in the variable `crntWindow`, so it can be restored later. The printer file `Lpt1:` is opened for output.

DrawGraph [SR 5] is then called, and starts by getting the values it needs from the `params` array. The values `vSpacing`, `vHeavy` (spacing and interval for emphasized vertical lines) and the corresponding horizontal values (`hSpacing`, `hHeavy`) are simply obtained from the corresponding elements of the `params` array. The values that have to be converted to screen coordinates (that is, number of pixels) are the top and left margins and the width and height of the grid. In each case the value is obtained from the corresponding array element and multiplied by the number of pixels-per-inch in the appropriate direction (`inchVert` or `inchHorz`).

As you might expect, FOR...NEXT loops are used to draw the vertical and horizontal lines. The general procedure is to use the STEP clause to give the proper spacing and step through the index values to place and draw the lines. We also have to check if a line should be heavy. We will use the counter-variable `heavy` for this purpose; it is initialized to 0. The space between the vertical lines is `inchHorz * Spacing`. For each line, we check to see if it is a heavy line: If it is, a wider PENSIZE is set. If `heavyFlag` is 0, no heavy lines at all are wanted; otherwise, we set the pen for a heavy line.

The `heavy` counter is then incremented by 1. The starting point for the vertical line is offset from `l` (the left margin), and its y coordinate is `t`, the top of the grid. The QuickDraw LINE routine (not to be confused with BASIC's LINE statement) is used to draw the line. Remember that the coordinates given to this routine are **relative** to the pen position and, since the line is vertical, the relative x coordinate is 0; that is, we want it to be the same as the pen's x position. The y coordinate is the value of the variable `high`, the height of the grid.

When the FOR...NEXT loop exits, one more line is needed, because we started the loop at 0; that is, on the left margin. If any vertical heavy line interval other than 0 was specified, this last line is drawn as a
heavy line, thus serving as the right border of the grid. The expression
\( v\text{Heavy} = 0 \) is used to set the correct pen size without another IF statement
being needed. (There's always a tradeoff between using relatively cryptic
code like this versus the complexity of further nesting of IF...THEN state-
ments.) In case you're wondering how the left border heavy line was ob-
tained, the first line on the left would also have been a heavy line, since
\( 0 \text{ MOD } h\text{Heavy} \) would have given 0 the first time through the loop.

The horizontal lines are drawn in exactly the same fashion, except of
course that this time the starting x coordinate (the left margin, or \( l \)) will
remain constant, while the y coordinate is offset from the top margin (\( t \)).

After DrawGraph is done, the grid has been printed on the printer as
well as displayed on the screen, since the printer file had been opened for
output. This file is now closed, and the Output window is set back to what it
was before the Print option was chosen.

**Suggestions for MacGrid**

A relatively simple thing to do would be to add edit fields for printing la-
blels or other text on your sheet of graph paper—perhaps at the bottom.
Some things you might want to have appear are: a title for whatever is
going to be plotted on the paper, your name, a page number, the number of
lines per inch horizontally and vertically, and the units being represented
on the grid (that is, degrees, kilograms, or whatever). You will probably
want to create a routine to handle these edit fields if you decide to add
more than one or two, so the edit screen won't get too crowded. You might
also want to have a separate routine print the text after DrawGraph is
done (but before you close the printer file).

Another useful enhancement might be to set the pen pattern to
some pattern that would give a dotted line for the minor divisions and a
solid black line for the major divisions, instead of varying only the thick-
ness of the lines.
MACGRID

MacGrid program listing

MacGrid
Make Graph paper on your Imagewriter.

INITIALIZATION

DIM params(7)
false = 0 : true = NOT false
top = 0 : height = 1 : left = 2 : wide = 3
wind2L = 16 : wind2T = 55
wind2W = 479 : wind2H = 270
inchHorz = 80
inchVert = 72

'80 pixels/inch horizontally
'72 pixels/inch vertically

WINDOW 1, "MacGrid", (4,28) - (506,332), 4
GOSUB TitleScreen

MENU 6, 0, 1, "MacGrid"
MENU 6, 1, 1, "Grid Setup"
MENU 6, 2, 1, "Print"
MENU 6, 3, 1, "Instructions"
MENU 6, 4, 0, "-"
MENU 6, 5, 1, "Quit to BASIC"
MENU 6, 6, 1, "Quit to Desktop"

ON MENU GOSUB MenuEvent : MENU ON

MAIN PROGRAM

WHILE true : WEND

MENU EVENTS

MenuEvent:
MENU 6, 0, 0
item = MENU(1)
ON item GOSUB GridSetup, Printit, Instructions, BasQuit, DeskQuit

CLS : GOSUB DrawGraph
MENU 6, 0, 1
RETURN

GridSetup: set up grid parameters
GOSUB OpenWindow
CALL TEXTFONT(1)
PRINT
PRINT " Edit the parameters to produce the grid you want. Enter fractions as"
PRINT "either their decimal or fraction value. All values are in inches except"
PRINT "the heavy line values which are an increment of the corresponding grid"
PRINT "spacing."
CALL TEXTFONT(0)
GOSUB SetEdit
eField = 1: EDIT FIELD eField
BUTTON 1, 1, "OK", (wind2W - 75, wind2H - 50) - (wind2W - 5, wind2H - 30)
BUTTON 2, 1, "Cancel", (wind2W - 75, wind2H - 25) - (wind2W - 5, wind2H - 5)
xit = false
WHILE NOT xit
  diag = DIALOG(0)
  IF diag = 1 THEN btn = DIALOG(1) : GOSUB XitBtn
  IF diag = 2 THEN eField = DIALOG(2)
  IF diag > 5 THEN eField = (eField MOD 8) + 1 : EDIT FIELD eField
WEND
WINDOW CLOSE 2
RETURN

PrintIt:  print grid on Imagewriter

Print: 

Print output grid on Imagewriter

crntwindow = WINDOW(0)
OPEN "Lpt1:" FOR OUTPUT AS #1
WINDOW OUTPUT #1
GOSUB DrawGraph
CLOSE#1
WINDOW OUTPUT crntwindow
RETURN

Instructions:  display instructions

WINDOW 2, (16,55) - (495,325), 2
PRINT
PRINT " MacGrid gives you the capability to print out a grid with whatever"
PRINT "spacing you want on a single sheet of paper."
PRINT " Set up the parameters for the grid with the 'Grid Setup' menu. All values"
PRINT " are entered in inches except the Heavy lines. The top and left margins are"
PRINT "the distance from the top left corner of the printer to the top left corner of"
PRINT "the grid. Because you can adjust the paper in the printer, the distance from"
PRINT "the corner of the sheet of paper will be somewhat different."
PRINT " Horizontal and vertical spacing is the distance between individual lines"
PRINT "on the grid (i.e., 1/10 gives you 10 lines per inch)."
PRINT " 'Heavy Line Interval' is the grid line multiple on which a heavy line will"
PRINT "be drawn (i.e., if the spacing is 1/10 inch and 'Heavy Line Interval' is 10,"
PRINT "then a heavy line will be drawn every inch). If this value is 0 then no"
PRINT "heavy lines will be drawn."
עמוד 46 של הספר "MIDNIGHT MADNESS"

CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1 : WINDOW CLOSE 2
RETURN

BasQuit: quit to BASIC

MENU RESET
END
DeskQuit:

' quit to Mac Desktop

MENU  RESET
SYSTEM

SUBROUTINES

SetEdit:

' set up parameter edit boxes

meX = 10 : meY = 100
RESTORE EdData
FOR n = 0 TO 7
  READ m$
  mX = meX + (n MOD 2) * 240 : mY = meY + INT(n / 2) * 30
  CALL MOVETO(mX, mY + 12) : PRINT m$;
  num = params(n) : GOSUB Numberize
  EDIT FIELD n + 1, n$, (mX + 140, mY) - (mX + 210, mY + 15), 2
NEXT n
RETURN

XitBtn:

' either button 'OK' or 'Cancel' has been hit

xit = true
IF btn = 1 THEN GOSUB ReadEdit
RETURN

ReadEdit:

' read parameter edit fields

FOR n = 0 TO 7
  m$ = EDIT$(n + 1)
  WHILE LEFT$(m$, 1) = " " : m$ = RIGHT$(m$, LEN(m$) - 1) : WEND
  WHILE RIGHT$(m$, 1) = " " : m$ = LEFT$(m$, LEN(m$) - 1) : WEND
  m = VAL(m$) : m1 = 0
  j = INSTR(m$, "")
  IF j = 0 THEN NoInt
  m1 = VAL(LEFT$(m$, j - 1))
  m$ = RIGHTS$(m$, LEN(m$) - j)
  m = VAL(m$)
NoInt:
  j = INSTR(m$, "/")
  IF j <> 0 THEN m = m / VAL(RIGHT$(m$, LEN(m$) - j))
  params(n) = m + m1
NEXT n
RETURN

Numberize:

' given number, returns N$ which is num w/fractions

n1 = INT(num) : n2 = num - n1 : n$ = ""
IF n1 <> 0 OR num = 0 THEN n$ = RIGHTS$(STR$(n1), LEN(STR$(n1)) - 1)
IF n2 = 0 THEN RETURN
FOR n4 = 2 TO 16
    n3 = n2 / (1 / n4)
    IF n3 = INT(n3) THEN Fraction
NEXT
n$ = n$ + RIGHT$(STR$(n2), LEN(STR$(n2)) - 1)
'n$ = RIGHT$(STR$(num), LEN(STR$(num)) - 1)
RETURN

Fraction:
    n3 = INT(n2 / (1 / n4))
    n$ = n$ + STR$(n3) + "." + RIGHT$(STR$(n4), LEN(STR$(n4)) - 1)
RETURN

DrawGraph:
    ' ______________________________________________________ draw graph with given parameters
    vSpacing = params(4) : vHeavy = params(5)
    hSpacing = params(6) : hHeavy = params(7)
    t = params(top) * inchVert : l = params(left) * inchHorz
    wid = params(wide) * inchHorz
    high = params(height) * inchVert

    CALL PENNORMAL: heavy = 0
    FOR n = 0 TO wid STEP inchHorz * hSpacing
        heavyFlag = (hHeavy AND (heavy MOD hHeavy = 0))
        IF heavyFlag THEN CALL PENSIZE(2,1) ELSE CALL PENNORMAL
        heavy = heavy + 1
        CALL MOVETO(l + n, t)
        CALL LINE(0, high)
    NEXT n
    IF hHeavy THEN CALL PENSIZE(2, 2 + (vHeavy = 0)) ELSE CALL PENNORMAL
    CALL MOVETO(l + wid, t) : CALL LINE(0, high)

    CALL PENNORMAL: heavy = 0
    FOR n = 0 TO high STEP inchVert * vSpacing
        heavyFlag = (vHeavy AND (heavy MOD vHeavy = 0))
        IF heavyFlag THEN CALL PENSIZE(1,2) ELSE CALL PENNORMAL
        heavy = heavy + 1
        CALL MOVETO(l, t + n)
        CALL LINE(wid, 0)
    NEXT n
    IF vHeavy<>0 THEN CALL PENSIZE(2 + (hHeavy=0), 2) ELSE PENNORMAL
    CALL MOVETO(l, t + high) : CALL LINE(wid,0)
RETURN

TitleScreen:
    ' ______________________________________________________ display opening screen
    RESTORE TitleParams
    FOR n = 0 TO 7 : READ params(n) : NEXT
    CALL TEXTSIZE(36) : CALL TEXTFONT(0)
    CALL MOVETO(175,50) : PRINT "MacGrid"
CALL TEXTSIZE(12)
CALL MOVETO(160,90) : PRINT "Design your own graph paper."
CALL TEXTFONT(1)
FOR ti = 1 TO 5
    GOSUB DrawGraph
    IF ti <> 5 THEN params(6) = params(6) / 2 : params(4) = params(4) / 2
    FOR t = 1 TO 1000 : NEXT
NEXT ti
CALL MOVETO(160,130) : PRINT "The MacGrid menu is now active."
RETURN

OpenWindow: ____________________________ open second window
   WINDOW 2, , (wind2L,wind2T) - (wind2L + wind2W, wind2T + wind2H), 2
RETURN

TitleParams:
   DATA 2, 2, .5, 5, 2, 4, 2, 4

EdData:
   DATA "Top Margin: ", "Total Height: ", "Left Margin: ", "Total Width: 
   DATA "Vertical Spacing: ", "Heavy Line Interval: 
   DATA "Horizontal Spacing: ", "Heavy Line Interval: 

There are several data base and financial-analysis packages for the Macintosh that can produce assorted fancy graphs and charts. Our MacGraph utility can't do everything that these expensive commercial programs can, but it provides a lot of their power at a cost of only a couple of hours of typing. With MacGraph, you first enter numeric data with an easy-to-use
data editor that supports all the standard Macintosh editing functions. Data can be saved to disk files or reloaded from disk, and MacGraph can even read data from standard text files in Microsoft BASIC input format. Once you have entered your data, you can provide labels and then create four different graphs: line, bar, pie graphs with percentages, and pie graphs with actual values.

What's more, graphs created in MacGraph can be transferred to a standard application like MacWrite or MacPaint via the Clipboard, or by saving a screen image to disk. This means you can incorporate your graphs in reports made with a word processor, or use MacPaint to add more elaborate labels, shading, and so on.

Another advantage of MacGraph is that once you've learned how its various functions work, you can modify them to suit your taste. If you decide you want an additional kind of chart, you can write code to produce that chart and simply add it to the menu. You won't have to write a data editor or file routines, since they're already written for you. If that expensive software package doesn't quite do what you want it to do, you're probably stuck with it—but MacGraph can become your all-purpose, expandable business-graphics toolkit!

How to use MacGraph

For our example MacGraph application, we'll create the (fortunately) fictitious Trailing Edge Software Company and trace its fortunes over the past year by graphing its sales (in thousands of dollars) for each of four quarters.

It seems that Trailing Edge started out as a fairly successful computer-game company, writing such classics as Bit Byter and Afternoon Madness (no relation to this book!) for the popular 8-bit home computers. It sold $120,000 worth of these games in the first quarter and then, emboldened by this modest success, adapted them to the Macintosh. Riding the growing tide of Mac popularity, Trailing Edge sold $210,000 worth of games in the second quarter. In the third quarter, the company added educational software, trivia games, and a real-time simulation of white-water rafting to its list and saw $348,000 worth of sales rung up in computer stores across the land.

In the last quarter, alas, disaster struck. Trailing Edge was acquired in a hostile takeover engineered by a Tibetan lama who had experienced a vision of millions of Macintoshes being turned into automatic prayer wheels. To finance this venture, he released Trailing Edge's most unusual (and, lamentably, final) product: Yak Attack, an arcade game in
which the player tries to lead a herd of balky yaks up the north slope of Mount Everest. Trailing Edge posted only $48,000 worth of sales in the last quarter and filed for bankruptcy shortly thereafter.

**Entering data**

When you run MacGraph, a title screen (shown in Figure 1) informs you that the program is ready and its two special menus are now active. These two menus appear to the right of the regular Macintosh and BASIC menus, and contain these options:

![MacGraph File menu](image)

![Graph menu](image)

**FIGURE 1. The MacGraph title screen**
Before we can produce graphs and charts, of course, we must get some numbers and words into the computer for MacGraph to work with. To enter the numbers for your graph, simply choose Edit Data from the MacGraph File menu, which produces a screen with a grid like that shown in Figure 2. An insertion point appears in box 1 (in the upper left corner of the grid), ready for your first number.

If you've been using a Macintosh for very long, you already know everything there is to know about using MacGraph's data editor, because it simply uses Microsoft BASIC's field-editing functions to provide all the standard Macintosh editing capabilities. You can cut, copy, and paste just as though you were editing icon names in the Finder, for example. Clicking on a box makes the insertion point appear in the middle of that box, and pressing the Tab or Return key moves it right (and down, if necessary) one box. You can simply enter your numbers, correct them if needed, and leave a blank box (or one with a negative number) to mark the end of your data. There are 40 boxes in all, which means there is room for 40 data points on your graph—enough for many types of charts.

You need to keep a few things in mind when entering data for a graph. First, MacGraph can graph only positive numbers. Pick units for

![Figure 2: The MacGraph data-editing screen](image)
your measurements that will be meaningful and not involve overly large numbers. For our Trailing Edge example, we picked thousands of dollars in sales as our unit; if we were doing Apple or IBM, we probably would have picked a larger unit. You don't have to worry about scaling your graph (making sure all values from the smallest to the largest fit on the chart): MacGraph does that for you.

After entering the data, simply click the OK button in the lower right corner; the data will stay in memory until they are changed.

**Saving and loading data**

Since you may plan to use your data in a later session (and perhaps edit it further), you'll probably want to save it to a disk file. To do this, simply choose Save As from the MacGraph File menu. A standard file dialog box will appear, and you'll be asked for the file name under which to save the data. You can load a previously saved data file from disk by choosing Open from the same menu and clicking on the name of the file you want.

In order for MacGraph to use a file, it must either consist of data entered through the MacGraph data editor, or it must be a *text* file in which there is only a series of numbers separated by commas (as they would be typed in response to an INPUT statement), or a list of numbers each on a separate line. You can create the file in BASIC's List window, but be sure to save it with the Text option. You can also copy lists of numbers from a word-processed document, or have them generated by a BASIC program, as long as you make sure that this format is followed.

**Labeling your graph**

A graph or chart with only numbers isn't very useful, since you wouldn't be able to tell whether it represents the gross national product of Outer Slobbovia or the number of Macintoshes owned by Fortune 500 companies. To add labels to describe your numbers, choose Edit Titles from the MacGraph File menu, which causes the screen shown in Figure 3 on the following page to appear. Using the same editing functions as the data editor, you can enter a title for the graph, a label for the vertical axis, and a label for the horizontal axis. In Figure 3 you can see what we entered for our Trailing Edge example, and in Figures 4 through 7 you can see how these titles appear on the finished charts. The names entered here, though, will not be kept with the data if the data is saved and must be re-entered each time the program is run.
Enter the titles to appear on the graph.

Graph Title:  Trailing Edge Software Company
Vertical axis label:  Sales in $1000s
Horizontal axis label:  Quarter

FIGURE 3. The MacGraph label-editing screen

Producing graphs

Now that you've entered the numbers and titles, you're ready to produce graphs. The Graph menu offers the following four options:

- A line graph, which plots the data points at the appropriate vertical and horizontal coordinates and connects them with a line (see Figure 4). Note that MacGraph simply numbers the horizontal axis from 1 to the total number of data points (in our example, 4 for the 4 quarters). The numbers along the vertical axis are automatically calculated to fit the range of data values provided. We'll show you how this works later.

- A bar graph, with the data represented as vertical bars equally spaced along the horizontal axis. The same numbers and titles used for line graphs are also used with bar graphs, as you can see in Figure 5.
FIGURE 4. A line MacGraph

FIGURE 5. A bar MacGraph
A pie graph, with the slices labeled with percentages based on the data points (see Figure 6). The first data point starts at the "twelve o'clock" position, and the slices are read clockwise. Note that on this and the other pie graph only the graph title appears, since there are no axes. The data values are normalized so the slices add up to 360 degrees. As you may have noticed in our example, percentages may not add up to exactly 100 percent, due to rounding each percentage to a whole number.

A pie graph, with the actual data values instead of percentages (see Figure 7). Other than that, it's the same as the preceding graph.

To produce a graph, simply choose the appropriate option from the Graph menu. If you haven't either entered or loaded some data, a message will inform you that there is no data; otherwise your graph will appear. You can then use the normal Macintosh screen-printing functions to make a hard copy of your graph (Command-Shift-4) or save a screen image to disk (Command-Shift-3).
Saving graphs to the Clipboard

If you want to combine your graph with another document or to use another application to add text and/or visual embellishments to your graph, simply choose Graph -> Clipboard from the MacGraph File menu. The graph currently shown on the screen will then be copied to the Clipboard, where it can be pasted into documents created with other applications.

Because a graph can be a little larger than the MacPaint window, part of the image on the Clipboard may be eliminated if it is pasted into a MacPaint document. You can get around this by using the key sequence Command-Shift-3 to save an image of the screen on disk as a MacPaint document. You can then open the document in MacPaint and clean it up, making whatever additional changes you desire.

The MacGraph File menu also provides the standard options found in most Midnight Madness programs: Quit to BASIC, Quit to the Mac Desktop, and Instructions.
How the program works

MacGraph is entirely menu-driven, which makes its structure very easy to understand. An “idle loop” simply runs until interrupted by a menu selection. When studying the program listing, pay particular attention to the use of edit fields for handling entry of data and labels. If you haven’t used edit fields before, you’ll find that they make it very easy to get data from the user by providing editing facilities that would otherwise require many lines of code to implement. We’ll also cover the basics of charting data in the line, bar, and pie formats. A pseudocode description of MacGraph is shown in Figure 8.

**Initialization**
- initialize data array and constants
- define Output window
- create and enable MacGraph menus
- display title screen
- initialize data array (InitData)

**Main Program**
(endless loop interrupted by menu events)

**Menu Events**
- MenuEvent:
  - disable Graph and File menus
  - find number of menu selected
  - get number of item within menu
  - if Graph menu selected, draw pertinent graph (DrawGraph)
  - if not, process file menu item selected
  - re-enable menus

  **FOpen:** open a file and get data for editing
  - ask for file name
  - open secondary window (OpenWindow)
  - open file, read data
  - convert negative numbers read to positive
  - if too many numbers in file for data array, display message
  - close window (CloseWindow)

  **SaveAs:**
  - ask for file name for saving data
  - save current data in file

  **EditData:** edit the data list
  - open secondary window (OpenWindow)
  - print instructions
  - calculate locations of edit fields and open them with data
  - handle editing in fields (EditFields)
  - convert string data in fields to numbers for data array
  - close window (CloseWindow)

**Figure 8. Program outline**
MAC GRAPH

Titles: edit graph titles
    open secondary window (OpenWindow)
    display instructions
    open edit fields
    handle editing in fields (EditFields)
    save graph title names
    close window (CloseWindow)

ClipGraph: copy current graph to Clipboard
    save picture string commands
    open and save picture string to Clipboard

Inst: display instructions
    open secondary window (OpenWindow)
    display instructions
    create button and wait for click to continue
    close window (CloseWindow)

BasQuit: quit to BASIC

DeskQuit: quit to Mac Desktop

DrawGraph:
    if no data in array, print message (NoData)
    on menu event in Graph menu, branch to appropriate subroutine:

LineGraph:
    draw graph frame (DrawFrame)
    draw line graph
    print number on axis if there's room (DrawYTick)

BarGraph:
    draw graph frame (DrawFrame)
    draw bar graph
    print number on axis if there's room (DrawYTick)

PiePercent:
    set flag for percentage labels
    draw pie chart (PieGraph)

PieIndex:
    set flag for value labels
    draw pie chart (PieGraph)

Subroutines

NoData: display no data message

PieGraph:
    define circle parameters and draw circle
    find total range of data for determining relative proportions of pie
    draw pie wedges, label for percentage or value (DrawStr)

Figure 8. Program outline (continued)
DrawFrame: draw and label graph axes
  find range of data and graph step (CalcMax)
  calculate scale, draw ticks on Y axis (DrawYTicks)
  print graph, vertical and horizontal titles (DrawStr)

CalcMax: calculate maximum value for scaling graph

DrawYticks: draw ticks and numbers up Y axis

EditFields: handle field editing events

InitData: initialize data array to blanks

OpenWindow: open secondary window

CloseWindow: close secondary window

Subprograms
  DrawXNum: print axis number
  DrawStr: print string centered relative to current pen position

Figure 8. Program outline (continued)

Initialization and menu setup

In [Init 1] two arrays are dimensioned to hold the edited data and titles. Although the program could be modified to accommodate a larger number of data points, there is a limit to the amount of information that can be shown on one graph. If you are doing mathematical graphing (plotting equations), the graph becomes more accurate as data points are added. MacGraph, however, is designed for business or other descriptive graphics, and for this purpose 40 data points are an adequate limit. Similarly, the three titles (or labels) allowed by MacGraph—the graph title and one label for each axis—will usually be enough. However, as noted, you can transfer the graph to MacPaint and add as many titles or labels as you wish.

The coordinates for a multipurpose, secondary window (wind2) are also specified in this part of the code. This window will be used in the data and title editors, for displaying data as it is being read from a file, and elsewhere. In MacGraph, we deal with two sorts of geometry—rectangular and circular—and the boundaries for both are defined next. First, minimum and maximum x and y coordinates are specified, as well as the total x and y range; these will be used as maximum boundaries for the line and bar graphs. Then the maximum circular boundaries and radius for
the pie graphs are defined. \( \pi \) is defined as a double-precision constant (note the "#" following the number). Making it double precision increases accuracy in drawing and proportioning the "slices" of pie graphs.

The variable \( gType \), initialized to 1, will hold the type of graph selected from the Graph menu.

The \texttt{Log10} function converts the "natural" logarithm function built into Microsoft BASIC to a base-10 logarithm suitable for scaling data so it will all fit on the graph; we'll discuss this later. After this function definition comes our standard statement for setting the size for the full-screen MacGraph window.

In [Init 2], the two menus, MacGraph File and Graph, are set up. The "-" menu option specified before the quit options on the File menu actually prints a line of dashes that separate the "working" options from the quit options. When you have a lot of options on one menu, such divisions help break up the options into related groups. Note the final 0 in the menu specification for this item: This is necessary to make the dividing line inactive (not selectable as an option). Menu event-handling is then set up, with MenuEvent \([\text{ME A0}]\) as the handler.

The title screen is drawn in [Init 3]. This screen is very simple and serves primarily to tell the user that MacGraph is now ready and that the two special menus are active. Following the drawing of the title screen, 
\texttt{InitData [SR 7]} is called to initialize the elements of the data array to \(-1\) (the \( gData \) array end-of-data marker). This makes sure that the array will be read as containing no data until data is actually entered by the user.

As previously noted, the main program \([\text{Main}]\) is an endless loop whose only purpose in life is to keep BASIC busy between menu events.

**Handling menu events**

The first thing MenuEvent \([\text{ME A0}]\) does is deactivate both menus to prevent further menu events from interrupting processing of the currently selected option. The BASIC \texttt{MENU(0)} function then returns the number of the \texttt{menu} selected, which is assigned to the variable \texttt{menuBar}. The menu selected will be either 6 (MacGraph File) or 7 (Graph). The normal Apple and BASIC menus that are active while MacGraph is running don't concern us here, since they are automatically handled by their respective managers and thus don't result in control coming to this subroutine. (Since BASIC's File menu is inactive during operation of MacGraph, we always mean the special MacGraph File menu when we refer to the File menu in
the following sections.) The MENU(1) function is then used to obtain the number of the item selected within that menu, which is assigned (not surprisingly) to the variable item.

If menuBar isn't 7, an item on the File menu has been selected, and control goes to the label FileMenu and from there to the appropriate subroutine via the ON GOSUB statement. Since at least some of these options have to be used before any graph is plotted, let's go through the implementation of the File options first.

Opening a data file

FOpen [ME A1] is used to open a data file to obtain data for a graph. This data is then made available for editing in the data editor if desired, or the user can proceed directly to adding titles and producing graphs. The standard BASIC function FILES$ is used to obtain the file selection, which is specified as being a text file. This is the only data format that MacGraph can handle, so it is the only type of file shown in the file-selection window.

If a valid file selection is made, OpenWindow [SR 8] is called upon to open our predefined multipurpose window. Here, the window is used to display the data numbers as they are read in. After returning to [ME A1], a message telling the user that data is being read in (and the name of the file it is being read from) is displayed, and the WHILE NOT EOF loop is entered. The variable i is used to keep a count of the number of data items read. After 1 is added to this count, a number is read in and assigned to the variable j. An inner box is drawn with the LINE statement with the bf option, using white (30) to clear part of the screen for messages. The value of the counter i is checked: If it exceeds the maximum number of data items (maxData), a message that the data file is too large is printed, and control goes to the label TooLarge. Once this happens, the same thing will happen for any further numbers until the end of the file is reached. None of the excess numbers will be added to the data array (gData).

The value of the number itself (j) is also checked. If it is negative, it is changed to positive with the absolute value (ABS) function, and a message to this effect is displayed. A negative number may mark the end of data if the user entered it for that purpose when editing data, but EOF becoming true is what actually terminates the data-reading loop in this subroutine. In any case, negative numbers are changed to positive, since the graphs have only positive coordinates. The number is assigned to the next element of the gData array and printed on the screen.
The variable $dNum$ (the number of data items read) is set to either the value of the counter when the WHILE loop finished, or the maximum number of data items ($maxData$, set to 40)—whichever is smaller. Then a message telling you how many items were read is displayed, and a WHILE...WEND loop is used to wait until the OK button included with the window is clicked. Since there’s only one button, the loop simply checks for DIALOG(0) becoming 1, indicating a button click in the active window. When this happens, CloseWindow [SR 9] is called to do just that. Note the CLS (clear screen) statement used to clear out anything remaining in the primary MacGraph window.

**Saving a data file**

SaveAs [ME A2] gets a file name via the ubiquitous FILES$ function (remember to use a 0 with FILES$ when you want to get a file name to save a file, and use a 1 when you want to get a file name for opening a file). If a valid name is obtained, the elements of the data array $gData$ are simply printed to the file. There is no need to check for an excess number of items here, since the program makes sure that no more than $maxData$ items are accepted when data is first read into $gData$ (see the previous section) or when it is edited (see the next section).

**Editing data**

The Edit Data option, handled by EditData [ME A3], is used to edit the data list (add and/or change data). The data may be read from a file or entered at the keyboard. The secondary window is opened, as described for the option to open a file, and the top and left boundaries of the editing grid are defined. A reminder message about ending the plotting sequence is displayed, and the flag $dataEnd$ is set to $false$.

A FOR...NEXT loop is used to position, display, and edit the data. The position for each edit field (which will be used to edit one data item) is first calculated. The offset from the left boundary ($leftEd$), assigned to the variable $l$, is the item number, $n$, minus 1 (since items start with 1) times the width of an edit-field box (70) MOD 5. The MOD is used because there are five items per row: Since we’re interested here in how far to the right we’re going, not how far down, we want the number of boxes “left over” (the remainder) when complete groups of five boxes have been accounted for. The offset from the top boundary ($topEd$), assigned to the variable $t$, is equal to the item number minus 1 divided by 5 (since we need to know how many complete rows to go down), times 20 (the height of an edit
Note this general procedure and its use of MOD and division: It allows you to draw and position any regular rectangular grid of objects.

After the position of each item is determined using these calculations, we need to know whether we’ve reached the beginning of a new line. If we have, we want to print one of the reference numbers (1, 6, 11, and so on) that run down the left side of the grid. If \( n - 1 \mod 5 \) is 0, we’re at the start of a new line, and the current item number \( n \) is printed in large type, located in and slightly down from the position of the first edit field in the new line. (Remember that the coordinates of the current edit field are \((l,t)\).) Control then continues at the label NotLine, where of course we also end up if the current edit field didn’t start a new line.

The current data item \( (gData(n)) \) is checked to see if it’s negative. If it is, we assume the user wanted to end the data there, and the flag Data-End is set to true. If the item marks the end of data, nothing (a null string) is printed; otherwise, the item is converted to a string. This is necessary so it can be edited with the EDIT FIELD statement. The EDIT FIELD statement contains the following specifications:

- The edit field ID, which in this case is the item number \( (n) \).
- The string expression to be edited \( (n$\), which we converted from \( gData(n) \).
- The specifications for the rectangle within which editing will take place. These are in the standard BASIC \((x,y) - (x,y)\) format, where the first pair is the upper left coordinate and the second pair is the lower right coordinate. In this case, the specifications are offset from the coordinates of the field position for the current item \((l,t)\) as previously calculated.
- The next specification, for which we put a comma (indicating the default), draws a box around the rectangle defining the edit field. It also causes the insertion point to move to the next field if the Return or Tab key is pressed.
- Finally, the parameter 2 is used to specify center justification, making user-entered numbers automatically centered in the edit fields.

If you haven’t used edit fields before, they’re worth discovering in more detail than we can offer here, and Chapter 14 of Macinations (Microsoft Press, 1985), would be a good place to start.

Once defined, all edit fields will remain active until the window is closed. Having defined an edit field for the current item, the loop continues positioning, defining, and displaying the edit fields until all 40 fields have been set up. The EDIT FIELD 1 statement that follows makes the first field the current edit field and displays the insertion point, ready for the
user to begin typing. The variable \textit{eField}, holding the number of the edit field currently in use, is set to 1, and the variable \textit{maxFields} (maximum number of fields to edit) is set to \textit{maxData}. This variable is used so that the same subroutine can be used to edit the title fields, which we'll discuss in the next section.

\textbf{EditFields [SR 6]} is used to actually manage editing the data. It sets the flag OK to \textit{false}: This flag will become \textit{true} when the user clicks on the OK button, setting DIALOG(1) to 1. The WHILE...WEND loop starts by getting the value of DIALOG(0) and assigning it to the variable \textit{diag}. If \textit{diag} = 1, OK is set to \textit{true}, the loop will terminate, and the subroutine will be exited on the next pass. If \textit{diag} = 2, the user has moved to a new edit field by clicking in that field with the mouse pointer. In this case, the current field to be edited (\textit{eField}) is set to DIALOG(2), because this dialog function is set up precisely to return the value of the current edit field. Besides using a mouse click, the user can move to a new edit field by hitting either the Tab or Return key, so we must check for these keys. If \textit{diag} is not 7 or 6, neither Tab nor Return was pressed by the user; control goes to the label NotTab, and nothing further need be done. Otherwise, the number of the new edit field is calculated: Using the MOD function allows the insertion point to wrap around back to the first field if it was in the last field when the key was pressed. The EDIT FIELD statement is then used to edit the new field in the manner discussed earlier.

Once the user's editing activity is completed, control returns to EditData. A loop is now used to go backwards through the edit fields to get the edited values and update the \textit{gData} array. The loop also determines where the end of data is and thus finds the total number of data items. The string function \textbf{EDIT$} is used to access the set of strings from all the edit fields in use. The string from the current field number, \textit{n}, which starts at \textit{maxData} (40 in the case of data fields), is assigned to \textit{j$} and checked to see if it is null (blank). If it is, there is no data (it may be the end-of-data marker, or there may be more blank, null, or \textit{−1} fields to go), and the string “\textit{−1}” is assigned. The \textbf{VAL} function is then used to get the numeric value of the current string. If it is negative (either because it was a null or blank or because it actually had a negative number in it), there can be no more than the current number of data fields, so \textit{dNum}, the total number of data fields, is reduced by 1. The value is then put into the current element of \textit{gData}. When the FOR...NEXT loop is finished, \textit{dNum} contains the number of items read, and \textit{gData} contains the numeric values of the items. The data-editing window is closed, and the option has been completed.
**Editing the graph titles**

Titles [ME A4] is used to edit the titles to be used on graphs generated from the current data set. The procedure followed here is very similar to that used in the data editor we’ve just discussed. As with that subroutine, OpenWindow is called to open the multipurpose window, and the top and left boundaries of the edit area are defined. A prompting message is then displayed via the subprogram DrawStr [SP 2]. This is used here, rather than having just a PRINT statement as in the case of the data editor, because it centers the string before printing it. As is commonly the case in our programs, formatting routines like DrawStr are often coded as subprograms rather than subroutines because the subprogram can have any string passed directly to it as a parameter rather than first having to assign the string to an agreed-upon variable.

The descriptions of the three edit fields are printed with a regular combination of MOVETO and PRINT statements because they don’t have to be centered. The three edit fields are then set up by a FOR...NEXT loop. Again, they are set up by offsetting them from the top and left boundaries. EDIT FIELD is used again, too: Since there are only three edit fields here and they’re directly underneath each other, the setup is much simpler than it was for the data edit fields. The array `title$` will hold the edited titles.

After `maxFields` is set to 3, EditFields [SR 6] is called to edit the fields. Note that the same subroutine was used to edit the data fields, but `maxFields` had to be changed before calling it in Titles. The flexibility of the BASIC edit-field facility allows us to get away with using one subroutine for both data and title editing. The final FOR...NEXT loop simply reads the contents of EDIT$ (where BASIC has put the results of the editing) into the corresponding elements of `title$`. CloseWindow is then called to close the window and clear the screen.

**Copying a graph to the Clipboard**

ClipGraph [ME A5] copies the currently drawn graph to the Clipboard so it can be transferred to other applications. The variable `dNum` is first checked to make sure there is some data currently in use from which to draw the graph. If no data is found, NoData [SR 1] is called to prompt the user to enter some. If data is present, we move to the PICTURE ON statement. This statement enables BASIC to record the graphics instructions that will be used to draw the graph in a string that can later be returned by the PICTURE$ function. The CALL SHOWPEN statement turns
the pen back on so that the graph will be drawn on the screen as well as being recorded. (The default with PICTURE ON is for the pen to be off and thus display no output to the screen.)

DrawGraph [ME B0] is called to select the type of graph to be drawn. It clears the screen and checks the value of \( d\text{Num} \) to make sure some data is present, prompting the user again via NoData if it cannot find data. The type of graph to be drawn is stored in the variable \( g\text{Type} \): It will be whatever was the last selection from the Graph menu. (We will discuss that menu and the actual mechanics of drawing the graphs in a later section.) After the graph is drawn, control returns to DrawGraph; the drawn graph is now stored in the picture string. The pen is turned back off, PICTURE OFF is issued to stop the recording of images, and the file Clip:Picture is opened for output. Remember that Clip is the standard Macintosh file name for the Clipboard and that the extension :Picture is needed when picture information is to be written to the Clipboard. The information in PICTURES$ is printed, and this option is complete.

**Other file-menu options**

We have met all the other options before; they are standard for our programs. Inst [ME A6] displays brief instructions, and BasQuit [ME A7] and DeskQuit [ME A8] return control to BASIC and the Mac desktop. When you use the quit options in your own programs, don't forget that if you've created any special-purpose menus, you'll want to include the MENU RESET statement to remove them, as we do here.

**Drawing the graphs**

As we've mentioned, DrawGraph [ME B0] is the point of departure for drawing the graphs. If MenuEvent determines that a selection was made from the Graph menu, the item number selected is stored in the variable \( g\text{Type} \). We have also noted that if the File menu-item Graph -> Clip is selected, DrawGraph is called to draw the graph currently specified in \( g\text{Type} \). No matter how it's called, DrawGraph clears the screen, checks to make sure there is data from which to draw the graph \( (d\text{Num} <> 0) \), and prompts the user via NoData if there is not, and then (assuming that data is present) branches to the appropriate graph routine via the ON \( g\text{Type} \) GOSUB statement.

Note that there are basically two geometries involved in our graphs: lines and rectangles, and circles. Both types involve the process of scaling the data based on the largest and smallest data values found,
and setting boundaries for drawing the graph, but the actual drawing of
the circular (pie) graphs is more complicated because of the trigonometry
involved in determining sections of a circle. Let's start by examining the
simplest case: the line graph.

The line graph

LineGraph [ME B1] handles the line graph. This graph requires a
rectangular frame with "ticks" marking regular graduations of values
along the vertical and horizontal axes. DrawFrame [SR 3] is called to pro-
duce that frame. It starts by finding the maximum value in the set of values
in the data array gData. To do this, it simply starts with max set to 0, then
uses a FOR...NEXT loop to compare each data value in turn to the current
maximum and replaces the current maximum with the data value if the
latter is larger.

DrawFrame in turn calls CalcMax [SR 4] to "scale" the data
values so that both the smallest and the largest values will fit on the graph.
The function Log10, defined back in [Init 1], finds the base-10 logarithm of
a number; that is, the power to which 10 would have to be raised to get that
number. (To be precise, it finds the highest integer power.) Microsoft
BASIC doesn't provide this kind of log directly, but it does provide a built-in
function for the "natural" logarithm. The base-10 logarithm of a num-
ber can be found by dividing the natural log of the number by the natural
logarithm of 10. You might file this little fact away so you'll be able to get
base-10 logs on the (admittedly infrequent) occasions when you might
need them in your BASIC programming.

The first statement in CalcMax sets the variable t to the integer
power of 10 closest to the maximum data value. This is our first rough ap-
proximation: In our example, where the largest data value is 348 (see Fig-
ure 2), t would be set to 10 to the second power, or 100. (Ten to the third,
the next power, is 1000, which is too large.) The variable m is initially set
to the value of t, and then a WHILE...WEND loop is used to repeatedly
add t to m until it exceeds the maximum data value stored in max. In our
example, we add 100 to m (which starts at 100) until we get 400; at this
point max, or 348, is exceeded, so the loop stops.

An IF statement next tests to see if max is less than m * .6; we're
now checking to see if we overshot and, if so, whether we've overshot
enough that we want to start moving m (our tentative maximum) back to-
w ard max. If so, control branches to the label CM1, t is divided by 10 (in
effect giving a finer scale to work with), and a WHILE...WEND loop sub-
tracts 2 * t from m as long as m is greater than max. Since "one too many"
subtractions will have been made when the loop is exited, \(2 \cdot t\) is added back to \(m\) to ensure that \(m\) will be larger than the largest data value (and thus that the data value will be within the graph). We then reach the label CM3, where \(max\) is set to \(m\).

To use an actual example, the largest data value, 348, is larger than \(m \cdot .6\), or 240 (400 \(\cdot .6\)), so \(m\) may not be far enough past \(max\) and the graph may thus be too crowded. There might be room to increase \(m\) by \(t\), adding one more interval. The IF statement increases \(m\) by \(t\) if \(m\) isn't already approximately twice \(t\), or a multiple of twice \(t\) (four times \(t\), or whatever). In our example, where \(m\) is 400 and \(t\) is 100, \(m / t = 4\), and \(4 \mod 2 = 0\), so \(m\) stays at 400. Control then jumps to CM3, where \(max\) is also set to 400. If you check Figure 4, you can verify that 400 is the value of the top line of the graph.

After CalcMax is finished, control returns to DrawFrame, and the scale (the distance that will represent one unit of value along the y or vertical axis) is set by dividing the maximum possible extent of the y axis on the screen, \(yRange\), by the \(max\) value calculated in CalcMax (remember, this is no longer the maximum data value; it is the largest value that would fit on the graph). If you examine the definitions back in [Init 1], you can see that \(yRange\) is equal to \(maxY - minY - 5\): in this instance, 225 - 25 - 5, or 195. Since \(max\) has been set to 400, the scale is 195 / 400, or a little less than 0.5. So, using our example values, a point on the line graph (or the top of a bar on the bar graph) will be approximately half its value in pixels from the bottom of the graph.

DrawYTicks [SR 5] is now called to draw the "ticks" or numbered hatch marks that provide reference points for the values along the y axis. The subroutine first calculates the distance between hatch marks on the y axis. The starting value, \(interval\), is again the largest power of 10 that is less than \(max\). \(max\) is then divided by \(interval\): If the result is less than 3, the interval is too large and not enough hatch marks and numbers will be drawn. In this case \(interval\) is set to \(interval / 5\).

Now it's time to draw the tick marks and the numbers on the y axis. The FOR...NEXT loop will be executed one plus the maximum divided by the interval (rounded up) times. If you look back at Figure 4, you can see that there are five tick marks, numbered from 0 to 400, and 400 / 100 (or 4) + 1 is indeed 5. (We need the extra 1 because we start at 0.) We multiply \(n\) (the tick number) times the interval and then convert it to pixels by multiplying it by the scale. Since the y-axis screen coordinates \(decrease\) as we move up the axis, the quantity we've calculated is subtracted from \(maxY\) to get the y pixel coordinate. The LINE statement
draws a horizontal line from 4 less than \( minX \) to \( maxX \); the fact that the line starts to the left of the y axis provides the “tick mark.”

We now want to convert the number for each tick mark \((n * interval)\) to a string for easy formatting and proper printing. This is done with the STRS function, and the result is assigned to \( tS \). We specify the INT of this number if the interval is over 1 in order to get rid of fractions that might have crept in when we divided \( interval \) by 5. Floating-point operations tend to produce values that aren’t quite integers, and INT is good insurance against such mishaps when it’s important to have whole-number results. The MOVETO statement positions the pen far enough to the left of the y axis \((min X)\) to accommodate the length of the string in pixels, and the string is then printed. When all the tick marks, lines, and numbers have been printed, the loop is exited and control returns to DrawFrame.

DrawFrame calls DrawStr [SP 2], which centers and draws the first title \((in titleS(1))\)—the title of the graph itself. The second title, which is displayed vertically along the y axis, is set up by finding the midpoint of the y axis \((min Y + (max Y - min Y) / 2)\) and then subtracting from it the length of the string in characters times the character size in pixels. This gives the starting point for displaying the string, which is then printed a character at a time by a FOR...NEXT loop that positions each character. The third title is simply displayed below the x axis by another call to DrawStr. Note that the numbers along the x axis haven’t been printed yet; they will be produced as part of the actual graphing routine.

Finished at last, DrawFrame returns to LineGraph, and we’re ready to draw the actual graph. First, the lengths of the line segments connecting the data points are determined; this is also the interval at which the tick marks and numbers along the x axis will be shown. Assuming there is at least one data value \((dNum > 1)\), the length of a segment is the total length of the x axis \((xRange, determined back in [Init 1])\) divided by the number of data points minus one. The “minus one” is there because it takes only \( dNum - 1 \) lines to connect \( dNum \) points; there are three line segments connecting the four data points in our example.

After the pen has been positioned at the graph’s origin point \((min X, max Y)\)—remember that \( max Y \) is the bottom of the y axis—a FOR...NEXT loop is used to draw the graph. First, the y and x coordinates for the current data point are calculated. The y coordinate is determined by multiplying the scale (determined back in DrawFrame) by the value of the current data point and then subtracting the result from the y coordinate of the last point determined. (For the first segment, the “last point” is the origin point.)
The x coordinate is determined by multiplying the line length by one less than the number of the current data point and then adding it to the x coordinate of the last point, since x coordinates increase along the graph. (Again, the “one less” is necessary because the first point is plotted on the y axis; that is, with an x coordinate of 0.) The second point is plotted with an x coordinate of \( minX + \text{lineLen} \), the third point at \( minX + \text{lineLen} \times 2 \), and so on. A \text{LINETO} statement is then used to draw the line segment and a \text{CIRCLE} statement to draw the black circle marking the data point’s location. The effect of a filled black circle is obtained by drawing first a circle with radius 2 and then a circle with radius 1 within it.

Next, the corresponding tick mark along the x axis is drawn. The IF statement with MOD 5 is used to draw every fifth line twice as long for easy reference. The \text{DrawXNum} subprogram [SP 1] is called to display the reference numbers along the x axis. This subprogram takes into account a potential problem: What if you have a lot of data points? If you do, there will be too many corresponding numbers to fit along the x axis, or at least the numbers are likely to crowd and run into each other. To handle this situation, \text{DrawXNum} first checks to see if the total of the numbers it is being asked to draw is more than 20. If so, an AND with the MOD function makes it draw the number for only every fifth data point. (This doesn’t happen in our example, since it has only four data points.) The calling routine is responsible for passing this maximum number to \text{DrawXNum}; this number is available as \( dNum \). \text{DrawXNum} in turn calls upon \text{DrawStr} to actually draw the number, and \text{DrawStr} provides the centering. Finally, LineGraph moves the pen to the new (x,y) point, which becomes the origin for the next line segment. When the FOR loop has gone through every value of \( n \), this has been completed.

The bar graph

\text{BarGraph} [ME B2] supervises the drawing of this graph. You’ll be pleased to learn that most of the procedures involved in creating the bar graph have been covered in the preceding section. Again, the main graph subroutine calls \text{DrawFrame}, which then calculates the scale and calls \text{DrawYTicks} to draw the ticks and reference numbers for the y axis, exactly as in the case of the line graph.

After control returns to \text{BarGraph}, the width of the bars is calculated: It’s simply the extent of the x axis (\( xRange \)) divided by the number of data points (\( dNum \)). A \text{FOR}...\text{NEXT} loop draws the bars and the x axis ticks and numbers, completing the graph. For each data point, the top of the bar (its y coordinate) is calculated in exactly the same way that the y
The location of a point on the line graph was calculated. The x coordinate of the bar is calculated the same way as the x coordinate of a point on the line graph was calculated, with the difference that the bar width rather than line length is used. The bar is then drawn with a simple LINE statement with the \texttt{bf} option. The bar is drawn to slightly less than the full bar width \((\texttt{barWidth} - 5)\) so that some white space will be left to separate the bars. The midpoint of the bar on the x axis is determined, and the tick mark is drawn on the x axis. As with the line graph, every fifth tick mark is drawn at double length. Again, DrawXNum is called to draw the numbers.

\textbf{The pie graphs}

Although there are two pie graph options (pie with percentages and pie with values), the only difference between the two is in the numbers used to label the slices. If pie with percent is chosen from the Graph menu, \texttt{PiePercent [ME B3]} is called. The subroutine sets the Boolean flag variable \texttt{percent} to \texttt{true}, then calls \texttt{PieGraph [SR 2]} to draw the graph. If pie with actual values (rather than percentages) is chosen from the Graph menu, \texttt{PieIndex [ME B4]} is called. This subroutine sets the \texttt{percent} flag to \texttt{false} and then calls \texttt{PieGraph}.

You might wonder why two subroutines are used whose only difference is in how they set a flag. We could have one subroutine (called simply \texttt{Pie}, perhaps) appear in both pie “slots” in the \texttt{ON GOSUB} statement in \texttt{DrawGraph}. The subroutine could set the \texttt{percent} flag according to the value of \texttt{gType}, the type of graph selected. This would certainly work, and it would shorten the code. However, it has the disadvantage that if someone wanted to add features to one kind of pie graph but not the other, more complicated logic would have to be put into the “general” \texttt{Pie} subroutine. With separate subroutines for the two kinds of pie, features can be added to either without affecting the other. Sometimes the “advantage” of saving a few lines of code is outweighed by the flexibility built into a clean design.

Now let’s look at the way the pie graph is actually drawn by \texttt{PieGraph}. To do this, we must turn our thinking from the rectangles and straight lines of the bar and line graphs to the world of circles and arcs. That means trigonometry—not everyone’s favorite pastime, we’ll admit. Remember, though, that you don’t really have to understand trig to use and even modify our pie-graphing routine. Since this isn’t a mathematics textbook, we’re not going to go heavily into detail, anyway.

The first step is quite simple. Using the boundaries for the circular chart area originally defined in \texttt{[Init 1]}, \texttt{PieGraph} establishes a standard
QuickDraw rectangle and uses a PAINTOVAL statement to draw a filled circle—the pie. Next, a FOR ... NEXT loop adds up all the data values in gData. This value, totalUnits, is divided into 2 * pi. This latter quantity, you may remember from your geometry class, is the equivalent of the circle's 360 degrees expressed in radians, a standard measure of angle. The unit of angle thus obtained is similar to the scale calculated for the line and bar charts: It's the angle that will represent one unit of value in the data. Next, the graph title is positioned 30 pixels above the top of the circle and drawn by a call to DrawStr. Note that (circleX,circleY) are the screen coordinates of the center of the circle, and that circleR is the radius of the circle. For pie graphs, only the first title (the overall graph title) is used, since these graphs aren't laid out on x and y axes like the line and bar charts.

Now it's time to draw the segments or "slices" of the pie. The basic method for finding the endpoint to which a given segment will be drawn uses the following rule of trigonometry: Starting from the center of the circle, and given an angle theta, in radians:

X coordinate of endpoint = centerX + circleR * SIN(theta)
Y coordinate of endpoint = centerY - circleR * COS(theta)

We already know the coordinates of the center of our circle (circleX,circleY), and its radius (circleR). The variable theta is used to keep track of the angle, which of course increases as data points are plotted as we move clockwise around the circle. The angle is initially set to 0, since the pie graph by convention starts at the twelve o'clock position. If you compare the general formula just described with the the first two statements inside the main FOR...NEXT loop, you'll see how MacGraph uses this formula to calculate an endpoint (endPtX,endPtY) each time through the loop. Again, the x coordinate is added to that of the center of the circle, and the y coordinate is subtracted from the center's y coordinate, because on the Mac screen y coordinate values on a circle decrease when x coordinates increase, and vice versa. (Our graph corresponds to the upper right quadrant of the traditional Cartesian grid you may remember pondering in algebra class.)

A LINE statement now draws a white line from the center of the circle to the endpoint. The next item on the agenda is to label the segment with a number, either a percentage or a value depending on the type of pie graph being generated. The same formula as just described is used, except that we use an angle, midTheta, which is calculated by multiplying the unit
angle by half the data value being plotted and adding it to the current value of theta. Since we want the number to appear exactly halfway along the arc of the pie slice, we want an angle that points at that midpoint. Also, since the numbers are printed a little way outside the circle, the radius of the circle has to be multiplied by 1.15 to get an “invisible circle” large enough for the numbers to be placed on it. Two IF statements check whether percents are wanted and formulate the string to be printed accordingly. DrawStr is called to draw the number. When the FOR...NEXT loop is exited, the complete graph has been drawn.

Suggestions for MacGraph

You can begin by using MacGraph to plot whatever kinds of data are relevant to your business, profession, or studies. Since MacGraph is quick and easy to use, you can readily translate many kinds of statistics into a visual form that’s usually more comprehensible than raw numbers. You can copy some graphs to the Clipboard and incorporate them into documents produced on your word processor, or save a screen image to disk and embellish it with MacPaint.

After you’ve experimented with MacGraph for a while, you may want to add features to it. An easy one would be to have MacGraph save the current titles along with the data in a disk file. You could also add a routine (or write a separate BASIC program) that will “capture” a graph and shrink it, or perhaps draw several graphs on the same screen. Of course, you can also add a second line to the graph title, change the pie to a white pie with black lines, or make many other modifications.

Since the structure of wholly menu-driven programs like MacGraph is simple, it’s easy to add or change one or more options without affecting the others. A bakery might provide fancier (and more expensive) pies, but with MacGraph you can bake them to your own recipes!
MacGraph program listing

MacGraph
Graph data on the screen

shasMaxData = 40
DIM gData(maxData), title$(3)
pi = 3.1415926#
wind2L = 16 : wind2T = 55 : wind2W = 479 : wind2H = 270
minX = 70 : maxX = 450 : xRange = maxX - minX
minY = 25 : maxY = 225 : yRange = maxY - minY - 5
circleX = 250 : circleY = 150 : circleR = 110
false= 0 : true = NOT false
gType = 1

DEF FN Log10(x) = LOG(x) / LOG(10)

WINDOW 1, , (12,34) - (500,330), 2

MENU 6, 0, 1, "MacGraph File"
MENU 6, 1, 1, "Open"
MENU 6, 2, 1, "Save As"
MENU 6, 3, 1, "Edit Data"
MENU 6, 4, 1, "Edit Titles"
MENU 6, 5, 1, "Graph -> Clipboard"
MENU 6, 6, 1, "Instructions"
MENU 6, 7, 0, "<";
MENU 6, 8, 1, "Quit to BASIC"
MENU 6, 9, 1, "Quit to Desktop"

MENU 7, 0, 1, "Graph"
MENU 7, 1, 1, "Line"
MENU 7, 2, 1, "Bar"
MENU 7, 3, 1, "Pie w/Percent"
MENU 7, 4, 1, "Pie w/Value"

ON MENU GOSUB MenuEvent : MENU ON

CALL TEXTSIZE(36): CALL TEXTFONT(0)
LOCATE 2, 7 : PRINT "MacGraph!"
LOCATE 3, 6
CALL TEXTSIZE(12)
PRINT "The MacGraph menus are now active."
CALL TEXTFONT(1) : CALL TEXTMODE(1)
GOSUB InitData

WHILE true : WEND
**MENU EVENTS**

MenuEvent:
```
MENU 6, 0, 0 : MENU 7, 0, 0
menuItemBar = MENU(0)
menuItem = MENU(1)
IF menuItemBar <> 7 THEN FileMenu
gType = menuItem
GOSUB DrawGraph
GOTO MEventEnd
```

FileMenu:
```
ON item GOSUB FOpen,SaveAs,EditData,Titles,ClipGraph,Inst,,BasQuit,DeskQuit
MEventEnd:
```
```
MENU 6, 0, 1 : MENU 7, 0, 1
RETURN
```

FOpen:
```
open file and read from it
fileName$ = FILES$(1,"TEXT")
IF fileName$ = "" THEN CLS : RETURN
GOSUB OpenWindow
CALL TEXTFONT(0)
CALL MOVETO(108,50) : PRINT "Reading data from: "; fileName$
OPEN fileName$ FOR INPUT AS #1
i = 0
WHILE NOT EOF(1)
i = i + 1
INPUT #1, j
LINE (100,55) - (520,75), 30, bf
CALL MOVETO(100,70)
IF i > maxData THEN PRINT " Data file too large! "; GOTO TooLarge
IF j < 0 THEN PRINT " Negative number changed to positive. "; j = ABS(j)
gData(i) = j
PRINT gData(i);
TooLarge:
```
```
WEND
IF i > maxData THEN i = maxData
dNum = i
CLOSE #1
```
```
CALL MOVETO(100,90) : PRINT dNum; " numbers found."
WHILE DIALOG(0) <> 1 : WEND
CALL TEXTFONT(1)
GOSUB CloseWindow
CLS
RETURN
```

SaveAs:
```
save list to file
file$ = FILES$(0,"File Name?")
IF file$ = "" THEN NoSave
OPEN file$ FOR OUTPUT AS #1
FOR n = 1 TO dNum : PRINT #1, gData(n) : NEXT
CLOSE #1
NoSave:
CLS
RETURN

EditData: ________________________________ edit data list
GOSUB OpenWindow
topEd = 60 : leftEd = 30
CALL TEXTFONT(0)
x = 28 : y = 22
CALL MOVETO(x,y) : y = y + 16
PRINT "Edit the data to be plotted. Remember, a blank box"
CALL MOVETO(x,y) : y = y + 16
PRINT "or negative number ends the plotting sequence."
dataEnd = false
FOR n = 1 TO maxData
i = leftEd + INT((n - 1) MOD 5) * 70
t = topEd + INT(((n - 1) / 5)) * 20
IF (n - 1) MOD 5 <> 0 THEN NotLine
CALL TEXTFONT(0)
CALL MOVETO(i - 30, t + 13) : PRINT n;
CALL TEXTFONT(1)
NotLine:
IF gData(n) < 0 THEN dataEnd = true
IF dataEnd THEN n$ = "" ELSE n$ = STR$(gData(n))
EDIT FIELD n, n$, (l,t) - (l + 60, t + 15), , 2
NEXT n
EDIT FIELD 1 : eField = 1
maxFields = maxData
GOSUB EditFields
dNum = maxData
FOR n = maxData TO 1 STEP -1
j$ = EDIT$(n) : IF j$ = "" OR j$ = "" THEN j$ = "-1"
j = VAL(j$)
IF j < 0 THEN dNum = n - 1
gData(n) = j
NEXT n
GOSUB CloseWindow
RETURN

Titles: ________________________________ edit graph titles
GOSUB OpenWindow
topEd = 130 : leftEd = 200
CALL TEXTFONT(0)
CALL MOVETO(wind2W / 2, 70)
CALL DrawStr("Enter the titles to appear on the graph.")
x = 30 : y = topEd + 12
CALL MOVETO(x,y) : y = y + 20 : PRINT "Graph Title:";
CALL MOVETO(x,y) : y = y + 20 : PRINT "Vertical axis label:";
CALL MOVETO(x,y) : y = y + 20 : PRINT "Horizontal axis label:";
CALL TEXTFONT(1)
FOR n = 1 TO 3
  l = leftEd : t = topEd + (n - 1) * 20
  EDIT FIELD n, title$(n), (l,t) - (l + 250, t + 15) , 2
NEXT n
EDIT FIELD 1 : eField = 1
maxFields = 3
GOSUB EditFields
FOR n = 1 TO 3 : title$(n) = EDITS(n) : NEXT
GOSUB CloseWindow
RETURN

ClipGraph:
  copy graph picture to Clipboard
IF dNum = 0 THEN GOSUB NoData : GOTO ClipEnd
PICTURE ON
CALL SHOWPEN : GOSUB DrawGraph : CALL HIDEPEN
PICTURE OFF
OPEN "Clip:Picture" FOR OUTPUT AS #1
PRINT #1, PICTURES$;
CLOSE #1
ClipEnd:
  RETURN

Inst:
  display instructions
WINDOW 2, , (16,55) - (495,325), 2
PRINT "MacGraph Instructions"
PRINT "This program allows you to graph a list of positive numbers on the"
PRINT "Macintosh screen. It cannot plot negative numbers. The list can be either"
PRINT "read out of a file, or entered from the keyboard. If you read from a file, it"
PRINT "must be a text file with the numbers entered just as if you entered them"
PRINT "in response to an INPUT statement: As a series separated by commas, or a"
PRINT "list of numbers each on its own line. You can also edit or enter the"
PRINT "numbers using the edit boxes in the Data Edit section. End a list of"
PRINT "numbers by either a blank edit box or any negative number."
PRINT "Once you have entered a list of positive numbers, you can display the list"
PRINT "as either a line graph, bar graph, or pie graph by choosing the appropriate"
PRINT "item from the Graph menu. In the case of the pie graph, the data are scaled"
PRINT "so the sum of all the numbers equals 360 degrees."
CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(O) <> 1: WEND
BUTTON CLOSE 1 : WINDOW CLOSE 2
RETURN
MacQuit: quit to BASIC

End

DeskQuit: return to Mac Desktop

End

MenuReset

System

More menu events

DrawGraph: draw current graph

CLS

IF dNum = 0 THEN GOSUB NoData

IF dNum <> 0 THEN gType GOSUB LineGraph, BarGraph, PiePercent, PieIndex

RETURN

LineGraph: draw line line graph

GOSUB DrawFrame

IF dNum > 1 THEN lineLen = xRange / (dNum - 1) ELSE lineLen = 0

CALL MOVETO(minX, maxY)

FOR n = 1 TO dNum

pointY = maxY - scale * gData(n)

pointX = minX + lineLen * (n - 1)

CALL LINETO(pointX, pointY)

CIRCLE (pointX, pointY), 2 : CIRCLE (pointX, pointY), 1

size = 2 : IF (n MOD 5) = 0 THEN size = 4

LINE (pointX, maxY - size) - (pointX, maxY + size)

CALL DrawYNum(pointX, maxY + 10, n, dNum)

NEXT n

RETURN

BarGraph: draw bar graph

GOSUB DrawFrame

barWidth = xRange / dNum

FOR n = 1 TO dNum

barTop = maxY - gData(n) * scale

barX = (n - 1) * barWidth + minX

LINE (barX, maxY) - (barX + barWidth - 5, barTop), , bf

midPt = barX + (barWidth - 5) / 2

size = 2

IF (n MOD 5) = 0 THEN size = 4

LINE (midPt, maxY) - (midPt, maxY + size)

CALL DrawYNum(midPt, maxY + 10, n, dNum)

NEXT n

RETURN

PiePercent: draw percentage pie

percent = true

GOSUB PieGraph

RETURN
PieIndex:                       draw pie with values
percent = false                
GOSUB PieGraph                 RETURN

SUBROUTINES

NoData:                        tell user no data to plot
CLS                            
CALL TEXTFONT(0)               
CALL MOVETO(255,100)           
CALL DrawStr("No data to plot. Enter data with 'Edit Data' menu.")
CALL TEXTFONT(1)               RETURN

PieGraph:                      draw pie graph
r%(0) = circleY - circleR     : r%(2) = circleY + circleR
r%(1) = circleX - circleR     : r%(3) = circleX + circleR
CALL PAINTOVAL(VARPTR(r%(0)))
totaiUnits = 0 : theta = 0
FOR n = 1 TO dNum
    totaiUnits = totaiUnits + gData(n)
NEXT
unitAngle = (2 * pi) / totaiUnits
CALL MOVETO(circleX ,
            circleY- circleR- 30)
CALL TEXTFONT (0) : CALL DrawStr(title$(1))
CALL TEXTFONT (1)
FOR n = 1 TO dNum
    endPtX = circleX + circleR * SIN(theta)
    endPtY = circleY - circleR * COS(theta)
    LINE (circleX,circleY) - (endPtX,endPtY), 30
    midTheta = theta + (unitAngle * gData(n)) / 2
    midPtX = circleX + (circleR * 1.15) * SIN(midTheta)
    midPtY = circleY - (circleR * 1.15) * COS(midTheta)
    CALL MOVETO(midPtX,midPtY)
    IF NOT percent THEN t$ = STRS(gData(n))
    IF percent THEN t$ = STRS(CINT((gData(n) * 100) / totaiUnits)) + "%"
    CALL DrawStr(t$)
    theta = theta + (unitAngle * gData(n))
NEXT n
RETURN

DrawFrame:                     draw frame for bar or line graph
max = 0                        'first, find largest number
FOR n = 1 TO dNum
    IF max < gData(n) THEN max = gData(n)
NEXT n
GOSUB CalcMax
scale = yRange / max          'calculate scale
LINE (minX,minY) - (minX,maxY)
LINE (minX,maxY) - (maxX,maxY)
GOSUB DrawYTicks
CALL TEXTFONT(0)
CALL MOVETO(minX + (maxX - minX) / 2, minY - 15)
CALL DrawStr(title$(1))
y = minY + (maxY - minY) / 2 - INT(LEN(title$(2)) / 2) * 16
FOR n = 1 TO LEN(title$(2))
   CALL MOVETO(minX - 60, y) : PRINT MID$(title$(2), n, 1);
   y = y + 16
NEXT n
CALL MOVETO(minX + (maxX - minX) / 2, maxY + 42)
CALL DrawStr(title$(3))
CALL TEXTFONT(1)
RETURN

CalcMax: calculate max value for scaling graph
t = 10 ^ (INT(FN Log10(max))
m = t
WHILE m < max : m = m + t : WEND
IF max < (m * .6) THEN CM1
IF (INT(m / t) MOD 2) <> 0 THEN m = m + t
GOTO CM3
CM1:
t = t / 10
WHILE m > max : m = m - 2 * t : WEND
m = m + 2 * t
CM3:
max = m
RETURN

DrawYTicks: draw ticks and numbers up y axis
interval = 10 ^ INT(FN Log10(max))
IF (max / interval) < 3 THEN interval = interval / 5
 FOR n = 0 TO INT(max / interval + .5)
j = maxY - scale * (n * interval)
LINE (minX - 4, j) - (maxX, j)
IF interval > 1 THEN t$ = STR$(INT(n * interval))
IF interval <= 1 THEN t$ = STR$(n * interval)
CALL MOVETO(minX - LEN(t$) * 8 - 4, j + 4)
PRINT t$;
NEXT n
RETURN

EditFields: edit edit fields
OK = false
WHILE NOT OK
diag = DIALOG(0)
   IF diag = 1 THEN OK = (DIALOG(1) = 1)
   IF diag = 2 THEN eField = DIALOG(2)
   IF diag <> 7 AND diag <> 6 THEN NotTab
eField = eField MOD maxFields + 1
EDIT FIELD eField
MIDNIGHT MADNESS

NotTab:
  WEND
  RETURN

InitData: '----------------------------------------------------------------initialize data array to blank
dNum = 0
  FOR n = 1 TO maxData : gData(n) = -1 : NEXT
  RETURN

OpenWindow: '-----------------------------------------------------------------open second window
  WINDOW 2, , (wind2L,wind2T) - (wind2L + wind2W, wind2T + wind2H), 2
  BUTTON 1, 1, "OK", (wind2W - 70, wind2H - 30) - (wind2W - 10, wind2H - 10)
  RETURN

CloseWindow: '-----------------------------------------------------------------close second window
  WINDOW CLOSE 2 : CLS
  RETURN

SUB DrawXNum(midX, midY, num, maxNum) STATIC ' axis numbers
  IF (maxNum > 20) AND (num MOD 5) <> 0 THEN NoYnum
  CALL MOVETO(midX,midY)
  CALL DrawStr(STR$(num))
  NoYnum:
  END
END SUB

SUB DrawStr(temp$) STATIC ' print centered string
  CALL MOVE(-LEN(temp$) * 7 / 2, 8) : PRINT temp$;
END SUB
MacCursor

MacCursor is a utility that lets you design customized mouse cursors you can use in your Microsoft BASIC programs. Using a facility that's very similar to MacPaint's FatBits option, you draw your cursor with mouse clicks and define how it will interact with the screen background. Other MacCursor facilities let you see what the cursor will look like when your program
uses it, save cursor definitions on disk and retrieve them, and generate a set of DATA statements that a BASIC program can use to create the cursor without loading a disk file.

The ability to redefine the cursor shape is useful in Macintosh programs for both aesthetic and practical reasons. MacPaint, for example, uses a number of cursors that depict a paintbrush, a hand, an eraser—whatever is appropriate to the current activity. These cursors give the user an almost tactile feel for what’s being done with the mouse. When you erase something in MacPaint, for example, you can almost see the eraser rubbing out the picture.

Several programs in this book use special cursors. In MacSpace, for example, the regular arrow-pointer cursor is changed to a cross-hair gunsight for aiming lasers and photon torpedoes. Programs will generally change the mouse pointer to a special shape while the mouse is in a certain area of the screen or in a window devoted to a particular function or activity. When the mouse is moved out of this area (for example, when the user wants to pull down a menu), the cursor is changed back to the standard arrow-pointer shape. You’ll no doubt find the ability to define new cursor shapes useful in many ways in your Microsoft BASIC programs.

Why MacCursor?

You’ve probably learned that Microsoft BASIC uses the Macintosh QuickDraw routine SETCURSOR to show the cursor in a specified pattern or shape. This pattern, as well as other cursor characteristics, must be stored in an array that is passed to the routine when it is called.

You can think of cursors as occupying a matrix that is 16 pixels square: Each row of 16 pixels in the cursor pattern is represented by a 16-bit integer, and there are 16 rows in all. In defining a cursor, the pattern of 1s and 0s corresponding to the black and white pixels in each row must be converted to a number and assigned to the corresponding element of the array. In addition to the 16 numbers that define the cursor pattern, 16 more numbers representing the mask—the pattern that determines how cursor pixels will interact with the existing screen pixels under them—must also be determined and put into the array. (It’s called a mask because you can think of it as a grid superimposed over the cursor grid, and, like a
party mask, it affects the way the cursor will be seen.) Finally, the y and x coordinates of the “hot spot” (the part of the cursor read as the actual mouse-pointer location) must be specified in the array.

Defining a cursor by hand is an extremely tedious process. After drawing a cursor pattern on graph paper, you have to convert each of its 16 lines to a hexadecimal or decimal integer to be stored in the cursor information array as elements 0 through 15. (BASIC won’t take binary numbers directly.) The 16 lines of the mask also have to be converted and stored as elements 16 through 31. Finally, the hot spot’s y and x coordinates must be stored as elements 32 and 33, respectively. You’d probably put all these numbers in DATA statements and have your program read them into the array with a FOR...NEXT loop. Then you’d have to write a program to display the cursor, and chances are you’d find mistakes in transcription or conversion and perhaps things that didn’t come out the way you wanted them to. You’d then have to go back and change the drawing or the binary numbers and their conversion to values in the DATA statements and try again.

MacCursor greatly simplifies the process of creating a cursor. You just draw the cursor in a 16-by-16-pixel FatBits-like grid and let the program take care of the arithmetic conversion—a sensible division of labor. As you create the magnified cursor pattern, you can bring it over to a special play area and see what it actually looks like in real-size screen pixels against a black or white background. An important feature of MacCursor lets you save your custom cursors as disk files and load them into your own programs or back into MacCursor for further editing. MacCursor can also generate the DATA statements necessary to produce a particular cursor. If you save these statements on the Clipboard, you can paste them into your BASIC programs to create “stand-alone” cursor initialization routines.

How to use MacCursor

When you run MacCursor, you see the screen shown in Figure 1 on the next page. This screen has two main areas: a work, or definition, area to the left, where you create your new cursor, and a play, or test, area to the right, where you can see what the currently defined cursor looks like in its actual size against a black or white background. We’ve defined a sample cursor shaped like a mechanical pencil that we’ll discuss shortly.
At the top of the screen you can see a new pull-down menu that's called MacCursor.

The options available on this menu are *Load* (a cursor definition file from disk), *Save* (a cursor file to disk), *Copy* (cursor DATA statements) to Clipboard, *Instructions*, *Quit to BASIC*, and *Quit to Desktop*. Choosing *Instructions* gives a brief summary of how to create, test, and save a cursor.

An option for creating a cursor is not needed, because that's MacCursor's default activity. If you're not asking to load, save, or copy a cursor or to exit the program, MacCursor assumes that you're working on defining a cursor and going back and forth between drawing and testing it.
To define a cursor, move the mouse pointer within the box on the left side of the screen. Think of this box as being divided into 256 little squares—16 across and 16 down—representing the 16- by 16-pixel grid that makes up a cursor. Wherever you want the cursor to have a pixel turned on (displayed as black), click the mouse button. You can draw a whole line of black pixels by dragging the mouse (this takes a bit of practice). If you want to erase a black pixel, move the pointer over the pixel and click the mouse button to turn it off (turn it back to white). To erase the whole cursor image, click on the Erase button below the work area. If you’ve used MacPaint’s FatBits, you’ll recognize this way of working.

You need to specify the hot-spot location, the one pixel of the 256 that make up the cursor grid that will be established as the actual mouse-pointer location. You do this by clicking on the HotSpot button below the drawing area and then clicking on the pixel you want to designate as the hot spot. The coordinates of the hot-spot location are displayed just below the work area, and the hot-spot pixel itself has a little dot drawn in its center. You can reset the hot spot at any time by simply repeating this process. The best location for the hot spot depends upon what the cursor is supposed to represent. The tip of a pen is its “business end,” so we picked one pixel at the tip of our pen cursor to be our hot spot. A gunsight cursor might have its hot spot in the center of its cross hairs, while an icon of a hand might have its hot spot in one of its fingers, and so on.

After you’ve drawn the cursor pattern itself, you may want to add a cursor mask. As mentioned, the purpose of the mask is to define how the cursor will act on different backgrounds. For example, by surrounding black cursor pixels with white mask pixels, a white outline will surround your icon and allow it to stand out on a black background.

To create a mask, first click on the round Mask button at the bottom left of the screen; thereafter, all the pixels you click on in the work area will be outlined in grey and become part of the mask (until you click the Cursor button). You can reverse a mask pixel by clicking on it again. If you click on a black cursor pixel, it becomes, in effect, both a cursor and a mask bit. But it’s important to understand that there is a difference between cursor pixels and mask pixels, even though both appear in the work area at the same time. The difference is that cursor bits and mask bits are stored in separate sections of the cursor array. The information in the bottom right corner of the screen summarizes the relationship between the state of the cursor bits, the state of the mask bits, and the resulting pixels on the screen. Experiment with different mask/cursor arrangements to see how they interact with each other and the background. The purpose
of the mask part of a cursor is to define how the cursor should act on different backgrounds. For example, by surrounding black cursor bits with white mask bits you can make the outline of your icon appear on either a black or white background.

Sometimes you might want the mask pixels and the cursor pixels to be exactly the same, so that the cursor won't change shape on different backgrounds. We've provided the large Cursor -> Mask button below the work area for that purpose. Clicking on this button copies all the cursor bits into the mask so that the two shapes coincide.

Since the image you're creating here is a considerably “blown up” version of an actual cursor, it's not always easy to guess what your new cursor will look like when it's in use, not to mention how the mask will affect it. To see an actual-size representation, move the mouse pointer to the play area on the right side of the screen. As the pointer enters the black rectangle, your new cursor appears.

To get some practice, define our pen cursor example by clicking on the pixels shown in black. Click on the HotSpot button and put the hot spot in the pixel shown with the little white dot in the black square. Click on the Mask button and turn on all the mask pixels shown in gray outline.

Test the cursor by moving the pointer into the black rectangle in the play area. You should see a black pen with a white outline. Move the cursor to the inner white rectangle, and a black pen with no outline appears. This behavior is caused by the mask. If a pixel is a white mask pixel but not a black cursor pixel (the outermost and center rows in our example), it is always white, as the relationship table in the bottom right corner of the screen shows. When the pen is displayed against a white background, the mask pixels are invisible, but when the background is black, they form a white outline around the pen and a white rectangle inside it. Since the black pixels that make up the pencil itself are also mask pixels, they’re always displayed in black regardless of the background (again, check the table). Thus you always see the black bits against a white background, either a white screen background or a white background created by the other mask bits.

While you're testing the cursor, the current pointer position in screen pixel coordinates is also displayed below the play area. The X and Y reference lines drawn through the play area help you note distances between the hot spot and other parts of your cursor and see how the cursor relates to the mouse-pointer coordinates.

When you've gone back and forth between drawing and testing the cursor and are satisfied with it, pull down the MacCursor menu and
choose Save from it. A dialog box asks you for the name of the file in which to save your cursor.

To load a saved cursor for further work, choose the Load option from the menu. You'll be given a standard file dialog box showing all the cursor files that are on the disk; just click on the name of the file containing the cursor you want.

**Using the cursor in your BASIC programs**

There are two ways you can use a cursor created with MacCursor in a BASIC program. The first is to save the cursor as a disk file as we just described. Your program should dimension an integer array to hold the cursor information; for example, `cursor(33)`. Your subroutine for setting up the new cursor would open the disk file and read the information into the cursor array (see our LoadCursor subroutine, marked [ME 1] in the program listing at the end of the chapter). To make the new cursor appear on the screen, simply use the statement:

```basic
CALL SETCURSOR(VARPTR(cursor(0)))
```

where `cursor` is the name of your cursor information array. To restore the normal pointer cursor, `CALL INITCURSOR`. That's all there is to it! Of course, the purpose and design of your program will determine when and where the new cursor (or cursors) will be used.

The second way of using MacCursor cursors in a BASIC program is to choose the Copy to Clipboard option after creating or loading a cursor. The information needed to define the cursor is copied to the Clipboard as a series of DATA statements containing a decimal integer for every cursor array element, complete with a comment to identify them. You can then start a new BASIC file (or load an existing program) and paste this information into the program. After that, you dimension an integer array for the cursor information as previously described, code a loop to READ the DATA statements into the array, and use `CALL SETCURSOR` and `CALL INITCURSOR` in the same way as before. The listing in Figure 2 on the next page shows the essentials for using our example pen cursor as a set of DATA statements: This program simply sets up and displays the pen cursor until the mouse button is pressed.

Why would you want to use this method instead of just loading a cursor from disk? Well, programs often get copied from disk to disk, and if you use a disk file, you have to remember to copy the cursor-definition file(s) along with the program file. Should you want to sell or distribute the
sample cursor-reading program

DEFINT a-z
DIM cursorInfo(33)
FOR n = 0 TO 33: READ cursorInfo(n): NEXT
CALL SETCURSOR(VARPTR(cursorInfo(0))) 'set to pen cursor
WHILE MOUSE(0) = 0 : WEND 'wait for button press
CALL INITCURSOR 'restore arrow-pointer

' cursor data from MacCursor
DATA 6, 9, 25, 46, 70
DATA 138, 276, 552, 1104, 2208
DATA 4352, 8704, 9216, 30720, 24576
DATA -32768, 15, 31, 63, 127
DATA 255, 511, 1022, 2044, 4088
DATA 8176, 16288, 32512, 32256, -1024
DATA -2048, 24576, 15, 0

FIGURE 2. Using a cursor defined in MacCursor in a BASIC program

program, you may not want to or be able to include a disk with the cursor or other graphics information file(s) on it. Putting the cursor information in the program code itself takes up a bit more space but enables the program to stand alone. On the other hand, having code in your program that reads a disk file makes it easy to create and test new cursors simply by saving them under a particular file name and having the program load them. It also allows you to have different BASIC programs access standard cursors. You could even make a library of cursors and put them all in one big disk file. Perhaps the best approach is to have code in your program that first looks for a particular cursor-definition file and loads it if present; if the file isn't found, the cursor is read from a set of DATA statements. We've taken this approach with cursor and other graphics data in some of the game programs in this book to give you the best of both worlds.

How MacCursor works

MacCursor's general structure is based on the idea that the user is always in one of two places: the work (or definition) area, editing cursor pixels, or the play (or test) area, viewing the new cursor. The QuickDraw SETCURSOR routine is used to show the new cursor, while INITCURSOR is used to change it back to the standard pointer (arrow) cursor. As the cursor is defined, three kinds of information are updated in the cursor array: the bit patterns indicating the state of every pixel in the array, the bit patterns for the cursor mask, and the y and x coordinates of the hot spot.
One major loop handles each of the areas, and the status is set to the other area as soon as the user leaves either one. Additionally, at any time the user may choose a menu item or press one of the five buttons below the work area. Since all menu items and buttons are independent, they are simply processed by the MenuEvent [ME 0] and ButtonEvent [BE 0] routines as they are encountered.

Understanding how this program works will give you a good grasp of how bit patterns are represented and used. Since many Quick-Draw routines use arrays of bit patterns, you'll be well equipped for using these routines in your programs. We'll also show the advantages of using subprograms rather than subroutines for certain functions. A program outline showing the program structure in more detail is shown in Figure 3.

---

### Initialization

- initialize arrays and constants
- define mouse location functions
- create and enable MacCursor menu
- set up output window
  - set up initial screen (DrawScreen)
  - show hot spot in work area (ShowHotSpot)
- create and enable buttons

### Main Program

WHILE (endless loop interrupted by menu and button events)

WHILE pointer in play area

- get pointer position (GetMouse)
- if not in play area, change to work area (SetWork)
- if pointer has moved, show new position (ShowMousePos)

WEND

WHILE pointer not in play area

- get pointer position (GetMouse)
- if pointer in work area and button down,
  - find corresponding bits and reverse their color/state
  - while button down, keep changing bits under pointer (AffectBits)
- if pointer now in play area, reset to play area (SetPlay)

WEND

WEND

### Menu Events

- MenuEvent: transfer control to appropriate subroutine
- LoadCursor: load cursor from disk
- SaveCursor: save cursor to disk
- CopyToClip: copy cursor as DATA statements to Clipboard
- Instructions: show instructions
  - then redraw screen (RedrawCursor)

---

FIGURE 3. Program outline
BasQuit: quit to BASIC
DeskQuit: quit to Mac Desktop

**Button Events**
- ButtonEvent: transfer control to subroutine
  - HotSpot: change location of hot spot
  - EraseCursor: erase work area
  - BitMode: change between cursor and mask drawing
  - CursToMask: make mask same as cursor

**Subroutines**
- RedrawCursor: 
  - redraw screen (DrawScreen)
  - update FatBits work area (ShowFat)
  - show hot spot location (ShowSpot)
  - redisplay hot spot coordinates (ShowHotSpot)
- DrawScreen:
  - draw whole screen (except FatBits in work area)
- AffectBits:
  - WHILE button is still down:
    - get new pointer position
    - if pointer in work area:
      - reverse corresponding bit
      - update cursor array, either cursor or mask
      - show FatBit or gray mask outline (ShowFat)
  - WEND
- ShowFat: show FatBit for cX and xY
- ShowHotSpot: show with text and graphics where hot spot is
- ShowSpot: invert dot on bit where hot spot is located
- ShowMousePos: show current mouse position
- SetPlay: show what new cursor looks like
- SetWork: reset to regular mouse pointer
- GetMouse: get current mouse status and location

**Subprogram**
- PrintPair: print pair of numbers in parentheses

Figure 3. Program outline (continued)
Initializing screen areas and location functions

All numeric variables in this program are integer (DEFINT A-Z). Two main arrays are dimensioned in [Init 1]. The cursor array will contain the information the program needs when calling the Macintosh SETCURSOR routine to show a newly defined cursor. This information will also be loaded from or saved to disk as a cursor definition. Figure 4 shows the relationship between the cursor array and the cursor and mask patterns.

As the diagram shows, elements 0 through 15 of the cursor array contain the actual data that describes how each pixel of the cursor will be displayed. They correspond to rows 0 through 15 in the cursor-definition work area. Recall that each row has 16 positions (small rectangles) representing the pixels in the cursor; and that since the Macintosh represents integers internally as 16-bit binary numbers (with decimal values of -32768 through 32767), we can use the 16 individual bit positions in one integer element of the cursor array to represent one row of pixels in the cursor or mask.
For example, let's suppose we want the first row, row 0, to have the last four pixels to the right displayed as black. In the binary representation of this row, a 1 is used to indicate a pixel that is black, while a pixel that is white is shown as a 0. Thus, the binary number 0000000000001111 would represent the current state of this row. To make the two center pixels in the next two rows black, the binary value 0000000110000000 would be used for each of these two rows. We have drawn these pixels in black, as shown in Figure 4.

Figure 5 shows some more sample cursor patterns. As a useful exercise, define one of these with MacCursor, then choose the Copy to Clipboard option from the MacCursor menu. Print out the resulting DATA statements and verify the way these values relate to the cursor, mask, and hot-spot values.

FIGURE 5. More example cursors
Figure 5. More example cursors (continued)
As the user makes changes by clicking the mouse button with the pointer over different pixel positions, MacCursor updates the cursor array by setting each corresponding bit to a 1 if a pixel has been made black or a 0 if it has been made white (erased).

The same scheme is used to represent the mask pixels as they are defined. As we have noted, the mask bit patterns go into elements 16 through 31 of the cursor array, and the hot-spot y and x coordinates (relative to the 16-by-16 grid, not the screen) go into elements 32 and 33.

The other main array in MacCursor is the bitTable array. This will be used in updating the cursor array. Note that this array is initialized in [Init 1] so that each of its elements has only one of its bits set. The first element, bitTable(0), is set to &H8000. If you haven't encountered this kind of notation before, the &H warns you that this is a hexadecimal number. &H8000 corresponds to binary 1000000000000000, or 2 raised to the fifteenth power (215). The FOR...NEXT loop that follows sets the next element, bitTable(1), to 2 to the fourteenth power (15 - 1), which is binary 0100000000000000, and each of the next elements to the next lower power of 2. Thus element 0 has only its first (counting from the left) bit set, element 1 has only its second bit set, and so on. Later, the elements in the bitTable array will be used with logical operators to turn bits in the cursor array on or off. We'll get into the gory details later.

The next thing the program initializes is the pattern used to draw gray borders around the mask pixels. As you may recall, a pattern is defined rather like a cursor, except that it has only four lines. The bit patterns for the lines are stored in the four elements of the gray array.
Also in [Init 1], three areas of the screen have their coordinates defined as constants: the work area on the left side of the screen, the outer (black) rectangle in the play area on the right side of the screen, and the inner (white) rectangle within the black one. Note the technique of defining coordinates in terms of previously defined ones. The cross hair where the X and Y reference lines meet has its coordinates derived simply by adding the left and right x coordinates of the play area and dividing by two, then doing the same for the y coordinates.

Three functions are defined in [Init 2]. InBits tells us whether the mouse pointer is in the work area, and InPlay does the same for the play area. The screen coordinates passed to the functions are simply checked to see whether they're within the appropriate borders. We'll describe BitOn later when it's used. As you've seen if you've looked at some of our games, functions that return logical values like these do are very useful because they can be used directly in a test (such as IF...THEN) to perform the appropriate action.

In [Init 3] the MacCursor menu is set up, and menu events are enabled by the ON MENU GOSUB statement. The MenuEvent subroutine [ME 0] handles all menu events. The Output window is specified next [Init 4], and then the startup screen is drawn. First DrawScreen [SR 2] is called to draw the boxes for the work area and the play area, including the inner white rectangle for the latter. It also displays the X and Y crosshair reference lines and prints the various text labels needed.

A neat little trick is used here to print the “X= ” and “Y= ” labels for the reference lines: The values of their respective coordinates (hX and hY) are converted to strings and then printed after being formatted with the RIGHTS$ function. This makes for a tidy display by eliminating unnecessary digits like leading zeroes and trailing spaces that BASIC would otherwise throw at you.

Next, ShowHotSpot [SR 5] is called to display the current location of the hot spot just below the work area. cursor(33), the x coordinate, is printed before cursor(32) so the screen will display the more usual (x,y) order. To print the coordinates, ShowHotSpot calls a little subprogram, PrintPair [SP 1], which is another example of string formatting—and also of the usefulness of subprograms. If we used a subroutine rather than a subprogram, we'd have to assign the string we're working with to the global string variable the subroutine is expecting to manipulate. We then might have to assign the formatted string back to the variable being used by the calling routine. With a subprogram, we simply pass the name of the string variable in parentheses after the CALL PrintPair; the subprogram
processes it, and we get a properly formatted pair of coordinates without having to use any assignment statements. Again note that numbers to be printed behave better when they're converted to strings and then manipulated with string functions than if they're left as numbers.

After printing the hot-spot coordinates, ShowHotSpot calls ShowSpot [SR 6]. This subroutine gets the position in the work area that corresponds to the hot-spot coordinates and inverts a dot there to mark it.

Control now comes all the way back to [Init 5], which sets up the five buttons under the work area and enables dialog handling. The subroutine ButtonEvent [BE 0] will (surprise!) handle button events.

The main program

The main program is an endless loop that is exited only by one of two Quit menu events, [ME 5] or [ME 6]. We'll describe later how each menu event and each button event is processed. First, though, let's look at MacCursor's default activity, which is handled by getting the mouse-pointer position and determining which of two activities the user is engaged in: drawing the cursor or mask by clicking bits on and off in the work area, or "playing with" or testing the cursor in the play area. Initially it is assumed that the mouse pointer is in the work area and that the user wants to define the cursor before defining the mask. Thus, inPlayArea and affectMask are set to false in [Init 1] prior to the beginning of the main program, and control falls through to the loop at [Main 2].

Work area activity

The WHILE...WEND loop in [Main 2] handles the cursor-drawing activities. First, the mouse-pointer position is checked by calling GetMouse [SR 10]. This subroutine returns the mouse state in status. Then an IF...THEN statement checks for the mouse button being down (status < 0) and the mouse being in the work area, the latter check being made with the InBits function defined in [Init 2]. If the mouse button is down and the mouse pointer is in the work area, the user wants to change the cursor pixel represented by the little rectangle under the pointer. In preparation for doing that, the specific bit that corresponds to the pointer location is found by dividing the mouse x and y coordinates by the size of the little rectangles (10).
Manipulating cursor pixels and bits

Now we’re ready for some fairly complicated bit manipulation. It would be tempting to use a two-dimensional array and store the states of the cursor and mask bits in it, but using a whole array element to store the on/off (1/0, black/white) value of one pixel would waste a lot of memory. Applications dealing with areas of the screen larger than 16 by 16 pixels would soon run out of memory. Also, the Macintosh SETCURSOR routine, like many other QuickDraw routines, expects an array of binary bit patterns. Thus, it makes more sense to work directly with the bit patterns, even though they’re harder to understand at first.

To make what follows clearer, let’s assume that the user has clicked on the fourth rectangle from the left in row 2. This rectangle has an x coordinate (cX) of 3 and a y coordinate (cY) of 1, since the coordinates start with 0. (Remember that these are coordinates in the 16-by-16 work area and are derived by division from the mouse pointer’s screen coordinates.) Since the user wants to reverse the color of this pixel, we want to “flip” the bit that represents the pixel to its opposite state. In other words, if it’s a 1 we want to make it a 0, or vice versa.

The BitOn function defined in [Init 2] will return a true value if the bit (or pixel) under the mouse pointer is a 1 (that is, if it’s black). If you look back at [Init 2], you’ll see that BitOn compares the value of the cursor array element for the row the bit is in (cY) with the bitTable element for the column the bit is in (cX) — in this case bitTable(3). Recall that we set up bitTable so that each element has only one bit — the bit corresponding to the column number, counting from the left with 0 — set to 1 [Init 1], so bitTable element 3 has the bit pattern 00010000000000000. cursor array element 1 may have any combination of 1s and 0s, but we’re interested only in the bit at cY, which we can represent as X, or 0000000000000000.

Another complication is that the pixel under the pointer may be part of either the cursor itself or the mask. This is why, when BitOn is called, the value of the cryptic expression $16 \times -affectMask$ is added to the value of cY. If the user has clicked on the Mask button, affectMask will be true (−1). Since $-(-1) = 1$, and $16 \times 1 = 16$, the value of cY will have 16 added to it, offsetting it into the Mask portion of the cursor array. Otherwise, if the user has not clicked the Mask button, affectMask will be false (0) and cY will stay in the cursor area of the array (elements 0 through 15).
The AND logical operator is used to make the comparison between the cursor array element and the bitTable element. With AND, if the two bits being compared are both 1, the result is 1, but if one of the bits is a 0, the result is 0. So, if the X bit in the cursor array element is 1, the AND leaves it 1 because the comparison bit in bitTable(3) is also 1; if it is 0, the AND leaves it 0.

If, after the AND comparison has been done and the BitOn function returns true indicating the bit is 1 (black), we set turnOn to false so we can flip the bit to 1 (white). If BitOn returns false indicating the bit is 0, we set turnOn to true so we can flip it to 1. We have finally gotten the effect we wanted, and we’re ready to change the bit, and therefore the pixel, to its opposite state.

We realize that all of this can be heavy going for those of you who haven’t dealt with binary numbers or assembly language. If what’s happening still isn’t clear to you, we suggest rereading our description of the cursor and bitTable arrays, the description of the cursor routines and logical operators in the Microsoft BASIC manual, and Chapter 15, “Quick-Draw,” in Microsoft Macinations (Microsoft Press, 1985).

After turnOn has been set, AffectBits [SR 3] is called. This subroutine is a loop that will run as long as the mouse button is being held down in the work area. This feature allows the user to draw a line across the work area by dragging the mouse. AffectBits gets the mouse-pointer location and checks to make sure the pointer is still in the work area. If it’s not, the subroutine returns control to the main program loop. Otherwise, the procedure we’ve just discussed is followed to get the location of the mouse pointer in the work area and offset cY if the program is in the mask-definition mode.

As you recall, we’ve already determined whether we need to set the pixel to black or white and have set turnOn accordingly. If turnOn is true (that is, the bit is 0), AffectBits now sets the bit to 1 (black) by ORing it with the number from bitTable, which we’ve learned has this bit set to 1. If you OR a 0 with a 1 you get a 1, since OR results in 1 if either bit is a 1, so this bit in the cursor array element corresponding to the row becomes a 1.

If turnOn is false (that is, the bit is 1), we want to turn the 1 off (make it a 0). The way we chose to do this is to AND it with a 0, since 1 AND 0 gives you 0 (remember, AND needs both bits to be 1 to result in 1). How do we make sure we are ANDing it with a 0?

We guarantee that there’ll be a 0 bit in the corresponding position of the bitTable number if you NOT the bitTable number. The NOT logical
operator reverses the state of every bit in the number. Since the bitTable number has a 1 in the position we’re interested in, the result of the NOT gives a number with a 0 in that position. That number, when ANDed with the cursor array number, makes the latter’s bit a 0 (a white pixel).

After updating the information in the cursor array, AffectBits calls the ShowFat (for “show FatBit”) subroutine [SR 4] to update the work area display using appropriate QuickDraw routines. ShowFat gets the boundaries for the rectangle of the graphic to be drawn and stores them in the graphics rectangle array r. If the bit for the pixel is now a 1 and if it belongs to the mask (cy is offset by 16), a call to FILLRECT fills the rectangle with the gray pattern. Otherwise, the rectangle is erased with a call to ERASERECT.

Next, the graphics boundary is drawn a little smaller on each side and BitOn is again checked, this time to see if the corresponding cursor bit is on. If it is, the PAINTRECT call draws the black rectangle representing the pixel. If the pixel is supposed to be white, an ERASERECT erases the rectangle. The rectangle for the cursor bit is a little smaller than the one used for the mask bit, so a gray border remains if the pixel is also a mask pixel. Since a pixel that is represented in the work area can be a cursor and/or a mask pixel, two rectangles are therefore needed in order to show the right combination.

If the current location is also the location of the hot spot, the hot-spot circle needs to be redrawn, so ShowSpot [SR 6] is called to accomplish that. We’ve already described how this subroutine works.

Control now returns to the WHILE...WEND loop in AffectBits, and changes continue to be processed as long as the mouse button is down. When the mouse button is no longer down, control returns to the outer NOT inPlayArea loop at [Main 2], and the mouse is checked until the button is pushed again or the mouse pointer is moved into the play area. This latter situation is determined by checking the function InPlay at the bottom of the loop. When this function returns a true value, SetPlay [SR 8] is called. SetPlay sets inPlayArea to true and finally calls the QuickDraw SETCURSOR routine to display the cursor as it is currently defined in the cursor array.

Since inPlayArea is now false, control goes all the way back to the top of the main program loop. Since the outermost WHILE...WEND loop will continue indefinitely until we choose Quit from the MacCursor menu, the loop for inPlayArea is executed.
Play area activity

While the mouse pointer is in the play area, the pointer position is obtained via GetMouse. If the InPlay function returns a false value, SetWork [SR 9] sets the variable inPlayArea to false and restores the standard pointer (arrow) cursor by a call to the Macintosh INITCURSOR routine. Next, if the pointer position has changed, ShowMousePos [SR 7] is called to show the current mouse-pointer position by printing its coordinates using the PrintPair subprogram. Control then falls through the bottom of the WHILE...WEND loop, and the WHILE NOT inPlayArea loop we discussed in the last section is executed again.

Button events

Button events (the user clicking on any of the five buttons below the work area) are handled by the ButtonEvent subroutine [BE 0], which first checks the DIALOG(0) function to make sure a button press rather than some other kind of dialog event has occurred. If a button press has occurred, the MacCursor menu bar is dimmed so the user can’t try to make a menu selection while the button selection is being processed. The first two buttons, which involve fairly time-consuming procedures, are also deactivated and dimmed for the same reason.

If the HotSpot button was pressed, HotSpot [BE 1] is called. This subroutine first calls ShowSpot [SR 6] to erase the current hot-spot circle after getting its screen coordinates from cursor(32) and cursor(33) and converting them to the corresponding FatBits work-area coordinates. The coordinates of the former hot spot are then erased from the display below the work area by printing a string of blanks over them. A WHILE...WEND loop next waits for the mouse button to be pressed, indicating where the user wants to put the new hot spot. When the button is pressed, the InBits function is checked to see if the pointer is in the work area: If it isn’t, a beep is sounded and control goes back up to the WHILE...WEND loop to wait for another button press. If the pointer is in the work area when the button is pressed, the pointer coordinates are converted to their corresponding cursor coordinates (by now, a familiar operation), which are then placed in cursor(33) and cursor(32) to update the cursor array. A second WHILE...WEND loop waits until the user has stopped pressing the mouse button if the user hasn’t released the button yet. A sound then indicates that the hot spot has been set, and ShowHotSpot [SR 5] is called to show the new hot spot. Control returns to ButtonEvent, where the menu and buttons are reactivated before return to the main program loop.
The next button that could have been pressed is the Erase button. If it’s pressed, EraseCursor [BE 2] is then called. This subroutine uses a FOR...NEXT loop to set the elements of the cursor array (except the coordinates of the hot spot) to 0, clearing out the bit pattern that defines the cursor and mask. DrawScreen [SR 2] is then called. This subroutine redraws all the screen except the pattern that was in the work area. Next, EraseCursor calls ShowHotSpot [SR 5] to draw the circle representing the current hot-spot position, and control then returns to ButtonEvent [BE 0].

The other three button events are quite simple. The Cursor and Mask buttons control whether the cursor or the mask is currently being edited. Clicking on either button calls BitMode [BE 3]. This subroutine uses the BUTTON statement with a 2 to show the button that was clicked as being selected (with a black circle in its center). Since the Cursor and Mask buttons are numbers 3 and 4, subtracting the pressed button from 7 gets you the other button, which is shown as blank (not selected) by a BUTTON statement with a 1. The value of the Boolean variable affectMask is set to true (or -1) if button 4 (Mask) was pressed; otherwise it is set to false (or 0). Remember that an expression like pressed = 4 is itself Boolean, since it will be either true or false, so the value of such an expression can be assigned to a variable.

The last button event occurs when the user clicks on the Cursor — > Mask button, which calls the CursToMask subroutine [BE 4]. This simply uses a loop to copy elements 0 through 15 of the cursor array into elements 16 through 31 (the mask elements). RedrawCursor [SR 1] is then called. This subroutine first calls DrawScreen [SR 2] to redraw all the screen except the pattern that was in the work area. RedrawCursor redraws the cursor pattern in the work area with two nested FOR...NEXT loops. The outer loop takes one row (one work-area y coordinate) and the inner loop takes one column (one work-area x coordinate) at a time from the current row; thus the whole grid is processed bit by bit.

The BitOn function is used to check whether either the cursor bit or the mask bit at a given location is a 1. If it is, ShowFat [SR 4] is called to draw the appropriate representation. Since in this case RedrawCursor was called by EraseCursor, the bits in the cursor array have been set to 0, so ShowFat erases (fills with white) all the little rectangles in the work area. Finally, RedrawCursor calls ShowHotSpot [SR 5] to draw the circle representing the current hot-spot position, and control then returns to CursToMask. The latter has nothing more to do, and so control goes back to ButtonEvent [BE 0].
After any button event has been processed, ButtonEvent [BE 0] concludes by reactivating the menu bar and the buttons so the user can make a new menu or button selection if desired. Whatever activity was interrupted by the event now resumes.

Menu events

There are six possible menu events, all handled by MenuEvent [ME 0]. This subroutine deactivates the menu, calls the appropriate subroutine, and then reactivates the menu. The events and their subroutines need only brief discussion.

LoadCursor [ME 1] gets the name of a file containing the cursor definition via the FILES$ function. FILES$ is used with the file attribute "Curs," so that only cursor file names will appear in the FILES$ dialog box. The file is opened, its contents are read into the cursor array, and RedrawCursor [SR 1] is then called to show the cursor.

SaveCursor [ME 2] uses FILES$ to ask for a file name, opens this file, sets its file attribute to "Curs" and saves the contents of the cursor array in it. Again, RedrawCursor must be called to redraw the cursor, because part of it was obliterated by the file dialog box.

CopyToClip [ME 3] opens the Macintosh Clipboard file (as text), prints an identifying comment, and then prints the values of the elements of the cursor array as decimal integers in a series of DATA statements. Every fifth data value (when \( n \ MOD \ 5 = 0 \)) has a carriage return and the word DATA printed to start a new line. The "" is a null string required by the syntax of the PRINT# statement: If you were printing to the screen, you could use just PRINT by itself to print a carriage return.

Instructions [ME 4] prints a window with brief instructions on how to use MacCursor. It uses INITCURSOR to restore the normal cursor, then creates a Continue button. The instructions are shown until that button is pressed. The button and the window are then closed, and the screen is redrawn via RedrawCursor.

There's a lesson behind the call to RedrawCursor. If the instructions were selected at the start of the session, there's nothing in the work area, but it's unwise to assume that a menu event will be selected only at a particular time. The user might want to review the instructions at any time, and something may therefore be in the work area. Whenever you
activate a menu or a button, you as programmer are responsible for prop­erly handling the selection(s) represented and restoring the user to the ex­pected environment afterward. Here, this means making sure that the user does not lose whatever may have been in the work area.

BasQuit [ME 5] and DeskQuit [ME 6] are our standard exit rou­tines. Note that they close the two buttons and reset the menu to restore the normal BASIC- or Macintosh-desktop environment and free up the memory used for the menu and buttons.

**Suggestions for MacCursor**

Since we’ve given you a lot of material to digest in this chapter, we’ll sim­ply suggest that you create some cursors and experiment with them. You may find it useful to add FatBits-like features to your own programs by using routines similar to ours. For example, you could write a program that would allow a user to create a custom graphics character set, edit it, and save and load it from disk.
MacCursor program listing

DEFINT A-Z
DIM r(3) 'used for temporary rectangle storage
DIM gray(3) 'gray pattern for QuickDraw
DIM cursor(33) 'cursor data
DIM bitTable(15) 'bit array lookup table

bitTable(0) = &H8000
FOR n = 1 TO 15
    bitTable(n) = 2 ^ (15 - n)
NEXT
FOR n = 0 TO 3
    gray(n) = &HAA55
NEXT
false = 0 : true = NOT false
inPlayArea = false : affectMask = false

bX1 = 10 : bY1 = 30
bX2 = bX1 + 159 : bY2 = bY1 + 159
pX1 = 245 : pY1 = 65
pX2 = 445 : pY2 = 165
hX = (pX1 + pX2) / 2
hY = (pY1 + pY2) / 2

'lnBits returns true if (fX,fY) is in work area
DEF FN lnBits(fX,fY) = (fX >= bX1 AND fX <= bX2 AND fY >= bY1 AND fY <= bY2)

'InPlay returns true if (fX,fY) is in play area
DEF FN lnPlay(fX,fY) = (fX >= pX1 AND fX <= pX2 AND fY >= pY1 AND fY <= pY2)

'BitOn returns true if bit at (fX,fY) in cursor is ON
DEF FN BitOn(fX,fY) = ((cursor(fY) AND bitTable(fX)) <> 0)

MENU 6, 0, 1, "MacCursor" 'add MacCursor menu to menu bar
MENU 6, 1, 1, "Load"
MENU 6, 2, 1, "Save"
MENU 6, 3, 1, "Copy to Clipboard"
MENU 6, 4, 1, "Instructions"
MENU 6, 5, 0, "-" 
MENU 6, 6, 1, "Quit to BASIC"
MENU 6, 7, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON 'enable menu event trapping
WINDOW 1, "", (10.28) - (502,332), 4
GOSUB DrawScreen
GOSUB ShowHotSpot

BUTTON 1, 1, "HotSpot", (bX1 + 5, bY2 + 20) - (bX1 + 75, bY2 + 40)
BUTTON 2, 1, "Erase", (bX1 + 85, bY2 + 20) - (bX1 + 155, bY2 + 40)
BUTTON 3, 2, "Cursor", (bX1 + 5, bY2 + 70) - (bX1 + 75, bY2 + 90), 3
BUTTON 4, 1, "Mask", (bX1 + 85, bY2 + 70) - (bX1 + 155, bY2 + 90), 3
BUTTON 5, 1, "Cursor -> Mask", (bX1 + 5, bY2 + 45) - (bX1 + 155, bY2 + 65)

ON DIALOG GOSUB ButtonEvent : DIALOG ON

MAIN PROGRAM

WHILE true
    WHILE inPlayArea
        GOSUB GetMouse
        IF NOT FN InPlay(mX, mY) THEN GOSUB SetWork
        IF mX <> oldX OR mY <> oldY THEN GOSUB ShowMousePos
    WEND

    WHILE NOT inPlayArea
        GOSUB GetMouse
        IF status > -1 OR NOT FN InBits(mX, mY) THEN NotDown
        cX = INT((mX - bX1) / 10)
cY = INT((mY - bY1) / 10)

        turnOn = NOT FN BitOn(cX, cY + 16 * -affectMask)
        GOSUB AffectBits

        NotDown:
            IF FN InPlay(mX, mY) THEN GOSUB SetPlay
    WEND
WEND

Menu Event:

Menu 6, 0, 0
item = MENU(1)
ON item GOSUB LoadCursor, SaveCursor, CopyToClip, Instructions, , BasOuit, DeskOuit
MENU 6, 0, 1
RETURN

LoadCursor:

file$ = FILES$(1, "TEXT")
IF file$ = "" THEN NoLoad
OPEN file$ FOR INPUT AS #1
FOR n = 0 TO 33
    INPUT #1, cursor(n)
NEXT
CLOSE #1
NoLoad:
  GOSUB RedrawCursor
  RETURN

SaveCursor:
  file$ = FILES$(0, "File name?")
  IF file$ = "" THEN NoSave
  OPEN file$ FOR OUTPUT AS #1
  NAME file$ AS file$, "TEXT"
  FOR n = 0 TO 33
    PRINT #1, cursor(n)
  NEXT
  CLOSE #1
NoSave:
  GOSUB RedrawCursor
  RETURN

CopyToClip:
  OPEN "CLIP:TEXT" FOR OUTPUT AS #1
  PRINT #1, "", _ cursor data from MacCursor".
  FOR n = 0 TO 33
    IF (n MOD 5) = 0 THEN PRINT #1, ": PRINT #1, "Data "; ELSE PRINT #1, "," ;
    PRINT#1, cursor(n);
  NEXT
  PRINT #1, ""
  CLOSE #1
RETURN

Instructions:
  Instructions for MacCursor"
  PRINT "Design a cursor by defining the following three cursor components:";
  PRINT " 1. Define the shape of a cursor by selecting 'Cursor' and clicking bits "
  PRINT "in the work area.";
  PRINT " 2. Design a mask for the cursor shape by selecting 'Mask'. As mask "
  PRINT "bits are turned on, they will appear as gray frames around their"
  PRINT "corresponding cursor bits.";
  PRINT " 3. Select 'HotSpot' and click on a bit of the cursor to place the hot "
  PRINT "spot.";
  PRINT "The rules governing the final appearance of the masked cursor are "
  PRINT "listed at the bottom right corner of the screen. Try out your "
  PRINT "developing cursor in the black and white play area. A cursor may be "
  PRINT "saved to a disk file or to the Clipboard as DATA statements.";

CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1 : WINDOW CLOSE 2
GOSUB RedrawCursor
RETURN
Mac Cursor

BasQuit:
FOR n = 1 TO 5 : BUTTON CLOSE n : NEXT
MENU RESET
END

DeskQuit:
FOR n = 1 TO 5 : BUTTON CLOSE n : NEXT
MENU RESET
SYSTEM

* * * * * * * * * * *

Button Events:

IF DIALOG(0) <> 1 THEN NotButton
pressed = DIALOG(1)
MENU 6, 0, 0 : BUTTON 1,0 : BUTTON 2,0
ON pressed GOSUB HotSpot, EraseCursor, BitMode, BitMode, CursToMask
MENU 6, 0, 1 : BUTTON 1,1 : BUTTON 2,1
NotButton:
RETURN

HotSpot:
GOSUB ShowSpot
CALL MOVETO(bX1 + 20, bY2 + 15)
PRINT " ";
NotHot:
WHILE MOUSE(0) > -1 : WEND
IF NOT FN lnBits(MOUSE(1), MOUSE(2)) THEN BEEP : GOTO NotHot
IX = INT((MOUSE(1) - bX1 - 1) / 10)
IX = INT((MOUSE(2) - bY1 - 1) / 10)
cursor(33) = IX : cursor(32) = IY
WHILE MOUSE(0) < 0 : WEND
SOUND 5000,1
GOSUB ShowHotSpot
RETURN

EraseCursor:
FOR n = 0 TO 31 : cursor(n) = 0 : NEXT
file$ = ""
GOSUB DrawScreen
GOSUB ShowHotSpot
RETURN

BitMode:
BUTTON pressed,2
BUTTON 7 - pressed, 1
affectMask = (pressed = 4)
RETURN

* * * * * * * * * * *

Desk Out:
* * * * * * * * * * *
FOR n = 1 TO 5 : BUTTON CLOSE n : NEXT
MENU RESET
SYSTEM
* * * * * * * * * * *
CursToMask: 

FOR n = 0 TO 15
    cursor(n + 16) = cursor(n)
NEXT
GOSUB RedrawCursor
RETURN

SR 1

RedrawCursor:

FOR cY = 0 TO 15
    FOR cX = 0 TO 15
        IF FN BitOn(cX, cY) OR FN BitOn(cX, cY + 16) THEN GOSUB ShowFat
    NEXT cX
NEXT cY

x = cursor(33) : y = cursor(32)

IF FN BitOn(x, y) OR FN BitOn(x, y + 16) THEN GOSUB ShowSpot
GOSUB ShowHotSpot
RETURN

SR 2

DrawScreen:

CLS
CALL TEXTFONT(0)
CALL TEXTMODE(4000) : CALL TEXTSIZE(24) : CALL TEXTFACE(8)
CALL MOVETO(330, 30) : PRINT "MacCursor"
CALL TEXTMODE(0) : CALL TEXTSIZE(12) : CALL TEXTFACE(0)
CALL MOVETO(8, 20) : PRINT "File name .....
LINE (bX1 - 1, bY1 - 1) - (bX2 + 1, bY2 + 1), b
LINE (pX1, pY1) - (pX2, pY2), , b
LINE (pX1 + 50, pY1 + 25) - (pX2 - 50, pY2 - 25), white, b
LINE (hX, pY1 - 30) - (hX, pY2)
LINE (pX1 - 50, hY) - (pX2, hY)
CALL TEXTMODE(1)
CALL MOVETO(hX + 5, pY1 - 10) : PRINT "X = "; RIGHT$(STR$(hX), 3)
CALL MOVETO(pX1 - 50, hY - 10) : PRINT "Y = "; RIGHT$(STR$(hY), 3)
CALL MOVETO(pX1 + 10, pY2 + 15)
PRINT "Mouse Position:
CALL MOVETO(0, 210) : CALL TEXTFACE(4)
PRINT TAB(25) "Cursor Bit" TAB(35) "Mask Bit" TAB(45) "Resulting Pixel"
CALL TEXTFACE(0)
PRINT TAB(25) "White" TAB(35) "Black" TAB(45) "Always white"
PRINT TAB(25) "Black" TAB(35) "Black" TAB(45) "Always black"
PRINT TAB(25) "White" TAB(35) "White" TAB(45) "Same as screen"
PRINT TAB(25) "Black" TAB(35) "White" TAB(45) "Inverse of screen"
CALL TEXTMODE(0)
RETURN

SR 3

AffectBits: 

WHILE MOUSE(0) < 0
    x = MOUSE(1) : y = MOUSE(2)
    IF NOT FN InBits(x, y) THEN NotInBox
cX = INT((x - bx1) / 10)
cY = INT((y - by1) / 10) ; tY = tY
IF affectMask THEN tY = tY + 16

IF turnOn THEN cursor(tY) = cursor(tY) OR bitTable(cX)
IF NOT turnOn THEN cursor(tY) = cursor(tY) AND NOT bitTable(cX)
GOSUB ShowFat

NotInBox:
WEND
RETURN

ShowFat:
; show fat bit for cX and xY
r(0) = 10 * cY + by1 : r(1) = 10 * cX + bx1
r(2) = r(0) + 10 : r(3) = r(1) + 10
IF FN BitoN(cX, cY + 16) THEN CALL FILLRECT(VARPTR(r(0)), VARPTR(gray(0)))
IF NOT FN BitoN(cX, cY + 16) THEN CALL ERASERECT(VARPTR(r(0)))

r(0) = r(0) + 1 : r(1) = r(1) + 2
r(2) = r(2) - 1 : r(3) = r(3) - 2
IF FN BitoN(cX, cY) THEN CALL PAINTRECT(VARPTR(r(0)))
IF NOT FN BitoN(cX, cY) THEN CALL ERASERECT(VARPTR(r(0)))

'if this is where hot spot is, then draw dot
IF cx = cursor(33) AND cY = cursor(32) THEN GOSUB ShowSpot
RETURN

ShowHotSpot:
; show with text & graphics where hot spot is
CALL MOVETO(bX1 + 20, by2 + 15)
PRINT "HotSpot: ";
CALL PrintPair(cursor(33), cursor(32))
GOSUB ShowSpot
RETURN

ShowSpot:
; invert dot on bit where hot spot is
r(0) = cursor(32) * 10 + by1 + 3 : r(1) = cursor(33) * 10 + bx1 + 3
r(2) = r(0) + 4 : r(3) = r(1) + 4
CALL INVERTOVAL(VARPTR(r(0)))
RETURN

ShowMousePos:
; show current mouse position
CALL MOVETO(pX1 + 10 + WIDTH("Mouse Position: "), pY2 + 15)
CALL PrintPair(mX, mY)
RETURN

SetPlay:
; show what new cursor looks like
inPlayArea = true
CALL SETCURSOR(VARPTR(cursor(0)))
RETURN

SetWork:
; reset to regular mouse pointer
inPlayArea = false
CALL INITCURSOR
RETURN
GetMouse:
  
  get current mouse status and location
  
  status = MOUSE(0)
  oldX = mX : oldY = mY
  mX = MOUSE(1) : mY = MOUSE(2)
  RETURN

SUB PrintPair(num1, num2) STATIC
  'print pair of numbers in parens
  temp1$ = STR$(num1)
  temp2$ = STR$(num2)
  PRINT "(" RIGHT$(temp1$, LEN(temp1$) - 1) ",";
  PRINT RIGHT$(temp2$, LEN(temp2$) - 1 ")";
END SUB
MacConvert

No, MacConvert won't change your religion: What it will do is convert your Microsoft BASIC 1.0 programs for the Mac to the enhanced, version 2 series. "What's the point?" you might ask. That question is best answered by comparing the two versions of BASIC. Once you understand version 2's improvements you'll want MacConvert in your arsenal of utility programs.
A little BASIC history

When Microsoft first brought out BASIC for the Macintosh, it chose to present a version that mimicked those it had developed for earlier computers, such as the IBM PC and Apple II. These versions had been in use for several years, and although they were popular, they lacked certain features that make other languages such as Pascal so appealing.

Probably the single biggest drawback of earlier versions of BASIC is their requirement that each line in the program be preceded by a number. This necessity presents several problems. First, it’s a nuisance to type in a number for each program line and then keep track of all these numbers, because if you insert a line into an existing segment of code, you have to make sure the number falls in the right place. (Although some versions of numbered BASICS offer automatic line numbering, they don’t always solve the problem: If you set up a special routine to start at line 3000, and BASIC renumbers the program, you may find that it now starts at line 2095 or 3015, which really plays havoc with GOSUB and GOTO statements that must specify line numbers.)

But besides the mechanical difficulties presented by numbered BASICS, they suffer from a fundamental conceptual flaw, which is that they don’t convey much information about what tasks specific sections of the program perform. The statement \texttt{IF NOT TRUE THEN GOSUB 2350}, for example, doesn’t tell us very much, other than that a test is being performed and a subroutine of some sort will be called if the result is false. On the other hand, the statement \texttt{IF NOT true THEN GOSUB Recover-FromError} describes what kind of activity will be performed if the result of the test is false.

Microsoft solved these problems with the version 2 series of BASIC by eliminating line numbers. Now we can use labels to describe modules of code and write true structured programs. However, thousands of useful and entertaining programs exist that were written with line-numbered BASICS, and Microsoft knew that many users of BASIC 2 would want to run these as well. So, Microsoft made BASIC 2 “backwards compatible” by allowing it to run programs that use line numbers. However, this doesn’t solve the real problem of line-numbered BASIC programs, which we just described, or allow us to easily modify them for our own individual needs.

We could go through an original line-numbered program and convert the line numbers to alphanumeric labels. Unfortunately, this involves more than simply deleting all the line numbers. We would have to
find all the statements that refer to a line number, such as GOTOs and GOSUBs, assign a label to that number, and replace the number referred to with the label. We would have to make sure that all references throughout the program to the same number were given the same label; that is, we would have to build a “symbol table” with each number and its corresponding label, and search the table each time a reference was encountered. Finally, we’d have to attach the labels to the lines that are referred to elsewhere in the program, and then delete the line numbers from all of the lines. You probably agree that as useful as the new labels would be, it wouldn’t be worth the effort, unless you happen to enjoy editing code for hours and hours.

Well, MacConvert provides a solution to the problem of converting these old programs to the new label format. This utility will take a program written in version 1 of Microsoft BASIC for the Macintosh and write out a new version with all the necessary labels substituted for line numbers. You’ll still have to edit the program, because MacConvert does create rather boring labels like LABEL1, LABEL2, and so on. However, MacConvert does enough of the dirty work to make translating your old programs an attractive proposition, especially if you intend to make further additions or modifications to them in the future.

What if you don’t have any programs written in version 1? MacConvert can still be useful to you. Many programs written in other versions of Microsoft BASIC that also use line numbers (such as BASIC-80 for CP/M systems), and even “cousins” like Applesoft BASIC, can be typed into your Macintosh “as-is” and converted to the numberless label format. Also, if you have BASIC programs on disk for another computer, you can transmit them to your Macintosh via a modem or serial line, for instance, and convert them as well.

Obviously, the amount of work you’ll have to do to get the original program to actually run on the Macintosh depends on how different its BASIC syntax is from Microsoft BASIC for the Macintosh. However, the MacConvert conversion would be an important first step that would make it easier for you to see the program’s structure, which would definitely aid in the adaptation of the program to the Macintosh. If the Mac is your first computer, you may not realize that thousands of useful and entertaining program listings can be found in books and magazines for the Apple II, for example. MacConvert can help you tap this vast and varied reservoir of almost free software!
How to use MacConvert

There are a number of steps involved in using MacConvert. You need to get your program ready by putting it into a form that MacConvert can read, then run MacConvert, and then make whatever additional changes are necessary to run the program with version 2. The following sections show you how. For those of you who don't have an old BASIC 1.0 program lying around, an actual program example is included.

Preparing your program for conversion

Before you use MacConvert, you must get the BASIC program you want to convert into a form that MacConvert can read. If the program is a Microsoft BASIC version 1.0 program on disk, it should be ready for conversion. If the program is listed in a book or magazine, simply run BASIC 2, choose New from the File menu to open a new file, and type in the listing, line numbers and all. If the program exists on disk for another type of computer, you'll have to transmit the program to the Mac using a modem or serial line and a telecommunications program.

When you've finished entering the program, choose Save from the File menu; when the dialog box appears, click the Text button at the bottom left of the window. This is important: A program file (other than a BASIC 1.0 file saved under that version of BASIC) must be of "text" type for it to be converted by MacConvert. BASIC 2's default is the "compressed" type, so you have to remember to select Text if you're typing in and saving a program under version 2. It's also a good idea to save a backup copy using the Save As option to protect against mishaps.

If the disk containing MacConvert is close to full, you should either free up some more space on that disk, or transfer MacConvert to a disk with more space on it. MacConvert needs about as much available space as the program file to be converted contains, because it creates a temporary file as part of its processing. This file will be deleted automatically after processing, however.

Using MacConvert

When you start MacConvert, the title screen shown in Figure 1 appears. The title BASIC 1 appears in the upper left corner of the screen, then scrolls toward the MacConvert rectangle at the top center of the screen, and emerges as BASIC 2 to the right of the rectangle. This bit of animation is part of our desire to use some Madison Avenue techniques to catch your eye, and to show graphically what the utility does: converting BASIC 1 (or
This program does most of the work involved in converting programs originally written in Microsoft BASIC 1 to Microsoft BASIC 2. It searches through the whole program to find all lines referenced by GOTO, GOSUB, and other keywords that specify line numbers. It replaces all references to a line number with references to a label, then it goes through the file and removes the line numbers at the beginning of each line. Any lines that are referenced elsewhere in the program have the proper label inserted in front of them. After this process is done, you might still have to fix other parts of the code before the program will run.

Choose 'Convert' from the MacConvert menu to convert a program.

FIGURE 1. The MacConvert title screen

to some extent, non-Mac versions of BASIC) to BASIC 2. You'll also see a brief summary of what the program does, along with the instruction to choose Convert from the MacConvert menu to convert a program.

When you pull down the MacConvert menu, you'll see that the options available are Convert, Instructions, Quit to BASIC, and Quit to Desktop. The Instructions option is dim because the instruction text is already on the screen. When you choose Convert from the menu, a dialog box appears asking for the name of the file containing the program you wish to convert. (If the file you want to work with doesn't appear in the list of files in the dialog box, it isn't stored as a BASIC 1 or text-type file; refer back to the preceding section to find out how to get the program into a text-type format.)

Once you've selected the file to convert to BASIC 2, the conversion proceeds without any further input on your part. As you can see in Figure 2 on the next page, MacConvert keeps you informed of the progress of the conversion by showing you which of the two passes it is in and which line it is working on. We'll see what MacConvert is actually doing in each of the passes when we explore how the program works.
Converting an example program

In case you don’t have an “old-fashioned” BASIC program handy for conversion, we’ve invented the sample program in Figure 3 as it might appear in the BASIC list window for you to practice with. It’s the sort of little program that might have appeared in Mighty Mite’s Journal of Amazing 8K Basic Games, a mythical publication of the latter 1970s. As a product of those benighted times, it of course contains line numbers. The job of this program is to create a personality for a role-playing game. It asks the player for the name of the character, and then asks whether the character is to be a fighter or a mage (wizard). If the character is a fighter, the subroutine at line 100 randomly determines and then prints the fighter’s strength: Otherwise, the program assumes the character is a mage and calls the subroutine at line 200 to determine and print the mage’s intelligence. (As we all know from classic fantasy, fighters have no brains and wizards can barely lift their wands!)
10 REM Role Playing Personality Program
20 CLS: RANDOMIZE TIMER
30 INPUT "What's your character's name ", N$
40 INPUT "Do you want your character to be a Fighter or Mage (F/M)", A$
50 GOSUB 400 : REM Get character land
70 IF UCASE$(A$) = "F" THEN GOSUB 100 ELSE GOSUB 200
80 PRINT: INPUT "Want to do another?", A$
90 IF A$ = "Y" OR A$ = "y" THEN 30 ELSE 299
100 REM Get fighter strength <=== First subroutine
110 STRENGTH = INT (18 * RND(1)) + 1
120 PRINT "Warrior "; N$; " of "; LAND$; " has a strength of "; STRENGTH
130 RETURN
200 REM Get Mage intelligence <=== Second subroutine
210 INTELLIGENCE = INT(18 * RND(1)) + 1
220 PRINT "Mage "; N$; " of "; LAND$; " has an intelligence of "; INTELLIGENCE
230 RETURN
299 END
400 REM Determine character home
410 LAND = INT(RND(1) * 4 + 1)
420 ON LAND GOTO 430,440,450,460
430 LAND$ = "Sardinia" : RETURN
440 LAND$ = "Thrace" : RETURN
450 LAND$ = "Cythera" : RETURN
460 LAND$ = "Nome" : RETURN

FIGURE 3. The line-number example program

After typing in this little gem and saving it with the text option, we ran MacConvert and then printed the resulting file shown in Figure 4 on the next page. We think you'll agree that the converted version has a clearer structure. The indentation of the code following each label helps you see the distinct blocks of code. The first major block, designated LABEL4, is revealed to be an infinite loop, since it ends with a THEN LABEL4. In the original program, you'd have to look back and find line 30 to see where control returns after a character has been created, and it would take a moment or two to realize what's happening.

We'd probably want to edit the listing to give meaningful names to the labels. LABEL4 could become CreateACharacter, while the subroutine LABEL2 could be called MakeFighter, and that at LABEL3 could become MakeMage. We could use BASIC's Replace option from the Search menu to search and replace all occurrences of a given string, such as LABEL3, to make sure that all references to a given label get changed.
REM Role Playing Personality Program
CLS: RANDOMIZE TIMER
LABEL4:
    INPUT "What's your character's name ", N$
    INPUT "Do you want your character to be a Fighter or Mage (F/M)", A$
    GOSUB LABEL1: REM Get character land
    IF UCASE$(A$) = "F" THEN GOSUB LABEL2 ELSE GOSUB LABEL3
    PRINT: INPUT "Want to do another?", A$
    IF A$ = "Y" OR A$ = "y" THEN LABEL4 ELSE LABEL5
LABEL2:
    REM Get fighter strength <= First subroutine
    STRENGTH = INT (18 * RND(1)) + 1
    PRINT "Warrior "; N$; " of "; LAND$; " has a strength of "; STRENGTH
    RETURN
LABEL3:
    REM Get Mage intelligence <= Second subroutine
    INTELLIGENCE = INT (18 * RND(1)) + 1
    PRINT "Mage "; N$; " of "; LAND$; " has an intelligence of "; INTELLIGENCE
    RETURN
LABEL5:
    END
LABEL1:
    REM Determine character home
    LAND = INT (RND(1) * 4 + 1)
    ON LAND GOTO LABEL6, LABEL7, LABEL8, LABEL9
LABEL6:
    LAND$ = "Sardinia" : RETURN
LABEL7:
    LAND$ = "Thrace" : RETURN
LABEL8:
    LAND$ = "Cythera" : RETURN
LABEL9:
    LAND$ = "Nome" : RETURN

FIGURE 4. The program in Figure 3, after processing by MacConvert

How MacConvert works

The program outline for MacConvert is shown in Figure 5. The structure follows a not-uncommon pattern of doing all the work in menu events, with the main program being simply an endless loop that is interrupted by menu events as they come up. Exiting is accomplished through our standard quit selections (Quit to BASIC and Quit to Desktop).

Before we get into the details of how each of the sections of this program works, it's important for you to think about the underlying techniques that we will be using. The techniques revolve around the idea of
building a table of references, or symbol table, by searching for certain key phrases or commands. When the commands or phrases are found, the line that they are in is parsed. What this means is that the relevant components of the line are separated. This way, the keyword and the line number that it refers to are identified, and anything else in the line is ignored. The referenced line number is checked to see if it has already been saved in the table of references: If it has, it is associated with a label; if it hasn't, it is then assigned a new label which is saved in the symbol table.

**Initialization**
- initialize data array and constants for graphics
- initialize data array and constants for file conversion
- define Output window and scrolling information (InitScrollInfo)
- create and enable MacConvert menu
- display title screen and instructions (ShowConvert, Instructions)

**Main Program**
- WHILE (endless loop interrupted by menu events)
- WHILE user does not choose Convert from menu
- scroll title string in window (ScrollOnce)
- WEND
- do conversion (Convert)
- WEND

**Menu Events**
- MenuEvent: disable menu
- branch to appropriate subroutine
- re-enable menu
- ConvertMenu: set conversion flag
- Instructions: display instructions
- BasQuit: quit to BASIC
- DeskQuit: quit to Mac Desktop

**Subroutines**
- Convert: convert file
  - disable menu items
  - get names of source and destination files
  - open source and temp files for conversion
  - do first pass (PassOne)
  - close files
  - open temp and destination files
  - do second pass (PassTwo)
  - close files
  - display message that conversion is complete, kill temp file
  - re-enable menu items

(continued)
PassOne: build symbol table and change line references
    display message
    WHILE not end of source file
    increment and display line number
    read line of text
    check for each key word (CheckWord)
    print converted line to temp file
    WEND

CheckWord: check text line for key words
    WHILE there is a line number following key word
    scan line for key word (SkipSpaces, ExtractNum)
    add number to symbol table (AddSymbol)
    replace number with appropriate label
    WEND

PassTwo: do second pass of translation
    display message
    WHILE not end of temp file
    increment and display line number
    read line of text
    extract the line number (ExtractNum)
    if in symbol table, attach corresponding label to line (OutputLabel)
    print converted line to destination file
    WEND

OutputLabel: send label to output file
ExtractNum: extract number from string
SkipSpaces: skip spaces in line of text
AddSymbol: add line number to symbol table
InitScrollInfo: initialize scrolling information
ScrollOnce: scroll words to right one time
ShowConvert: show "MacConvert" inside box
ClearBottom: clear bottom of screen

Figure 5. Program outline (continued)

The symbol table in MacConvert consists of the two arrays, lines! and symbol$, where each of the line numbers and corresponding labels created by MacConvert is stored. This symbol table is then used to build the new list of code. With this in mind, let’s take a closer look at how Mac­Convert works.
Initialization of graphics

Most of the initialization in [Init 1] concerns the scrolling that will be used to show BASIC 1 being "converted" to BASIC 2 on the screen. The idea behind the program scrolling is to "slice" the phrases BASIC 1 and BASIC 2 into a number of "frames" that will be used to give the appearance of motion to the frames as they are scrolled. The strings BASIC 1 and BASIC 2 each contain 34 frames, as defined by the statement frameNum = 33, and they're stored in the arrays num1Pict% and num2Pict%. The array pen%, which has two elements, 0 and 1, positions the pen for printing text.

The next set of arrays and constants is involved with the actual conversion of the file to BASIC 2. The temporary file called MacCnvrt.??? in the listing is used to hold the results of the first conversion pass; that is, MacConvert creates a temporary file so that it does not have to erase the original source file. This way, you can still compare the final results with the original file. This is a safety precaution: If you forget to make a backup before you start, you might wind up in trouble if MacConvert erased your original file without creating this temporary file to fall back on.

What kind of trouble could you be in? There are a number of things that could go wrong. A power failure, a defective disk, not enough room on the disk, and hardware problems happen from time to time. There are a couple of problems that are unique to conversion efforts in particular. These occur when the computer you are using doesn’t have enough memory to work with (this can happen with the 128K Macintosh), or the program to be converted is too large, causing the array indexes for the symbol table and line numbers to go out of bounds.

The array symbol$ will hold the symbol-table labels, while the array lines! will hold the corresponding line numbers. (Remember that the exclamation point indicates a single-precision integer variable.) Since line numbers in Microsoft BASIC for the Mac only go up to 65529, a single-precision integer has enough room for any line number. We use them in MacConvert to save memory. We have dimensioned the arrays to allow for a maximum of 500 lines in the original program to be converted, which should be plenty for most BASIC line-numbered programs you're likely to encounter. If you ever do need more, or you get an array-bounds error when you try to convert a program, you can redimension the arrays to a higher number. Be aware, though, that you'll eventually run into memory problems on a 128K Mac if you redimension the arrays too high.

Since it's very unlikely that every line in the program will reference another line, you may be wondering why the symbol table has room
for as many labels as there are lines. Well, it’s safer not to make assumptions about what a “typical” program to be converted will look like. If MacConvert could handle up to 500 lines of code but only 100 labels, we’d have to document this, and then you would have to count line-number references! As it is, in extreme circumstances the most you might have to do is estimate the number of lines in the program, which is much easier than counting up all the line-number references—indeed, many word processors will count the lines in a file for you. In most cases, of course, you won’t have to worry about the number of lines or references in the program.

The number of BASIC keywords and an array key$ to hold their names is specified in [Init 1]. Using this setup, and the CheckWord subroutine [SR 3], it wouldn’t be hard to add another keyword. For example, if you wanted to convert from a BASIC that has CALL statements to line numbers, you could do this with only a few changes to the code. You may have noticed that the ON GOTO and ON GOSUB keywords aren’t included in the six keywords in key$—good for you! These will be handled using a little trick we’ll explain later.

The string validDigits$ will be used for determining which characters in a line constitute a line number. The string passOK$ consists of those characters that should be skipped over when parsing (determining the components of) the line. These characters are the space, the horizontal tab, and the line feed. (The horizontal tab is often put in source code to indent lines, while the line feed continues a line on the next screen line. The Output window’s type, size, and position are set in [Init 2] and the window is given the title MacConvert.

The SCROLL statement

Next, InitScrollInfo [SR 9] is called to set up the opening scrolling display at the top of the screen. Although this piece of code doesn’t have any real effect on the conversion process, it reveals the operation of the SCROLL statement. This subroutine starts innocuously enough by printing BASIC 1 at the left of the MacConvert box, with the 1 in boldface type. But some very interesting things start to happen when the FOR...NEXT loop GETs frameNum frames of the string BASIC 1 and stores the successive frames in the array num1Pict%. Let’s look at the first two frames to see how this works in action.

The first time through the loop, the index (n) is 0. Note that the constant hSep has a value of 2. If you substitute these values into the GET statement, you’ll find that the first frame has (10,10) as the coordinates
for its upper left corner, and (12,25) for the lower right corner. (GET, you'll recall, needs these two corners to define the screen image to be stored in the array.) This frame is therefore two pixels, or $hSep$ pixels wide, and $25 - 10$, or 15 pixels high. The next time through the loop, the index ($n$) becomes 1, so the next frame has $(12,10)$ as its upper left corner (the constant $hSep$ was set to 2, and $1 	imes 2 = 2$). The lower right corner is now at $(14,25)$; if you compare these coordinates with those obtained the last time around, you'll see that this is the next two-pixel-wide frame to the right. The loop continues until the whole string on the screen has been divided into frames and stored away. The same procedure is followed to get the frames for the string BASIC 2 on the right side of the MacConvert box. These frames are saved in the num2Pict% array. We'll see how these two sets of frames are used for scrolling when we look at ScrollOnce [SR 10], just as soon as we start looking at the main program loop.

InitScrollInfo's final task is to define the rectangles within which BASIC 1 and BASIC 2 will be scrolled. They are the same size, and on opposite sides of the MacConvert rectangle in the top center of the screen. When you watch the title screen in action, BASIC 2 doesn't appear to emerge from the MacConvert rectangle until BASIC 1 has disappeared into it and has had enough time to reach the other side. To do this, we have to scroll just BASIC 1 a certain number of times before we start scrolling BASIC 2. The number of times to scroll BASIC 1 is equal to the distance from the left side of BASIC 1's scrolling rectangle to the left side of BASIC 2's rectangle, divided by the width of one frame, or $hSep$. That's the value assigned to the variable justOne.

Finishing the title screen

Since this program will be menu driven, the MacConvert pull-down menu is created next. Menu events will be handled by MenuEvent at [ME 0]. In preparation for the main program, the rest of the title screen needs to be shown, and the scrolling started. ShowConvert [SR 11] starts by drawing the MacConvert box. Note that the subroutine uses coordinates derived from the coordinates of the two scrolling boxes on either side of it. The word MacConvert is then displayed inside the center box. After ShowConvert is finished, Instructions [ME 2] is called to display the text of the program description. It disables the Instructions option, since that function is now in progress, and calls ClearBottom [SR 12] to clear out the area where the text progress messages will be displayed. When ClearBottom is finished processing, control returns to Instructions, and the text is printed in the cleared-out window.
ClearBottom makes sure that any text left over from other functions (such as the conversion status display with pass and line numbers) is erased. To do this, it employs the Macintosh ERASERECT routine. You'll probably recall that routines of this kind need to be passed an array whose elements are the top, left, bottom, and right boundaries of the rectangle in which the action will take place. ClearBottom gets the boundary for the top of the rectangle \((tRect(0))\) by adding 10 to the value of \(oneRbot\), the bottom of the right scrolling window (either window would do, since they're at the same height). The left side of the rectangle is the left side of the screen, or 0. The bottom is the height of the Output window, returned by the WINDOW(3) function, minus the top of the window in \(tRect(0)\). The right boundary is the width of the Output window, returned by WINDOW(2). Since these latter two dimensions will vary with the size of the Output window, this procedure ensures that the correct area will be erased regardless of the window's size. Specifying dimensions in this way requires a little more care and thought, but makes for a more versatile and easier-to-modify program.

The main program

We've now reached the main program [Main]. We've already pointed out that this is just an endless loop, since it tests WHILE \(true\), and \(true\) has a value of \(-1\) ("NOT false," or NOT 0). The inner WHILE...WEND loop checks the value of \(convertFlag\), which is initially \(false\), to see whether the user has selected Convert from the MacConvert menu. If not, ScrollOnce [SR 10] will be called.

This subroutine first uses a SCROLL statement with the upper left and lower right coordinates of the first scrolling rectangle (the one with BASIC 1), the width of a frame \(hSep\), and a 0. These parameters, in effect, tell SCROLL: "Within the rectangle specified, scroll everything 2 pixels to the right, but don't scroll anything up or down (because the change in the y direction is 0)."

This works fine, but if that's all we did, the words would simply disappear to the right as they were scrolled. This is because the SCROLL routine doesn't "wrap around"; that is, graphics that disappear to the right don't automatically reappear on the left. Therefore, we have to restore one frame of our string to the left every time a frame disappears on the right. This is done, as you've probably guessed, by GETting the frame corresponding to the one that disappeared from our array of stored frames (in this case, \(num1Pict\)), and PUTFting it in the rectangle in the leftmost position. Before we arrived at the main program, \(num1\) and \(num2\) were set to
33, the last frame; so the first time through this loop frame 33, the rightmost one, disappears and then we put it back on the left side of the rectangle. The number of the frame to be restored is then reduced by 1, so that next time through the loop frame 32 will disappear and be replaced to the left. If this number goes below 0, it is set back to 33 to start over again.

Until scrolling has happened *justOne* times (that is, long enough for the words BASIC 1 to appear to go into the MacConvert box and begin moving out the other side), “just region one” will be scrolled by each call to ScrollOnce. (Recall that *justOne* was determined in the initialization section of the program.) After that, both regions will be scrolled each time the subroutine is called. The code for scrolling region two (BASIC 2) is therefore controlled by an IF...THEN statement that checks to see if *justOne* has been exceeded: If it has, it’s time to start scrolling the BASIC 2 rectangle as well. ScrollOne will be called, and both regions will continue to scroll, until Convert is selected and *convertFlag* becomes *true*. When this happens, we finally get down to the real business of the program, the file conversion.

File conversion

File conversion is handled by Convert [SR 1], which “masterminds” this two-stage process. The subroutine starts by disabling the Convert and Instructions menu items so the conversion process won’t be interrupted by them once it has begun. The bottom of the screen is cleared by calling ClearBottom, which we’ve already discussed. The name of the file to be converted is obtained from the user via the FILES$ function. Only two types of files are eligible for conversion, and thus will have their names shown in the selection box: standard Macintosh text files, and files with type “MSBA,” which were created by BASIC 1. If the user doesn’t select a file name, the conversion is canceled and control returns to the main program. The name of the file to be converted is assigned to *infile$* by the FILES$ function.

Next, the versatile FILES$ function is called upon again and asks for the name of the file name for the converted file to be saved under. This name is assigned to *outfile$*. If no valid name is given, the conversion routine is exited. After this second dialog box disappears, a call to ShowConvert restores the MacConvert animation area.

The file to be converted is now opened for input, and the temporary file, *MacCnvt.???,* whose name is in *tempFile$,* is opened for output. The first pass will read the input file and write it with modifications to the temporary file.
Pass One

PassOne [SR 2] is called to perform the first pass of the conversion. As you can see in Figure 2, after the message for Pass One is displayed, MacConvert prints out a number representing how many lines it has read (20 in Figure 2). Note that these are simply numbers in sequential order: The first line is 1, the second 2, and so on. Don't confuse them with the line numbers given to the lines in the program being converted. The Macintosh GETPEN routine is called to save the pen location in pen% so the line number can be printed later on in the right place.

The loop beginning WHILE NOT EOF performs the actual work of Pass One. The variable count, containing the number of the current line, is incremented, and that number is printed on the screen at the saved pen location. The line is read into the variable text$ using the LINE INPUT statement, and its length is checked. If the length is 0, the line is simply a blank line (that is, just a carriage return), and control then jumps to the label NextLine. This kind of line needs no processing, so it's simply counted and printed in the output file. If the line is not blank, it must be checked against the six keywords that might be followed by one or more line numbers. The FOR...NEXT loop's last action is to call CheckWord [SR 3] to check each of these keywords.

CheckWord checks for the presence of the keyword in key$(key). Since the loop in PassOne calls the keyword when the values in key are from 1 to 6, the loop will check all six keywords, one per call. The position of the keyword as returned by the INSTR (in-string) function is assigned to the variable found. A WHILE...WEND loop tests the value of found. If this position is 0, the keyword was not found, control falls through the bottom of the loop, and CheckWord returns to PassOne.

If the keyword was found (found > 0), we need to look for a number or numbers following the keyword. The variable chrPos is set to found (the position of the first character of the keyword) plus the length of the keyword. This makes chrPos "point to" the character position just after the keyword. An inner WHILE...WEND loop is then used to look for numbers. First, SkipSpaces [SR 7] is called. This has a WHILE...WEND loop that looks at the characters starting at the current position of chrPos, and advances the pointer to the right (by incrementing chrPos) as long as the character encountered is in passOK$; that is, it's a blank, horizontal tab, or line feed. If a character that is not one of those is encountered, or the end of the line is reached without encountering a valid character, SkipSpaces returns to CheckWord.
As an aside, the procedure that we are describing here is called parsing, and is very similar to what BASIC does when it runs a program: It looks at each line of code and determines if it is a valid statement. If it is, BASIC then checks to see if there are any BASIC commands which transfer control to another section of the program, and if there are, it determines where that control gets transferred to.

After returning from SkipSpaces, chrPos is now at the first character position of what might be a number following the keyword. (It doesn't have to be a number, of course. For example, if the keyword is THEN, the statement might read IF p = 1 THEN 1000, but it might instead be IF p = 1 THEN PRINT P$.)

Since what we're interested in is line-number references in the BASIC statements, ExtractNum [SR 6] is called to see if there's a number. It first calls SkipSpaces; the first time ExtractNum is called for a given line, SkipSpaces will already have been called, and SkipSpaces will return immediately, but if ExtractNum is called a second or subsequent time for the same line, it will probably have to skip over spaces or other characters to get to the next number. This happens with ON GOTO or ON GOSUB statements, which are followed by two or more line numbers. ExtractNum uses a WHILE...WEND loop to scan the characters starting at chrPos (stored as iStart) to see if they are valid digits. As long as valid digits are found, the counter variable i is incremented. When the end of the line or a character other than a valid digit is found, control drops through the bottom of this loop with the variable i now equal to the original chrPos plus the length of the number found.

If i is still equal to iStart, there was no number, so ExtractNum sets the “number's” value to -1 and its length to -1, and returns. Otherwise, the length of the number is i - chrPos, and the value of the number (which, being a line number, is an integer), is found by extracting the number string. This number string, which starts at chrPos and is i - chrPos characters long, is extracted from text$. It is evaluated by applying the VAL function to it. This value is assigned to numVal. Next, numLen is added to chrPos to move the “pointer” to the character following the number, and control jumps to the label ExtLExit (skipping over the “not a number” code), and ExtractNum returns to CheckWord.

If the length of the “number” is -1, or its value is 0, CheckWord “knows” that no number was found, or it was 0, so control jumps to the label NotNum, and then back to the top of the WHILE changeNums loop in CheckWord to look for more numbers. (The reason we don't want to generate a label for 0 is that it has a special use; for instance, in statements like
ON ERROR GOTO 0, which disables error trapping.) If a number is found, the variable lineNumber is assigned the value in numVal. AddSymbol [SR 8] is called to add this line number to the symbol table. The symbol table consists of two arrays: lines!, which contains line numbers, and symbol$, which contains the corresponding labels created by MacConvert.

First, the elements of the lines! array, which are line numbers corresponding to previously assigned labels, are checked to see if the newly found line number is already there. Only the elements up to the current value of labels—the number of labels placed in the symbol table so far—need to be checked. If the number is found in lines!, then the label is one that already exists in symbol$(check); this label is assigned to symbol$ and AddSymbol returns to CheckWord. If the number was not found in the lines! array, a new label must be assigned and the number and label must be recorded in the symbol table. This is done by adding 1 to the total labels (labels), and creating a new label by concatenating (combining) the string LABEL with a string made from the current value of labels. The string is extracted starting from its second character by MID$ because STR$ precedes numbers it converts with a leading space (if positive). We don't want this space, since labels can't have spaces in them. The result is that if, for example, we've just added the twentieth label, the label LABEL20 will be added to the symbol table by putting it in the symbol$ array. The corresponding line number is put in the lines! array. AddSymbol is now finished, and control returns to CheckWord.

Next, the line number found in the original code is replaced by the label established for it. temp$ is assigned the part of the line (text$) up to just before where the line number was found, and symbol$, containing the label, is added to it. (Remember that this label is either a previously used one found by AddSymbol, or a new one added by it.) The part of text$ containing the text following the line number is then concatenated with temp$ to make a new text$ containing the label in place of the line number. Finally, chrPos is updated so it points to the character position following the label, which is the next character that needs to be checked.

Next, the part of text$ starting at the position where a keyword was last found (lastFnd) is checked for the word ON. This is necessary for dealing with the added complication of the ON GOTO and ON GOSUB statements, which have multiple line numbers following them. If the keyword currently being considered by CheckWord isn't 3 or 4 (GOTO or GOSUB), or ON wasn't found, we're not dealing with one of these ON statements. (We have to check for both keyword 3 or 4 and ON so we can eliminate other kinds of ON statements like ON DIALOG or ON MOUSE.)
If the statement wasn’t ON GOTO or ON GOSUB, control is sent to the label NoOn, and then back up to the top of the changeNums loop.

If we do have an ON GOSUB or ON GOTO, only the first number in its number list has been taken care of, so SkipSpaces is called to get to the next number. If the first “not OK” character reached by SkipSpaces wasn’t a comma, then the end of the number list has been reached, and control goes to the label NoMore and then down to the WEND and back up to the WHILE changeNums loop. If a comma was found, 1 is added to chrPos to skip past it. The flag changeNums is set to true, so when control falls through the remaining labels and back up to changeNums, the whole loop will be executed again and the next number will be processed.

If control has returned to the top of the WHILE changeNums loop with no number having been found, or the number was 0, or only one number was processed (that is, a keyword other than ON GOTO or ON GOSUB), changeNums is false, and control drops through to the bottom of CheckWord’s main loop. The variable lastFind is set to found (the starting position of the last keyword), and the next instance of the keyword is searched for and its position is assigned to found. Control then goes to the top of the main loop: If a keyword was found, found is greater than 0, and the whole process of looking for numbers and handling possible ON GOTOs and ON GOSUBs is gone through again. If a keyword is not found this time, CheckWord is finished, and control is returned to PassOne.

"Whew!" we hear you say. "If all those operations were needed to check just one line, are we ever going to get to the end of this program?" A word of cheer is in order now: We’re just about done with PassOne. After the line is checked for each keyword by the FOR...NEXT loop, the possibly modified line is printed to the output file. The WHILE NOT EOF loop continues until all the lines in the input file have been checked and processed as required. Control then returns to Convert, and the input and output files are closed. The temporary file created by PassOne is opened for input for Pass Two [SR 4], and the file name specified by the user is opened for the final output from Pass Two.

**Pass Two**

PassTwo [SR 4] is now called to perform Pass Two of the conversion process. You’ll be happy to learn that its operation is considerably less complicated than that of PassOne. PassTwo starts by printing a message describing its activity on the screen. (When a lengthy process is going on, it’s important to inform the user about what is going on and the amount of progress made.)
A WHILE...WEND loop reads through the input file, which is the temporary file containing the results of PassOne. The value of \textit{count} (the number of the line being processed), is displayed in the same way used in PassOne: A line is read from the temporary file, and its length is checked. If the length is 0, the line is just a carriage return, and doesn't have a line number, so control jumps to the label NextLine2, the line is printed, and we go back to the top of the WHILE...WEND loop to get the next line. Otherwise, the line has a number and the line number has to be removed. Additionally, if this line number is referred to by any other part of the program, the appropriate label has to be attached to it. ExtractNum is called upon to get the value of the line number. If the value returned by ExtractNum is $-1$, no line number was found, so control then jumps to the label NotLineNum, the line is printed, and we go back to the top of the main loop to get the next line.

If a number was found, an inner WHILE...WEND loop checks the symbol table (the part in the \textit{lines[]} array containing referenced line numbers) to see if the line number has been referenced. If it is found, OutputLabel [SR 5] is called to print the corresponding label to the output file. A colon is appended first, since labels themselves, as opposed to references to them, must end in a colon. The label is then printed on its own line in the output file. This helps it stand out from the section of code to which it is attached. The flag \textit{checking} is set to \textit{false} and control drops through the bottom of the WHILE checking loop.

If the label is not found in the symbol table by the time the value of \textit{check} exceeds the number of labels, control drops through the WHILE checking loop. You can see that whether a label is found and output or not, we eventually get past the loop and reach the statement that removes the line number from the line of code. Since ExtractNum left the \textit{chrPos} pointer just past the line number, we get rid of the number by adding four spaces to the part of the line starting at \textit{chrPos}. This has the effect of indenting all the lines in the output file four spaces, again helping us visualize the blocks of code with their associated labels. We then reach the statement that prints the modified line, and return to the outer loop to get the next line. When the last line has been read, PassTwo and the conversion process are finished. Control returns to Convert.

Convert closes the files used in PassTwo, prints a message saying that the conversion is finished, and KILLS the temporary file so it isn't left cluttering up the disk. Control now returns to the main program in [Main].
ConvertFlag is set to false, control goes back to the top of the endless loop, and because convertFlag is false, the scrolling loop is started up again, to continue until another menu event is selected.

We've already mentioned how the description is printed, and you're familiar with how the two quits (Quit to BASIC and Quit to Desktop) work, so it's not necessary to go into those menu events. In other words, we're finished!

Suggestions for MacConvert

MacConvert is an example of a class of programs known as preprocessors. What this term describes are programs that, like MacConvert, make specific changes to another program. Another, more general term is filters. The logic behind filter programs is basically the same as that for preprocessors, except that they make a wider range of modifications to files.

Now that you understand the structure of a preprocessing program, you might want to write a filter or preprocessor program of your own. For example, you could create a filter program that would make changes to a word-processed document file that your word processor can't handle, such as stripping out special control characters before you sent the file via modem, or converting ASCII files back to your word processor's format. An example of a preprocessor you might create is a program that would allow you to alter other programs to include structures not available in the language you are using. For example, although Microsoft BASIC for the Macintosh doesn't have a CASE statement (a multiway branching structure that can handle several tests at once), you could write a preprocessor that would accept such a statement and convert it to the appropriate sequence of IF...THEN...ELSE or ON...GOTO statements.
MacConvert program listing

MacConvert
Convert MS BASIC 1 programs to MS BASIC 2

**INIT 1**

IF FRE(0) < 22000 THEN CLEAR , 22000
false = 0 : true = NOT false
DIM pen%(1)
frameNum = 33
hSep = 2
DIM num1Pict%(20,frameNum)
DIM num2Pict%(20,frameNum)
DIM tRect%(3)
tempFile$ = "MacCnvrt.????"
DIM symbol$(500)
DIM lines!(500)
keyWords = 6
DIM key$(keyWords)
key$(1) = "THEN"
key$(2) = "ELSE"
key$(3) = "GOTO"
key$(4) = "GOSUB"
key$(5) = "RESTORE"
key$(6) = "RESUME"
validDigit$ = "0123456789"
passOK$ = " " + CHR$(10) + CHR$(9)
WINDOW 1, "MacConvert", (2,40) - (508,337), 1
GOSUB InitScrollInfo
GOSUB ShowConvert
GOSUB Instructions
num1 = frameNum : num2 = frameNum

**INIT 2**

GOSUB Instructions
WHILE true
    WHILE NOT convertFlag
        GOSUB ScrollOnce
    WEND
    GOSUB Convert
    convertFlag = false
    WEND

MAIN PROGRAM

MENU EVENTS

MenuEvent:
    MENU 6, 0, 0  ; disable menu bar
    MENU 6, 2, 1  ; enable instruction option if disabled
    item = MENU(1)
    ON item GOSUB ConvertMenu, Instructions,, BasQuit, DeskQuit
    MENU 6, 0, 1
    RETURN

ConvertMenu:
    convertFlag = true
    RETURN

Instructions:
    display instructions
    MENU 6, 2, 0  ; disable option
    CALL TEXTFONT(1)
    GOSUB ClearBottom
    LOCATE 5, 1
    PRINT "This program does most of the work involved in converting programs"
    PRINT "originally written in Microsoft BASIC 1 to Microsoft BASIC 2."
    PRINT "It searches through the whole program to find all lines referenced"
    PRINT "by GOTO, GOSUB, and other keywords that specify line numbers. It"
    PRINT "replaces all references to a line number with references to a label,"
    PRINT "then it goes through the file and removes the line numbers at the"
    PRINT "beginning of each line. Any lines that are referenced elsewhere in"
    PRINT "the program have the proper label inserted in front of them. After this"
    PRINT "process is done, you might still have to fix other parts of the code"
    PRINT "before the program will run."
    PRINT "Choose 'Convert' from the MacConvert menu to convert a program."
    RETURN

BasQuit:
    user wants to quit to BASIC
    MENU RESET
    CLEAR
    END

DeskQuit:
    user wants to quit to Mac Desktop
    MENU RESET
    CLEAR
    SYSTEM
SUBROUTINES

Convert:

`Convert: Convert file`

`MENU 6, 1, 0`  `disable Convert menu item,`
`MENU 6, 2, 0`  `and Instructions menu item`
`GOSUB ClearBottom`

`inFile$ = FILES$(1,"TEXTMSBA")`  `open either text or BASIC 1 files`
`IF inFile$ = "" THEN CancelCnvrt`

`outFile$ = FILES$(0,"Save converted file as?")`  `open either text or BASIC 1 files`
`IF outFile$ = "" THEN CancelCnvrt`

`CLS : GOSUB ShowConvert`
`CALL TEXTFONT(0) : CALL TEXTMODE(0)`
`LOCATE 5,1 : PRINT "Converting " inFile$ " to " outFile$ "."`
`PRINT`
`OPEN inFile$ FOR INPUT AS #1`
`OPEN tempFile$ FOR OUTPUT AS #2`
`GOSUB PassOne`  `PassOne copies from source to temp file`
`CLOSE #1 : CLOSE #2`

`OPEN tempFile$ FOR INPUT AS #1`
`OPEN outFile$ FOR OUTPUT AS #2`  `PassTwo copies from temp file to final file`
`GOSUB PassTwo`
`CLOSE #1 : CLOSE #2`

`PRINT : PRINT "The conversion is now finished."`
`KILL tempFile$`  `make sure temp file isn't left around`
`CancelCnvrt: user wanted to cancel this run`
`GOSUB ShowConvert`
`MENU 6, 1, 1`  `re-enable Convert menu option,`
`MENU 6, 2, 1`  `and Instruction option`
`RETURN`

PassOne: first pass of translation

`This reads the whole source file. It builds a symbol table of all the`
`line numbers that are referenced ('GOTO 10') and at the same time`
`replaces referenced line numbers with labels ('GOTO LABEL43');`
`it writes the entire file out to a temporary file.`

`PRINT "Pass One: Build the symbol table and change line references"
PRINT " to label references."
PRINT "Currently on line: ";`
`CALL GETPEN(VARPTR(pen%(0)))`
`count = 0`
`WHILE NOT EOF(1)`
`    count = count + 1`
`    CALL MOVETO(pen%(1),pen%(0)) : PRINT count`
`    LINE INPUT #1, text$`
`    IF LEN(text$) = 0 THEN NextLine`
`    FOR key = 1 TO keyWords`
`        GOSUB CheckWord`  `check line against each keyword`
`NEXT`
NextLine:
  PRINT #2, text$
WEND
PRINT
RETURN

CheckWord: check text$ for keywords
  found = INSTR(1, text$, key$(key))
  lastFnd = 1
  WHILE found <> 0
    chrPos = found + LEN(key$(key))
    changeNums = true 'execute next loop at least once
  WHILE changeNums
    changeNums = false
    GOSUB SkipSpaces
    numStart = chrPos 'save char at which num starts
    GOSUB ExtractNum 'get number if there
    IF numLen = -1 OR numVal = 0 THEN NotNum 'don't change if not num
    lineNumber = numVal : GOSUB AddSymbol 'add to sym table
      'next two lines replace number with label string
      temp$ = LEFT$(text$, numStart - 1) + symbol$ 
      text$ = temp$ + MID$(text$, numStart + numLen, LEN(text$))
    chrPos = numStart + LEN(symbol$)
  GOSUB SkipSpaces 'is either ON GOSUB or ON GOTO: change all
  IF MID$(text$, chrPos, 1) <> "," THEN NoMore
  chrPos = chrPos + 1
  changeNums = true 'set to true so will keep going down list
NoMore:
  NoOn: 'no more numbers in list
NotNum: 'either wasn't number, or was '0': don't change
WEND
  lastFnd = found
  found = INSTR(found + 1, text$, key$(key))
WEND
RETURN

PassTwo: second pass of translation
  Pass two reads out of the temporary file. It gets the line number of
  each line it reads and searches the symbol table to see if that line
  number is in it. If it is there, then the label that corresponds to that
  line number is inserted before the line. In either case, the line is
  stripped of its line number and written to the final output file.
  PRINT
  PRINT "Pass Two: Remove line numbers and add labels."
PRINT "Currently on line: ";
CALL GETPEN(VARPTR (pen%(0)))
count = 0
WHILE NOT EOF(1)
count = count + 1
CALL MOVETO(pen%(1), pen%(0)) : PRINT count
LINE INPUT #1, text$
IF LEN(text$) = 0 THEN NextLine2
chrPos = 1: GOSUB ExtractNum
IF numLen = -1 THEN NotLineNum

check = 1 : checking = true
WHILE checking
    IF numVal <> lines!(check) THEN NotSame
    GOSUB OutputLabel
    checking = false
NotSame:
    check = check + 1
    IF check > labels THEN checking = false
WEND
text$ = SPACES(4) + MID$(text$, chrPos, LEN(text$))
NotLineNum:
NextLine2:
PRINT #2, text$
WEND
RETURN

OutputLabel:
send label to output file
temp$ = symbol$(check) + ":"
PRINT #2, temp$
RETURN

ExtractNum:
extract number from string
'Given: text$, chrPos
'Return: numVal = int value, numLen = # of chars
'Side affect: chrNum to point to the next char in text$
GOSUB SkipSpaces
i = chrPos : iStart = chrPos
WHILE i <= LEN(text$) AND INSTR(1, validDigit$, MID$(text$, i, 1))
i = i + 1
WEND
IF i = iStart THEN NoNumber
numLen = i - chrPos
numVal = VAL(MIDS(text$, chrPos, i - chrPos))
chrPos = chrPos + numLen
GOTO ExtLExit
NoNumber:
numVal = -1 : numLen = -1
ExtLExit:
RETURN
SkipSpaces: move chrPos along text$ while it points to 'char in passOK$

WHILE chrPos <= LEN(text$) AND INSTR(passOK$, MIDS(text$, chrPos, 1))
    chrPos = chrPos + 1
WEND
RETURN

AddSymbol: add line number to symbol table

FOR check = 1 TO labels
    IF lines!(check) = lineNumber THEN symbol$ = symbol$(check) : RETURN
NEXT
labels = labels + 1
symbol$ = "LABEL" + MIDS(STR$(labels), 2, LEN(STR$(labels)))
symbol$(labels) = symbol$
lines!(labels) = lineNumber
RETURN

InitScrollInfo: initialize what is needed to scroll sideways

' Each word to scroll is divided into FrameNum frames, or pictures of hSep pixels wide. The words are written on the screen, then the frames are read back into picture rectangle.

CLS
CALL MOVETO (10,20) : PRINT "BASIC ";
CALL TEXTMODE(1) : CALL TEXTFACE(1)       'bold, OR onto screen
PRINT "1"
FOR n = 0 TO frameNum              'get frameNum frames from screen
    GET (10 + n * hSep,10) - (10 + (n + 1) * hSep, 25), num1Pict%(0,n)
NEXT

CLS
CALL TEXTFACE(0) : CALL MOVETO (10,20) : PRINT "BASIC ";
CALL TEXTFACE(1) : PRINT "2"       'change back to normal
CALL TEXTMODE(0) : CALL TEXTFACE(0)
FOR n = 0 TO frameNum              'get FrameNum frames from screen
    GET (10 + n * hSep, 10) - (10 + (n + 1) * hSep, 25), num2Pict%(0,n)
NEXT

CLS
oneRleft = 30 : oneRtop = 30
oneRright = 170 : oneRbot = 30 + 15       'and corners of rectangle to scroll BASIC 1 through

twoRleft = 290 : twoRtop = 30
twoRright = 430 : twoRbot = 30 + 15       'justOne is how many times just BASIC 1 is scrolled

justOne = (twoRleft - oneRleft) / hSep
RETURN

ScrollOnce: scroll words to right one time

SCROLL (oneRleft,oneRtop) - (oneRright,oneRbot), hSep, 0
PUT (oneRleft,oneRtop) - (oneRleft + hSep, oneRbot), num1Pict%(0,num1), OR
num1 = num1 - 1
IF num1 < 0 THEN num1 = frameNum
IF times < justOne THEN NotTwo "haven't scrolled enough: don't do BASIC 2
SCROLL (twoRleft,twoRtop) - (twoRright,twoRbot), hSep, 0
PUT (twoRleft,twoRtop) - (twoRleft + hSep, twoRbot), num2Pict%(0,num2), OR
num2 = num2 - 1
IF num2 < 0 THEN num2 = frameNum
times = justOne
NotTwo:
times = times + 1
RETURN

ShowConvert:
' __________________________ show word 'MacConvert' inside box
LINE (oneRright + 1, oneRtop - 10) - (twoRleft - 1, oneRbot + 5), , b
CALL TEXTFONT(0) : CALL TEXTMODE(1)
CALL MOVETO(oneRright + 20, oneRtop + 10) : PRINT "MacConvert"
RETURN

ClearBottom:
' __________________________ clear bottom of screen
tRect%(0) = oneRbot + 10 : tRect%(1) = 0
tRect%(2) = WINDOW(3) - tRect%(0)
tRect%(3) = WINDOW(2)
CALL ERASURECT(VARPTR(tRect%(0)))
RETURN
MacAnimate

Most of you, we’re sure, have probably spent some time with MacPaint, and perhaps with other applications that allow you to create precise drawings with the “FatBits” option. After doing all the work of creating them, have you ever wished that your pictures could come to life and move around on the screen? If you have, MacAnimate is for you!
Animation is the process by which inanimate objects are manipulated so they appear to move. If you hold a strip of movie film up to the light you will see that it's made up of separate "frames" of images that change slightly from frame to frame. When played through a projector they give the impression of movement because the film runs fast enough to fool the eye into seeing a continuous flow of images. Another factor that contributes to the relative quality of the animation is the degree of change from one frame to the next. If the two views are radically different, the animation appears "choppy." For example, if in one frame a man's hand is by his side and in the next frame he's holding it straight out, when the film is projected it will appear that he jerked his hand up in a sudden spasm.

MacAnimate lets you create "frames" of images with a Frame Editor that uses the familiar FatBits picture-editing functions. You can create up to 12 different frames (18 with a 512K Fat Mac). Using a Sequence Editor, you then specify the order in which they are to appear. Finally, you can see your frames animated and control the speed and direction of the animated image. It's easy to make changes in the frames or the sequence, and then run the new animation. Both your frames and your sequence can be saved to or loaded from the disk.

Those of you using version 2.1 of Microsoft BASIC on a 128K Mac will be disappointed to learn that MacAnimate is too large to run with the new version. This is because version 2.1 uses more of the Mac's memory than version 2.0, and the system of overlays presented here won't quite fit into a Slim Mac. The program will run nicely under version 2.0, though, and if you have a Fat Mac, you won't have a problem with either version.

The MacAnimate code features two important programming concepts: the manipulation of graphics stored in bit arrays and the use of overlays. Bit arrays allow graphics to be stored in a minimum of memory space, making it possible to have enough frames in memory to make animation feasible. The use of overlays allows more memory to be reserved for graphics while simultaneously permitting execution of the several sets of subroutines required to implement MacAnimate's many features. In effect, overlays allow your Mac to be as "fat" as you need it to be.

How to use MacAnimate

Let's go through the steps involved in creating an animated drawing. We'll demonstrate the process by creating and animating a helicopter with front and rear propellers.
After starting MacAnimate, you'll see the title screen shown in Figure 1. The MacAnimate menu controls all the steps in the animation process with these choices:

If you choose Instructions from the menu, you're presented with the window shown in Figure 2 on the next page, which provides a general description of the menu choices.

Choose an option from the MacAnimate Menu.

FIGURE 1. The MacAnimate title screen
The Frame Edit option is the first step in creating an animation. After choosing this option from the MacAnimate menu, you'll see a screen like that in Figure 3. As we mentioned, this screen uses the FatBits method for drawing pictures. The large rectangle at the left represents one “frame” of your animated sequence, and when you click the mouse pointer within the rectangle the “bit,” or tiny square representing one screen pixel under the pointer, will be switched to its opposite state. This means that if the little square is white, it will turn black, while if it is black, it will be erased (turn white). The grid shown in Figure 3 is optional, and appears when the Grid button is clicked.

You use the Frame Editor to create each frame you need for your animation. How many frames you need (up to the maximum for your machine size) depends on how smooth you want the animation to be and what kind of motion you want to produce. For example, if you just wanted to create the image of a jet plane and move it around the screen, you might only need one frame because the animation routine includes the ability to “steer” the image around the screen. This would hardly be animation,
though, since there would be no movement or change within the object itself. In the case of our helicopter, we wanted to show the two propellers spinning, so we used four frames, one for each propeller position that could be shown. If we had had a larger, more elaborate propeller in mind, we would have needed more frames to show the various positions. If you wanted to do a realistic animation of a little man performing a tap dance, you might need to use all the available frames to show the small changes in position involved and produce a smooth animation.

Some other things to note about the Frame Editor screen are the frame number, which is displayed in the upper right corner of the screen, and the actual-size image of the contents of the current screen, which is shown just to the left of the frame number. Finally, in the bottom right corner of the screen is a whole flock of buttons. Let’s look at what they do: We’ll go down the columns of buttons starting on the left.

*The Frame Editor buttons*

*Fwd* moves to the next frame, which you can then edit. If you are currently on the last available frame (frame #12 or #18, depending on your machine memory size), it will “wrap around” to frame #1.
*Back* moves to the previous frame and enables you to edit it. If you are on frame #1, it will wrap around to the last frame (frame #12 or #18).

*Cpy Fwd* ("copy forward") copies the contents of the current frame to the next frame. Again, it will wrap around if necessary. Any existing contents of the frame being copied to will be replaced. Having made the copy, you can then edit it. This enables you to make small changes to an otherwise identical image, for instance, if only an arm is moving by small increments from frame to frame.

*Grid* superimposes the grid shown in Figure 3, which enables you to see exactly where the pixels are placed.

*OK* returns you to the "default screen" shown in Figure 4. This screen shows you the frames currently in use and the default file name (the last file you've opened or saved to, if any). You can then make another selection from the MacAnimate menu.

*Up* shifts the entire image up one pixel. Note that any pixel that moves beyond the top boundary of the editing area will be lost; you won't be able to get it back by clicking the Down button.

![Image of MacAnimate Frame Display screen](image-url)

**FIGURE 4.** The MacAnimate Frame Display screen
*Down* works like *Up* except that it shifts the image one pixel down. Pixels that move below the bottom boundary of the editing area are lost.

*Left* shifts the image left one pixel. Pixels that move to the left of the left boundary are lost.

*Right* shifts the image right one pixel. Pixels that move to the right of the right boundary are lost.

*Cut* erases the image in the current frame and copies it to a pseudo-clipboard.

*Copy* copies the image in the current frame to the simulated clipboard but does not erase the image.

*Paste* copies an image stored on the “clipboard” to the current frame, replacing the frame’s contents.

*Invert* inverts (reverses) all the pixels in the edit area. White pixels become black, and black ones become white.

**Some editing tips**

If you haven’t done much FatBits-style drawing, you’ll find it takes a bit of practice to get the image you want. You can draw a straight line by dragging the mouse, but it requires a steady hand. You may find it easier, though more time-consuming, to click on the pixels one at a time. The Grid option is useful for making precise and regular changes from frame to frame so you can animate a part of the figure—our propellers are a good example of this technique.

You can use the shift options (Up, Down, Left, and Right) to shift frames progressively in a specified direction. You can then combine this frame-to-frame motion with the motion controls in the Animation procedure to get interesting effects, such as a planet that appears to revolve in a circle (through shifting), rotate (through changing the surface features from frame to frame), and travel through space (through motion commands in the Animation procedure)...all at once!

Once you’ve edited a sequence of frames and animated them (we’ll describe the animation procedure a bit later), you’ll probably find that you want to go back and edit the frames some more. If you’re editing a set of frames from a disk file and make a mistake (such as shifting the image part of the way out of the editing area), don’t worry. If you’ve already saved the sequence as a disk file, the contents of your disk file aren’t changed until you tell MacAnimate to again save your current frames to disk. Only saving to the current file name will change the contents of the file you last loaded.
Saving and reading sets of frames from disk

It can take a lot of work to create a series of frames for animation, so once you’ve completed one it’s a good idea to save it to a disk file. You do this by choosing Save As from the MacAnimate menu. (If you’re in the Frame Editor window, exit it first by clicking on the OK button.) After you’ve selected Save As, a standard file-save dialog box appears, and you can then specify the file name under which the animated sequence is to be saved.

A set of frames is read into memory by choosing Open from the MacAnimate menu. A standard file-opening dialog box appears, and you simply click on the name of the file to be opened (it must be a file created by MacAnimate). The file will be read into memory, and the full set of frames will be shown on the Frame Display screen. The file name selected is remembered; if you later choose the Save option, the frames in memory will automatically be saved to this file name. If a sequence file (discussed below) exists for the opened file, it will be read in also.

Setting an animation sequence

Animation requires more than a collection of images or frames. You also need to specify the order in which the frames will be shown. To do this, you must first have either created or read in from disk a set of frames. To set the sequence, choose Set Sequence from the MacAnimate menu. You will then see the screen shown in Figure 5.

The set of frames in use is shown at the top of the screen. At the bottom of the screen are 50 little boxes. Each box represents one frame, and the number that you place in the box represents the number of the frame you want to be shown at that point in the animation sequence. For example, we wanted our propeller animation to show the propellers revolving once each time frames #1 through #4 were displayed in order. If you look at the frames at the top of the screen, you can visualize this movement. We designed our frames so that the propellers move 45 degrees in each frame: Since both ends of a propeller move simultaneously, 180 degrees of motion of one blade gives the effect of the whole propeller moving 360 degrees. Because of the number of pixels used for the propellers and their relationship to the helicopter body, the propellers are most easily moved in steps of 45 degrees.

For our sequence we simply specified frames #1 through #4 to be shown in order. We didn’t have to specify frame #1 being shown again after frame #4, since the defined sequence is automatically and endlessly
Define an animation by entering a sequence of frame numbers in the edit boxes.

FIGURE 5. The Set Sequence screen

repeated during animation; you have to specify only one complete sequence. The order, of course, depends on how you want the image to appear to move, so it is important when defining sequences to make sure there is a smooth transition from the end of the sequence to its beginning.

The Sequence Editor uses standard Macintosh edit field functions (similar to those you would use to change the name of an icon, for example). The first box is shown in inverse (black on white), indicating that it is the frame position currently being edited. Clicking the mouse pointer in a box makes the box current, and a cursor appears in it. Numbers can be entered, cut, pasted, and so on using the usual Macintosh editing commands. If you want to clear out an existing sequence and start over again, click on the Clear button. Click on the OK button when you have finished entering the sequence.

The currently specified sequence will be saved to disk with the current collection of frames whenever you do a Save or Save As operation. The file name for the sequence will be the file name used in the Save operation, with the extension .Seq. When a file is opened, any existing sequence file is automatically loaded along with the frame file.
Running the animation

Once you have both a set of frames and a sequence, you're ready to see the animation in action. Choose Animate from the MacAnimate menu, and the screen in Figure 6 appears. You will then see your animation running near the center of the screen.

Note the instructions at the top of the screen. As they indicate, the F and S ("fast" and "slow") keys control the speed of the animation. The delay is initialized to 500 (rather slow), but it can be reduced as low as 1 by pressing the F key repeatedly. By using the S key, on the other hand, you can increase the delay to a large number so that you can see the animation in very slow motion and observe the movement closely from one frame to the next.

Note that the I, J, K, and M keys form a diamond on the keyboard. You can think of them as a kind of joystick that controls the speed and direction of the image. The I key moves the image toward the top of the screen; hitting the key repeatedly makes it move faster. The M key moves the image toward the bottom of the screen. If the image is already moving up the screen, hitting the M key first slows down its upward movement and...
then begins an increasing downward movement. The J and K keys work the same way to move the image left or right on the screen, respectively.

The X and Y speeds are shown in the bottom left corner of the screen. A negative Y speed is up; positive is down. A negative X speed is toward the left, and positive is toward the right. Since up/down and left/right speeds are independent of each other, you can be moving both up and to the right at different speeds, for example. The period key stops all directional motion of the image, but animation of the image itself continues.

Directional movement wraps around the screen: Going above the top of the screen resets the image to the bottom, and so on. When you're through running the animation, click the OK button in the lower right corner of the screen.

Now that you know all the steps involved, we encourage you (if encouragement is needed!) to create some frames and sequences and play with them. We suggest starting with simple images and sequences. Don't forget to save your work to disk if you want to keep it. When you want to exit MacAnimate (perhaps for food or sleep), you can select our usual Quit to BASIC or Quit to Desktop option from the MacAnimate menu.

**How MacAnimate works**

The program outline for MacAnimate is shown in Figure 7 on the next page. As you can see from it, and from the program listing itself at the end of the chapter, this is a very long program (although it really isn't too much more complicated than most of the others in the book). Consequently, our explanation of the program in the following sections will necessarily be a bit less thorough than those for the other programs in the book. But we've made sure to cover the most interesting and important aspects of the program. We also sometimes refer to other chapters where some of the techniques used in MacAnimate are discussed in depth.

This chapter also introduces a new aspect of BASIC programs, called *overlays*, which required that we somewhat modify our labeling scheme for the program outline and listing. The sections of the main MacAnimate program are labeled as usual, beginning with [Init 1], but you will discover that each of the four overlay sections of the program has a special sequence of keyed labels, prefixed with the overlay number. Thus, Overlay 0 is labeled from [OVR0-1] through [OVR0-4]; Overlay 1 is labeled from [OVR1-1] through [OVR1-15]; and so on.
### Initialization

Each time program is run, find number of current overlay

- if Overlay 0 not present, merge and run it (BringIn)

### Main Program

Run endless loop, interrupted by menu events

### Menu Events

- **MenuEvent**: branch to appropriate subroutine
- After menu subroutine processed, redraw frames (AfterMenu)

- **FIO**: load or save file
  - if Overlay 2 not present, merge and run it (BringIn)

- **Animate**: run animation
  - Open second window (OpenWindow)
  - Show instruction text
  - Create OK button in corner
  - If no sequence loaded, display error message
  - Initialize animation variables
  - Display animation settings (ShowSettings)
  - **WHILE** not quitting (OK button clicked)
    - Find current location and frame number of image
    - If image not moving, draw with PSET
    - If image is moving, draw with XOR
    - If animation delay or direction keys pressed, change delay or direction variable
    - Pause for length of delay before redrawing
    - Change location of image
    - Modify location if window boundary crossed
    - Increment sequence number
    - Draw next image at new location
  - **WEND**
  - Close second window

### Sequence

- Set animation sequence
  - Open second window (OpenWindow)
  - Create sequence edit fields (EditFields)
  - Display instruction text
  - Show frame set at reduced size (ShowFrames)
  - Create OK and Clear buttons
  - **WHILE** not quitting (OK button clicked)
    - If Clear button clicked, clear all edit fields (EditFields)
    - If tab key pressed, move to next edit field
  - **WEND**
  - Save contents of edit fields in sequence array
  - Close second window

### EdFrame

- Edit individual frame
  - If Overlay 1 not present, merge and run it (BringIn)

(continued)
MAC ANIMATE

ClearAll: clear all frames
  clear screen
  reset current filename
  save blank image into each element of frame array

Inst: display instructions
  if Overlay 3 not present, merge and run it (BringIn)

BQuit: quit to BASIC

DQuit: quit to Mac Desktop

Subroutines
AftrMenu:
  show all current frames (ShowFrames)
  enable menu selection

ShowSettings: print animation settings

EditFields: display grid of edit fields

ShowFrames: display grid of frame collection

OpenWindow: open second window for frame operations

BringIn: merge and run specified overlay

Overlays
Overlay 0: Initialize program (MacAnimate.Ovr0)
  determine whether running on Slim or Fat Mac
  dimension arrays for frame bitmaps, frame sequence
  initialize arrays, cursor pattern and grid pattern
  define constants for window size, animation delay, etc.
  set main output window
  display title screen
  install and enable MacAnimate menu

Overlay 1: edit frames (MacAnimate.Ovr1)
  open second window (OpenWindow)
  set constants and dimensions of edit area
  define function to determine if mouse pointer in edit area
  create buttons for editing
  display instruction text
  WHILE not quitting (OK button clicked)
    show current frame number and contents
    draw background grid if selected
  WHILE not a new frame and not quitting
    get mouse pointer location
    if pointer in edit area, change to pencil cursor
    if mouse button is down while pointer in edit area
      calculate corresponding bitmap pixel
      reverse color of pixel

Figure 7. Program outline (continued)
WHILE button still down (AffectBits)
    get cursor location
    keep drawing in same color
WEND
    save new state of frame in frame array
    if mouse button is down while pointer not in edit area
        check if editing button clicked
        branch to appropriate subroutine (see below)
WEND
WEND

Overlay 1: Subroutines

Fwd: increment frame number
Shift: shift image up, down, right or left
Cut: cut image to "clipboard" array (ToClip)
Back: decrement frame number
Copy: copy image to "clipboard" array (ToClip)
CopyFwd: copy image forward to next frame (Fwd)
Paste: copy image from "clipboard" array (FromClip)
Grid: set background grid for edit area if selected
Invert: reverse all bits in edit area, resave image to array
OKBtn: set flag to leave WHILE...WEND loop

ToClip: copy image to "clipboard" array
FromClip: copy image from "clipboard" array
AffectBits: change bits in image while button still down

Overlay 2: file input/output (MacAnimate.Ovr2)
on menu event, branch to appropriate subroutine:

OpenFileOvr: get frame and sequence files from disk
    get filename from user
    if file not found, return
    load frame file into bitmap array
    if sequence file not found, return (FileError)
    load sequence file into array

SaveFileOvr: save frame and sequence arrays to disk
    use current filename
    save frame set array
    save sequence array

SaveAsOvr: save as: save frame and sequence arrays to disk
    get filename from user
    if no name entered, return
    save files (SaveFileOvr)

Overlay 3: display instructions (MacAnimate.Ovr3)
open second window
display instruction text
create OK button for quitting overlay
wait for button click to end instructions

Figure 7. Program outline (continued)
Program overview

The key to the structure of MacAnimate is its use of overlays. When the program is first run, the code listed under Main Program is loaded into memory. This code includes (among other things) the usual "endless" main program loop that waits for menu selections, the main menu event-handler, a secondary handler for file I/O (input/output) routines, and finally the subroutines that run the actual animation.

The MacAnimate program has to handle many operations, including initialization, disk-file operations, frame editing, and showing instructions. If all this code had to be kept in memory all the time, there wouldn't be much memory left for the array of bit maps that holds the frames to be animated. In fact, when writing this program we ran into a real problem trying to provide all of MacAnimate's nifty features and still leave memory for enough frames for animation: The problem is especially acute on a 128K Slim Mac.

The solution was to break the code into blocks of logically related operations and then load in only the segment needed for the operation being performed. Thus, when frame editing (for example) is called for, the subroutines involved in frame editing are loaded and added to the program in memory. If the user then decides to perform disk operations, the code for frame editing is replaced or "overlaid" with the code for disk I/O.

This process is actually implemented by ending the main program (the part that always remains in memory) with the labels 5000 and 5999 [SR 6]. These labels define the overlay area—the "slot" into which overlays are inserted. When the program is first run, Overlay 0, containing the initialization subroutines, is brought in. Later, when the user makes a menu selection, the appropriate subroutine checks to see whether the overlay containing the subroutines needed to perform the requested function is currently in memory. The variable overlay contains the number of the overlay currently in use (there are four overlays, named MacAnimate.OvrN, where N is the overlay number from 0 through 3). Each overlay contains a DATA statement with the overlay's identifying number, so the program can READ it in order to determine which overlay is in memory when the program is run. (This is necessary because the program may have been stopped and then run again from BASIC, for example.)

Overlays are "brought in" (loaded and added to the main program) by Bringln [SR 6]. The variable newOvr specifies the number of the overlay to be brought in, and the current overlay number in overlay is also set to this overlay's number. The CHAIN statement actually brings the overlay into the program.
The reserved word CHAIN indicates that a program will be brought in from disk. MERGE specifies that the program being brought in will merge with, rather than replace, the existing program (MacAnimate itself). The variable $ovr$ contains the name of the overlay program, and 5000 is the line number to which control will be transferred (the Microsoft BASIC manual emphasizes that you have to use a number, not an alphanumeric label, for this purpose). All overlays in MacAnimate begin with 5000 and end with 5999: Between them, regular alphanumeric labels can be used (and are in MacAnimate).

The ALL specifies that all variables in the main program (except variables local to subroutines, if any) will be available to the overlay as well. The \texttt{DELETE 5000-5999} tells BASIC to delete any existing code between the numbers 5000 and 5999 before bringing in the overlay (that is, to delete the current overlay before bringing in the new one). The CHAIN statement can be used with other options: for example, to transfer control to another program without overlaying.

### Initialization

The \texttt{FileName$} function, defined at the start of \texttt{[Init]}, finds the colon that separates the volume (disk) name from the name of the file itself, then extracts the latter. This avoids having to display very long file names when the volume part is unnecessary.

We've already given a general description of the way overlays work. The initialization overlay, known by the file name \texttt{MacAnimate.Ovr0 [OVR0-1]}, is brought in by \texttt{BringIn [SR 6]}. When the program is run, the number of the overlay currently loaded into memory is determined. This is done by using \texttt{READ} to get the number specified in the DATA statement at the beginning of the current overlay starting at the label \texttt{OverlayData}.

We have used a "dummy" overlay number at the end of the main program to force the program to merge the first, initialization overlay. The way you should enter these programs is to type in the main program and save it under the name \texttt{MacAnimate}. Then type the overlays in separately, saving each under the name \texttt{MacAnimate.OvrN}, where \texttt{N} is the number of the overlay. By entering the program in this way, it will be easy to edit and debug each module independently, without affecting any of the others.

At the beginning of [OVR0-1] is a sequence of code common to all the overlays. The code sequence begins with a \texttt{GOSUB} to whatever subroutine is needed to continue appropriate processing. Here, it's Initialize
(in other overlays, control may be switched to subroutines to process other program functions, such as the current menu selection). After the subroutines in the overlay are processed and control returns to the start of the overlay, the ON MENU GOSUB and MENU ON statements must be repeated. This is due to the fact that the CHAIN MERGE statement used to bring in overlays disables all event trapping; so we have to make sure that event trapping is turned on before jumping back (via a GOTO) to the endless loop in the main program.

In those overlays that are called in response to menu events (that is, all but the initialization overlay), AftrMenu [SR 1] is called just before the program returns to the main loop. This subroutine clears the screen and shows the “default screen” using ShowFrames [SR 4] (Figure 4), which is always shown when a menu event has been processed. If a default file isn’t present (file$ = ""), the user is reminded to choose an option from the MacAnimate menu. If a file is present, its name is shown (using the FileName$ function to remove the unnecessary volume designation). At the end of AftrMenu, the MacAnimate menu is re-enabled.

If an overlay is already in memory when it is needed, an ordinary GOSUB in that menu-event subroutine in the main program is used to jump to the appropriate subroutine in the overlay. There is no need in this case to execute the statements at the beginning of the overlay, since event trapping and the menu don’t have to be re-enabled. Upon return from that subroutine, control will return to the endless main loop in the normal way from the menu-event subroutine without a GOTO being necessary.

Now let’s look at what the initialization overlay does, beginning with [OVR0-2]. The number of sequence positions maxSeq, and the parameters for displaying each frame are given first. Next, the memory capacity of the machine running the program is determined. The BASIC function FRE(0) returns the number of free (unused) bytes in BASIC’s memory space. A number larger than 30000 could be used, but we know that BASIC will never have as many as 30,000 free bytes on a 128K Mac (there are less than 21,000 free bytes even when no program is loaded). You might find this technique useful if you write a program that can offer additional features or capacity on larger machines but that will also run as a “bare bones” version on Slim Macs. Here, maxFrames, the number of frames that can be used in an animation set, is set to 18 if the Mac is fat and 12 if it is not.

Now the three main arrays are dimensioned. Each element of the frame array is a bit-mapped image. The size of one element in bytes is determined using a simplified form of the formula given in the Microsoft
BASIC documentation under the GET statement. The number of integer elements (remember, an integer is two bytes or 16 bits) in one frame works out to 327, or 654 bytes. (You can see why we must use whatever tricks we can find to maximize the amount of memory available for frames.) The frame array thus has maxFrames elements of arraySize integers.

Clip, the array that holds one frame copied from the Frame Editor, has one element of arraySize. The array seq has maxSeq integer elements, each holding one sequence position number. A little further on, seq is initialized by a FOR...NEXT loop to all negative 1s. This is done so we can abort an animation if there is no current sequence (sequence(1) = -1), and also so the Sequence Editor will show all the edit fields as blank.

The pencil cursor is next read into an array called pencil. The cursor was created with MacCursor, another utility found in this book, using data statements generated by MacCursor and transferred via the Clipboard. The DATA statements appear in [OVR0-4]. If you want a different cursor (a quill pen, perhaps?), you can easily create it with MacCursor and substitute its data statements for ours. The grid pattern (gridPat), used as an option with the Frame Editor, is then initialized.

Next, various constants used by the program are set. The patterns patBic and patOr stand for the "black is changed" and "OR" modes. These will be used in the Frame Editor to draw a grid as selected. The boundaries of the secondary window (which is used for Frame Edit, Sequence Edit, and Animate), as well as the size of a FatBit in pixels are specified, as are the current frame number (1) and delay (500).

Clear All [ME 5] is called to initialize the bit maps for the frames in the frame array to empty. The screen is cleared, and the current frame file name is set to null (that is, there is no current frame file). This is done with a FOR...NEXT loop in ClearAll that simply GETs the image in the upper right corner of the screen corresponding in dimensions to the frame edit area. Since the screen is blank, the image stored in the bit map in the array is also blank. When the loop is finished, all of the frame array elements represent blank frames.

Next, in [OVR0-3] the primary window is set up and the title screen is drawn. Its operation is very straightforward.

The last step of the overall initialization sequence is creating the MacAnimate menu. Note that when the menu is first created, the Save option is dimmed, since there can be no current file to save when the program first starts up. When the Save As option is used, thus establishing a current frame file, the Save option will be enabled.
Control now returns to the beginning of Overlay 0, where Initialize was called so very long ago. Then, as mentioned before, menu event-trapping is enabled, and a GOTO sends us to the endless loop in the main program [Main] to wait for a menu selection.

We’ll now describe the operations involved in handling the menu items, taking them in order. Some of the options are simple enough that you can easily figure out how they work with the aid of the program outline. We’ll reserve detailed explanations for the more complex ones.

File I/O operations

If MenuEvent [ME 0] determines that one of the three file I/O operations (Open, Save, Save As) was selected, control goes to the file-handler subroutine FIO [ME 1]. This subroutine calls BringIn [SR 6] to bring in Overlay 2 (MacAnimate.Ovr2) if it’s not already loaded. The options are then processed [OVR2-2] using Overlay 2’s subroutines.

If Open is chosen, a new frame file is opened by the subroutine OpenFileOvr [OVR2-2]. The BASIC FILES$ function is used to get a file name from the user: The file must be a TEXT type, so only files of that type are shown. If FILES$ returns a null file name, then no file was selected (the Cancel button was pushed instead), and OpenFileOvr returns. Otherwise, the variable pict, which counts the number of frames read in, is set to 0.

A WHILE...WEND loop reads in the frames. In addition to testing for EOF, it tests for more than maxFrames being read in. If maxFrames is exceeded, any further images in the file aren’t read in—to do so would cause the index to the frame array to go out of bounds. If maxFrames isn’t exceeded, the frame read with INPUT$ is displayed on the screen by a PICTURE statement, and the image is then stored from the screen into the current element of the frame array. Finally, the current file name is set to the name just used, and the Save menu option, which is now relevant, is enabled as mentioned earlier.

The sequence file is then read in. The extension .Seq is added to the file name, and the program attempts to open this file. If an error occurs (probably due to the file not being present on the disk), FileError [OVR2-5] is called. This subroutine halts the program completely (with GOTO 0) if the file error is something other than “File not Found” — perhaps an unresolved program bug. Otherwise, it simply sets the flag fileHere to false and OpenFileOvr RETURNS; otherwise the sequence file is read in.

The Save option is handled by SaveFileOvr [OVR2-3]. First, a FOR...NEXT loop clears the screen, then saves each frame by turning
on picture recording (PICTURE ON), displaying the corresponding element of the frame array, getting the graphics in string form from PICTURE$, and printing the length of the string in bytes and then the string itself to the file. You can check the OpenFileOvr subroutine again to verify that pictures are read back as a byte count (that is, string length) followed by the picture string itself. The file name for the sequence file is then created by adding the extension .Seq, and a second FOR...NEXT loop is used to save the sequence file.

Save As works the same way and also uses SaveFileOvr. However, since a file name is needed, it is first obtained from the user by a call to the SaveAsOvr [OVR2-4], which uses the FILE$ function to get a file name. Since Save As, like Open, establishes a current file name, the Save menu option is enabled.

Animation

The Animate subroutine [ME 2] handles the animation and sequence of a frame set. A secondary window is first opened by a call to OpenWindow [SR 5]. Animation doesn’t require an overlay, so all necessary routines are in the main program.

First, the starting (x,y) coordinates for the instruction text are set, and the instructions telling the user which keys to use are displayed. The OK button is set up relative to the lower right corner of the screen. Then the first element of the seq array is checked; if it’s a -1, there is no current animation sequence. The user is then told to define an animation sequence before using this option, and the subroutine returns.

If there is an animation sequence, the current frame number is set to 1, and the working (x,y) coordinates for the animation in (aX,aY) are initialized. The speeds xSpd and ySpd are set to 0. ShowSettings [SR 2] is called to show the current value of the delay factor, and the current x and y speeds of the sequence.

A WHILE...WEND loop beginning with WHILE NOT aQuit is used to do the animation until the user clicks OK to quit (setting the flag aQuit to true). In the next line, the variables x and y are set to the working values (aX,aY), and the variable c is set to the current frame number (crnt).

To understand the next three statements, you must note that at the end of the loop a new set of coordinates for the image is calculated based on its x and y speeds. The result is, in effect, a “new” image: the next one in the sequence (this is true every time except the first time the loop is executed, of course). If both the x and y speeds are zero (IF NOT moving), the image hasn’t moved since the last time it was checked. So the current
frame (that is, the one in the sequence after the last one drawn) is drawn with a PUT statement using the PSET option. There is no need to worry about the background, since the image hasn’t moved, and the new image simply replaces the old. If either the x or y speed isn’t zero (IF moving), the image has moved since last time through the loop. We therefore want to preserve the background we are moving over, so we draw the image by PUTting it with the XOR option.

DIALOG(0) is then checked to see if the user wants to quit. If so, the aQuit flag is set to true so the loop will exit at the end of this pass. We next check for a key press, first converting any key in the INKEY$ buffer to uppercase so the user doesn’t have to worry about case. The keys can be considered in two groups: speed keys and direction keys.

For the F and S speed keys, the delay is decreased and increased respectively and then the new delay is shown. For the F (faster means less delay) option, by multiplying the current delay by a reduction factor, we create an adjustment that starts out with large increments but becomes finer and finer until it only changes the delay by 1. On the other hand, the delay can increase indefinitely via the S-key option and will do so in larger and larger increments. (Eventually, of course, you’ll overflow the integer variable, but unless you’re doing something like an animated study of real-time fungus growth, there is little point in having really large delays. If you just want to closely study individual frames, the Frame Editor works much better.) For both the F and S keys, the new delay value is shown by calling ShowSettings at the end of the section.

The direction keys (J, K, I, M) simply increase or decrease the appropriate speed by one pixel per key press. The period key stops directional movement by setting both speeds to zero. The new x and y speeds are shown via a call to ShowSettings in all cases.

The WHILE INKEY <> "" : WEND loop following the key processing is a useful trick to remember. Impatient users have been known to hit a key a dozen times or so before they’re satisfied with what is happening, but this loop clears out the keyboard buffer so only one key is processed at a time. The user gets a chance to see what happened — and only then can press another key with effect.

A delay loop next pauses for the length of the specified delay. The x and y speeds are then added to the working variables aX and aY, and we check to see if the new coordinates would place the image outside a window boundary. If you visualize the coordinates in terms of the window width and height, you’ll see that if a boundary is exceeded, the image’s coordinates are changed to put it just one frame width inside the opposite
boundary. Next, the number of the frame to be shown is incremented. The MOD function is used to wrap the frame after maxSeq to frame 1. If there are less than maxSeq sequence numbers, the number following the last valid one will have been left as -1 by the Sequence Editor. Therefore, if a -1 is encountered, the current sequence number is reset to 1.

Finally, if either or both of the x and y speeds are not 0, the image is drifting across the screen. The effect is created by erasing the current image by PUTting it at its location with the XOR option, then returning to the top of the loop to draw the new image at the new position.

When the user clicks the OK button, control drops through the bottom of the WHILE...WEND loop, the secondary window is closed, and control returns to the main program loop.

The Sequence Editor

The Set Sequence menu option is handled in [ME 3]. The secondary window is first set by calling OpenWindow. Then the top and left boundaries of the edit field area are set. EditFields [SR 3] is called to open the sequence edit fields. In this subroutine, the flag seqEnd is set to false, and the number of columns is set to INT (maxSeq / 5). Note that the extra effort expended here and throughout the routine to make the layout more flexible means that the routine can be used to set up a smaller or larger number of edit fields if desired.

The FOR...NEXT loop in EditFields first finds the left and top coordinates for each edit-field box. The left coordinate is the left boundary of the edit area (leftEd) plus the column number times the width of one edit field (40). As is usual with offset-oriented operations, the column number must start with 0 for this to work, so we subtract 1 from the value of the loop index which is the same as the number of the edit field. Using MOD with the index gives us a column number that wraps around to 0 each time the number of columns (col) is exceeded.

The top of the edit-field box is determined in a similar fashion, except that we use INT to find the integer result of dividing by the column number. This means that we add the height of one row each time we use up one set of columns, thus putting each edit field in the correct row. The final coordinate is obtained by multiplying by 20, the height of an edit field.

Since we want to print a guide number to the left of each row of edit fields, (n - 1) MOD col is again checked to see if we are at the start of a new column. If we are, the loop index is printed as a guide number. The value of the element in the seq array corresponding to the current edit
field is then checked, and the flag seqEnd is set to true if a -1 is found. The string to be printed in the edit field is null (nothing will be printed) when the value is -1; otherwise, the value is converted to a string for printing.

The EDIT FIELD statement opens each edit field. The edit field is given the loop index n as an identifier, the string converted from the seq array value as the string to be shown and edited, a rectangle defined from the coordinates (l,t) we just calculated, and options indicating that a return or tab key will move the user to the next field and that text will be center justified. The current edit field is set to 1. (A more detailed discussion of edit fields can be found in the chapter on MacGraph.)

After the edit fields have been established, control returns to Sequence [ME 3]. The coordinates for the start of the instruction area are set, and the instruction lines are printed relative to them. ShowFrames [SR 4] is then called to display the current set of frames at the top of the Sequence Editor screen. This subroutine is general purpose: It is also used to show the frames on the “default” (waiting for menu events) screen. The variable reduction controls the size of the frame images: The width and height in pixels are both divided by the value of the reduction. After this calculation is done and the width and height of the frame are reduced accordingly, the left (showL) and top (showT) boundaries of the area in which the frames will be shown are set.

The FOR...NEXT loop in ShowFrames displays the total of maxFrames frames. The width plus 15 (to allow for the space between frames) is multiplied by the current column position. The height plus 20 is multiplied by the current row. (We need a larger gap between rows so the frame numbers can be printed there.) The column and row numbers are calculated using the same method that was used for spacing the edit fields, except that the total number of columns here is in the constant fPerLine. A LINE statement with the “box” option is used to draw each empty frame, and then the corresponding image in the frame array is PUT into the box. The pen is then positioned below the frame, and the number is printed as a string, right-justified by the use of the RIGHT$ and LEN functions.

After the frames have been shown and control has returned to Sequence, the OK and Clear buttons are established. OK is set to false, and a WHILE NOT OK...WEND loop is used to actually edit the fields. The value of DIALOG(0), assigned to diag, is checked first to see whether a button was pushed. If it was, DIALOG(1) is checked to see which button was pushed. If DIALOG(1) = 1, the OK button was pushed, so the value of OK is set to true. If DIALOG(1) = 2, the Clear button was pushed, so the first
element of seq is set to −1 and EditFields is called. We’ve already seen that under these conditions, EditFields will show all the edit fields and their contents as empty.

If diag = 2, the mouse has been moved into a new edit field and clicked, and the identifier of the new field is returned by DIALOG(2)—it is defined this way by the BASIC field-editing routines. In this case, the current edit field is set to the new edit field. The values 6 and 7 in DIALOG(2) indicate that the Return (6) or Tab (7) key was used to move to the next edit field. If so, the current edit field is incremented by one (or wrapped around to 1 if necessary).

Once the user’s editing is finished (the OK button is clicked), a FOR...NEXT loop gets up to maxSeq values from the edit fields (the values are accessed by the special function EDIT$, defined by BASIC). If the value is negative or greater than maxFrames, it is set to −1 and the sequence ends at that point. These values are then stored in the corresponding elements of the seg array. Closing the secondary window completes the Set Sequence option.

The Frame Editor

The EdFrame subroutine found in [ME 4] first calls in Overlay 1 (MacAnimate.OVR1) if it’s not in memory. The EditOvr subroutine [OVR1-2], which has the dubious distinction of being one of the longest to be found in our programs, starts by opening the secondary window via a call to OpenWindow. The standard rectangular coordinates for the frame edit (FatBits) area are defined. (The 2 has to be subtracted in each equation in order to align the results of the InBits function with the actual pattern of FatBits in the edit area.

The coordinates for the borderless frame that will display the actual-size image are offset from the upper right corner of the edit area. The function InBits at [OVR1-2a] is then defined to check whether a given pair of coordinates is within the edit area. A FOR...NEXT loop reads in the names of the 13 buttons from DATA statements and sets them up. The instruction text is then displayed.

We now set OK to false as usual, then initialize gridMode to the BIC (black is changed to white) mode. This background is in effect invisible against the white screen, except that the crisscross lines change the black in the image to white and thus show the black pixels outlined in white. You can see the effect of this when using any of the image-shifting buttons: All the pixels are momentarily bigger, then shrink as the white
grid is written over them. The pattern is toggled back and forth between this one and OR mode (patOr), which draws the black-lined grid, by clicking the Grid button.

The WHILE NOT OK...WEND loop at [OVR1-2b] edits frames until the user clicks the OK button. First, the current frame number is printed in the upper right corner of the screen. Then the image in the current element of the frame array is PUT in the edit area. The pen mode is set to the current value of gridMode, and the grid pattern is set. The PAINTRECT statement fills the edit area with the current pattern, after which the pen is set back to normal. We will now let the user edit this frame until he or she either quits or activates a new frame for editing.

The WHILE NOT (newFrame OR OK)...WEND loop at [OVR1-2c] first checks the mouse status and location. If the InBits function indicates that the mouse isn’t in the “bits” (edit) area, the cursor is set back to normal and a button press is checked for and processed if found.

If the mouse pointer is in the edit area, it is then checked to see whether the button is being held down. If it is, the grid coordinates of the FatBit the mouse is on are calculated. Next, the image currently being displayed to the upper right of the edit box is checked: The pixel in that image corresponding to the FatBit being clicked in the edit area is found by offsetting the FatBit coordinates from the edge of the “invisible” frame around the image, whose upper left corner is (sX1,sY1). This is a clever trick that lets us use the BASIC POINT function to obtain the color of this pixel so we can change it to its opposite. (The approach to this problem used in MacCursor is to keep a separate bit map and check the coordinates against it. Our approach here is equally arcane but has the advantage of not requiring the memory for maxFrames additional bit maps, which we don’t have to spare!)

Once we’ve set the first pixel, AffectBits [OVR1-15] is called to change the color of this and all other pixels encountered as long as the mouse button is down in the edit area. The WHILE...WEND loop in AffectBits gets the screen coordinates of the mouse location and uses a PSET statement to offset into the real-size image and turn on the corresponding bit. The coordinates of the FatBit are then calculated, and it is drawn with a LINE statement with a box-fill in the required color. When editing on each frame is completed, the new frame is saved to the frame array. When the mouse button is no longer down, the new mouse status and location are obtained and the subroutine returns to EditOvr [OVR1-2].
The program outline gives a thorough description of how button options [OVR1-3] through [OVR1-14] are processed, so we'll leave it to you to compare the outline and the actual code and figure them out. Note that some buttons can affect the frame currently being edited. If this happens, control goes back up to the main WHILE...WEND loop in EditOvr and the new frame is set up. When editing is completed, the overlay concludes as described previously for other overlays.

Remaining menu options

The Clear All option calls ClearAll [ME 5], which we've already discussed as part of the program initialization. The Instructions [ME 6], Quit to BASIC [ME 7], and Quit to Desktop [ME 8] options are all standard for our programs, except that the instructions are in Overlay 3. This overlay is handled in the same way as the others, with the actual instructions appearing via InstOvr [OVR3-2].

Suggestions for MacAnimate

If you've followed us through this chapter, you'll no doubt agree that some of it was rough going, and there was certainly a lot of material presented. When you understand this program, you can definitely consider yourself a Macintosh applications expert!

When you feel you've gained a thorough understanding of how this program works there are a number of improvements that you might want to make. Before doing so, remember that adding significantly more code will either reduce the number of frames the program can handle, or will require the addition of new overlays and code to manage them.

You can create frame-editing tools other than a pencil (perhaps a brush?) using the MacCursor program found in this book. After creating the cursor, paste the DATA statements generated by MacCursor in place of [OVR0-4], or include both and provide buttons so the user can choose which of several different tools are to be used.

As in the case of our helicopter, many images you might work with will be considerably smaller than the grid size of 64 x 64 that we've used in MacAnimate. You could try changing the grid size to 16 x 16, which would allow many more frames to be used (though not sixteen
times as many, due to overhead). Alternatively, you can create your images only in the upper left quarter of the grid, and have programs read in only that much of the encoded “picture string.”

A simple but helpful little change is to have MacAnimate create all of its files with the GRPH file type (simply by using this as an extension on the file name in a NAME...AS statement). This will mean that only those files will appear in the file dialog boxes, making it easier to find and use these files on your disk.

Two other options to consider are using animated sequences created in MacAnimate in other BASIC programs, and creating images in MacPaint and loading them into MacAnimate via MacInterface, another program in this book. Let’s look at each of these procedures individually.

**Using MacAnimate sequences with other BASIC programs**

The general procedure for using a set of MacAnimate frames in a BASIC program is to dimension an array to hold them, read them in from the MacAnimate file, and show them in the desired sequence. The following is a more detailed step-by-step description.

1. Dimension your image array as \((arraySize, maxFrames)\). Calculate \(arraySize\) from the image grid dimensions using the formula given in [OVR0-2]. \(maxFrames\) is the total number of frames your program will use. Dimension an array to hold the sequence file if it will be used. You could also insert an option into MacAnimate itself to calculate and display the dimensions needed for a given number of frames.

2. The code in [OVR2-2] can be used with little modification as a stand-alone subroutine to read in a frame and a sequence file. Change the variable names to conform to your own, and remove references to subroutines you will not be using.

3. The code in the MacAnimate routines (starting with [ME 2]) should give you the basic ideas on how to animate by drawing, erasing, and moving objects. (A number of other programs in this book—nearly all the games, for example—contain other examples of animation.) If you want to use the sequence file to control the animation, simply read it into an array and step the array index through the array to set each frame number in succession. You can use a counter with MOD (as we do in [ME 2] near the end of the main loop) to automatically restart the sequence when the last frame is reached.
Using MacAnimate with MacInterface

If you have read the chapter on MacInterface, you know how that program uses the Clipboard to move images back and forth between MacPaint (and other applications) and disk files accessible to BASIC programs. Since MacInterface reads images from and copies them to the Clipboard, you can, for example, create an image using MacPaint, copy it to the Clipboard, exit MacPaint, run MacInterface to read the Clipboard and create a disk file from the image or images, and then run MacAnimate and load the file for editing and animation.
MacAnimate program listing

MacAnimate
A frame animation program with overlays.

'returns filename from 'Volume:Filename' string
DEF FN FileName$(f$) = RIGHT$(f$, LEN(f$) - INSTR(f$, ":"))

RESTORE OverlayData
READ overlay 'find number of current overlay
IF overlay = 0 THEN GOSUB Initialize
IF overlay <> 0 THEN newOvr = 0 : GOTO BringIn 'if not init, reset overlay

ON MENU GOSUB MenuEvent : MENU ON

MainLoop:
WHILE true : WEND 'loop until menu event

MenuEvent: 'all menu events come through here
MENU 6, 0, 0 'disable menu
item = MENU(1)
ON item GOSUB FIO,FIO,FIO,Animate,Sequence,EdFrame,ClearAll,Inst,,BQuit,DQuit
GOSUB AfterMenu
RETURN

FIO: 'all file I/O comes here
IF overlay = 2 THEN ON item GOSUB OpenFileOvr, SaveFileOvr, SaveAsOvr
IF overlay <> 2 THEN newOvr = 2 : GOTO BringIn
RETURN

Animate: animate frames
GOSUB OpenWindow
x = 120 : y = 20
CALL MOVETO(x,y) : y = y + 16
PRINT "Change animation delay with F and S keys."
CALL MOVETO(x,y) : y = y + 16
PRINT "Change direction of motion with I, J, K, and M keys."
CALL MOVETO(x,y) : y = y + 16
PRINT "The '.' key stops any motion."
BUTTON 1, 1, "OK", (wind2W - 60, wind2H - 30) - (wind2W - 10, wind2H - 10)
aQuit = (seq(1) = -1)

IF NOT aQuit THEN NotAQuit
CALL MOVETO(x - 15, y + 32) : TEXTFONT(0)
PRINT "You must define an animation sequence first!"
CALL TEXTFONT(1)
WHILE DIALOG(0) <> 1 : WEND
NotAQuit:
crnt = 1: aX = 150: aY = 100
xSpd = 0: ySpd = 0
GOSUB ShowSettings
WHILE NOT aQuit
x = aX : y = aY : c = crnt
moving = (xSpd <> 0 OR ySpd <> 0)
IF moving THEN PUT (x,y), frame(1,seq(c)), XOR
IF NOT moving THEN PUT (x,y), frame(1,seq(c)), PSET
IF DIALOG(0) = 1 THEN aQuit = true
key$ = UCASE$(INKEY$)
IF key$ = " " THEN NoKey
IF key$ = "F" THEN delay = INT(.9 * delay)
IF key$ = "S" THEN delay = 1.1 * delay + 1
IF key$ = "J" THEN xSpd = xSpd - 1
IF key$ = "K" THEN xSpd = xSpd + 1
IF key$ = "I" THEN ySpd = ySpd - 1
IF key$ = "M" THEN ySpd = ySpd + 1
IF key$ = "." THEN ySpd = 0 : xSpd = 0
GOSUB ShowSettings
WHILE INKEY$ <> "": WEND 'empty keyboard buffer
NoKey:
FOR t = 0 TO delay : NEXT
aX = aX + xSpd : aY = aY + ySpd
IF aX < 0 THEN aX = wind2W - bitsW
IF aX > wind2W - bitsW THEN aX = 0
IF aY < 0 THEN aY = wind2H - bitsH
IF aY > wind2H - bitsH THEN aY = 0
crnt = (crnt MOD maxSeq) + 1 : IF seq(crnt) < 1 THEN crnt = 1
IF xSpd <> 0 OR ySpd <> 0 THEN PUT (x,y), frame(1,seq(c)), XOR
WEND
WINDOW CLOSE 2
RETURN

Sequence: set animation sequence
GOSUB OpenWindow
topEd = 170 : leftEd = 20
GOSUB EditFields
x = 325 : y = 20
CALL MOVETO(x,y) : y = y + 16 : PRINT "Define an animation by"
CALL MOVETO(x,y) : y = y + 16 : PRINT "entering a sequence of"
CALL MOVETO(x,y) : y = y + 16 : PRINT "frame numbers in the"
CALL MOVETO(x,y) : y = y + 16 : PRINT "edit boxes."
reduction = 2
GOSUB ShowFrames
BUTTON 1, 1, "OK", (wind2W - 55, wind2H - 25) - (wind2W - 5, wind2H - 5)
BUTTON 2, 1, "Clear", (wind2W - 55, wind2H - 50) - (wind2W - 5, wind2H - 30)
OK = false
WHILE NOT OK	diag = DIALOG(0)
IF diag <> 1 THEN NotSBtn
OK = (DIALOG(1) = 1)
IF DIALOG(1) = 2 THEN seq(1) = -1 : GOSUB EditFields
NotSBtn:
IF diag = 2 THEN eField = DIALOG(2)
IF diag <> 7 AND diag <> 6 THEN NotTab
eField = eField MOD maxSeq + 1
EDIT FIELD eField
NotTab:
WEND
FOR n = 1 TO maxSeq
    j = VAL(EDIT$ (n))
    IF j < 1 OR j > maxFrames OR INT (VAL (EDIT$ (n))) <> j THEN j = -1
    seq(n) = j
NEXT n
WINDOW CLOSE 2
RETURN
EdFrame:
    'save new sequences
    IF overlay = 1 THEN GOSUB EditOvr
    IF overlay <> 1 THEN newOvr = 1 : GOTO Bringln
RETURN
ClearAll:
    'clear all frames
    CLS : file$ = ""
    IF overlay <> 0 THEN MENU 6, 2, 0
    FOR n = 1 TO maxFrames
        GET (0, 0) - (bitsW - 1, bitsH - 1), frame(1, n)
    NEXT n
RETURN
Inst:
    'show instructions
    IF overlay = 3 THEN GOSUB InstOvr
    IF overlay <> 3 THEN newOvr = 3 : GOTO Bringln
RETURN
BQuit:
    'quit to BASIC
    MENU RESET
    END
DQuit:
    'return to Mac Desktop
    MENU RESET
    SYSTEM
**SUBROUTINES**

AfterMenu:

- ___________ after any menu event; also called by overlays

  **CLS**
  reduction = 1
  GOSUB ShowFrames
  CALL MOVETO(100,280)
  IF file$ = "" THEN PRINT "Choose an option from the MacAnimate Menu.";
  IF file$ <> "" THEN PRINT "Current filename: "; FN FileName$(file$); MENU 6, 0, 1 'restore menu
  RETURN

ShowSettings:

- ___________ during animation, show delay and x,y speeds

  CALL MOVETO(10, wind2H - 30) : PRINT "Delay = "; delay
  CALL MOVETO(10, wind2H - 10) : PRINT "X speed = "; xSpd " Y speed = "; ySpd
  RETURN

EditFields:

- ___________ open sequence edit fields

  seqEnd = false : col = INT(maxSeq / 5)
  FOR n = 1 TO maxSeq
    l = leftEd + ((n - 1) MOD col) * 40 : t = topEd + INT(((n - 1) / col) * 20
    IF ((n - 1) MOD col) = 0 THEN CALL MOVETO(l - 28, t + 12) : PRINT n;
    IF seq(n) = -1 THEN seqEnd = true
    IF seqEnd THEN n$ = "" ELSE n$ = STR$(seq(n))
    EDIT FIELD n, n$, (l,t) - (l + 30, t + 15), 2
  NEXT n
  EDIT FIELD 1 : eField = 1
  RETURN

ShowFrames:

- ___________ show all frames

  bw = bitsW / reduction : bh = bitsH / reduction 'use reduction ratio
  showL = 10 : showT = 5
  FOR n = 1 TO maxFrames
    x = showL + ((n - 1) MOD fPerLine) * (bw + 15)
    y = showT + INT(((n - 1) / fPerLine) * (bh + 20)
    LINE (x - 1, y - 1) - (x + bw, y + bh), b
    PUT (x,y) - (x + bw - 1, y + bh - 1), frame(1, n)
    CALL MOVETO(x + ((bw - 20) / 2), y + bh + 15)
    PRINT "; "; RIGHT$(STR$(n), LEN(STR$(n)) - 1)
  NEXT n
  RETURN

OpenWindow:

- ___________ open second window

  WINDOW 2, (wind2L,wind2T) - (wind2L + wind2W, wind2T + wind2H), 2
  RETURN

BringIn:

- ___________ bring in overlay number newOvr

  ovr$ = "MacAnimate.Ovr" + MID$(STR$(newOvr), 2, 1)
  overlay = newOvr
  CHAIN MERGE ovr$, 5000, ALL, DELETE 5000-5999
OverlayData:

DATA 0

Initialize:
DEFINT A - Z
OPTION BASE 1
maxSeq = 50
bitsW = 64 : bitsH = 64
fatMac = (FRE(0) > 30000)
IF fatMac THEN maxFrames = 18 ELSE maxFrames = 12
arraySize = INT(1 + bitsW / 16) * (bitsH + 1) + 2
DIM frame(arraySize,maxFrames)
DIM clip(arraySize)
DIM seq(maxSeq)
DIM pencil(34)
DIM gridPat(4), btRect(4)
FOR n = 1 TO maxSeq : seq(n) = -1 : NEXT
RESTORE PencilData
FOR n = 1 TO 34 : READ pencil(n) : NEXT
gridPat(1) = &H44FF : gridPat(2) = &H4444
gridPat(3) = &H44FF : gridPat(4) = &H4444
false = 0 : true = NOT false
white = 30 : black = 33
patBic = 11 : patOr = 9
wind2L = 16 : wind2T = 55
wind2W = 479 : wind2H = 270
fatSize = 4
fPerLine = 6
crntS = 1 : delay = 500
GOSUB ClearAll
WINDOW 1, "", (2,24) - (508,337), 4
CLS : CALL TEXTFONT(0) : CALL TEXTSIZE(36) 'show title screen
CALL MOVETO(108,130) : PRINT "MacAnimate!"
CALL TEXTSIZE(12)
CALL MOVETO(144,170) : PRINT "A frame animation program."
CALL TEXTFONT(1)
CALL MOVETO(100,280)
PRINT "Choose an option from the MacAnimate Menu."

MENU 6, 0, 1, "MacAnimate" 'install MacAnimate Menu
MENU 6, 1, 1, "Open"
MENU 6, 2, 0, "Save"
MENU 6, 3, 1, "Save As"
MENU 6, 4, 1, "Animate"
MENU 6, 5, 1, "Set Sequence"
MENU 6, 6, 1, "Frame Edit"
MENU 6, 7, 1, "Clear All"
MENU 6, 8, 1, "Instructions"
MENU 6, 9, 0, "-"
MENU 6, 10, 1, "Quit to BASIC"
MENU 6, 11, 1, "Quit to Desktop"
RETURN

PencilData: 'cursor data from MacCursor
DATA 1920, 1088, 2112, 3200, 4992
DATA 4352, 8448, 8704, 16896, 17408
DATA -31744, -30720, -4096, -8192, -16384
DATA -32768, 1920, 1984, 4032, 3968
DATA 8064, 7936, 16128, 15872, 32256
DATA 31744, -1024, -2048, -4096, 0
DATA 0, 0, 15, 0

5999 '**************************** END OF OVERLAY 0

GOSUB EditOvr
ON MENU GOSUB MenuEvent : MENU ON
GOSUB AfrtMenu
GOTO MainLoop

OverlayData: 'number of this overlay
DATA 1
'editOvr:
  EditOvr: edit frames
  GOSUB OpenWindow
  bx1 = 10 : by1 = 10  
  bx2 = bx1 + newSize * bitsW - 2
  by2 = by1 + newSize * bitsH - 2
  bRect(1) = by1 : bRect(2) = bx1
  bRect(3) = by2 + 1 : bRect(4) = bx2 + 1
  sx1 = bx2 + 20 : sy1 = by1
  sx2 = sx1 + bitsW - 1 : sy2 = sy1 + bitsH - 1

  'fn InBits returns true if (x,y) is in bit area

  DEF FN InBits(x,y) = (x >= bx1 AND x <= bx2 AND y >= by1 AND y <= by2)

  btnX = 275 : btnY = 145  
  RESTORE BtnData
  FOR n = 0 TO 12
     READ btn$
     bc = btnX + (n MOD 3) * 70
     br = btnY + INT(n / 3) * 25
     BUTTON n + 1, 1, btn$, (bc, br) - (bc + 60, br + 20)
  NEXT n

  BtnData:
  DATA "Fwd", "Up", "Cut", "Back", "Down", "Copy"
  DATA "Cpy Fwd", "Left", "Paste", "Grid", "Right", "Invert"
  DATA "OK"

  CALL MOVETO(bnx + 20, bny + 100) : PRINT "Click on a fatbit"
  CALL MOVETO(bnx + 20, bny + 116) : PRINT "to change it."
  OK = false : gridMode = patBic

  WHILE NOT OK
     CALL MOVETO(sx2 + 20, sy1 + 20) : CALL TEXTFONT(0)
     PRINT "Frame #"; cntS
     CALL TEXTFONT(1)
     PUT (sx1, sy1), frame1, crnS, PSET
     LINE (bx1 - 1, by1 - 1) - (bx2 + 1, by2 + 1), b
     PUT (bx1, by1) - (bx2, by2), crnS, PSET
     CALL PENVMODE(gridMode) : CALL PEN PAT(VARPTR(gridPat(1)))
     CALL PAINTRECT(VARPTR(bRect(1)))
     CALL PENORMAL

     newFrame = false
     WHILE NOT (newFrame OR OK)
        status = MOUSE(0) : x = MOUSE(1) : y = MOUSE(2)
        IF NOT FN InBits(x,y) THEN CALL INTCURSOR : GOTO NotDown
        IF FN InBits(x,y) THEN CALL SETCursors(VARPTR(pencil(1)))

        IF status <> -1 THEN NotDown
           cx = INT(x - bx1) / newSize
           cy = INT(y - by1) / newSize

           'cursor is down in fatbit area
           'calculate which bit cursor is on
           'reverse color of current bit
           IF POINT(sx1 + cx, sy1 + cy) = black THEN clr = white ELSE clr = black

           GOSUB AffectBits
NotDown:
   IF DIALOG(0) <> 1 THEN NoButton
   btn = DIALOG(1)
   ON btn GOSUB Fwd, Shift, Cut, Back, Shift, Copy, CopyFwd, Shift, Paste, Grid, Shift, Invert, OKBtn
NoButton:
   GOSUB Fwd, Shift, Cut, Back, Shift, Copy, CopyFwd, Shift, Paste, Grid, Shift, Invert, OKBtn
   WEND 'while not newFrame
   WEND 'while not OK
   CALL INITCURSOR
   WINDOW CLOSE 2
   RETURN

Fwd:
   newFrame = true : crntS = (crntS MOD maxFrames) + 1
   RETURN

Shift:
   b = INT((btn - 2) / 3)
   x = (1 - (b = 2) * -2) * (b > 1)
   y = (1 - (b = 0) * -2) * (b < 2)
   GET (sX1 + x, sY1 + y) - (sX2 + x, sY2 + y), frame(1, crntS)
   newFrame = true
   RETURN

Cut:
   GOSUB ToClip
   LINE (sX1, sY1) - (sX2, sY2), white, bf
   GET (sX1, sY1) - (sX2, sY2), frame(1, crntS)
   newFrame = true
   RETURN

Back:
   newFrame = true
   crntS = crntS - 1 : IF crntS = 0 THEN crntS = maxFrames
   RETURN

Copy:
   GOSUB ToClip
   RETURN

CopyFwd:
   GOSUB Fwd : GET (sX1, sY1) - (sX2, sY2), frame(1, crntS)
   RETURN

Paste:
   GOSUB FromClip : newFrame = true
   RETURN

Grid:
   newFrame = true
   IF gridMode = patBic THEN gridMode = patOr ELSE gridMode = patBic
   RETURN
Invert:
LINE (sX1,sY1) - (sX2,sY2), black, bf
PUT (sX1,sY1), frame(1,crntS), XOR
GET (sX1,sY1) - (sX2,sY2), frame(1,crntS)
newFrame = true
RETURN

OKBtn:
OK = true
RETURN

ToClip:                                        transfer image to clip array
GET (sX1,sY1) - (sX2,sY2), clip
RETURN

FromClip:                                     move image from clip array to frame array
PUT (sX1,sY1), clip, PSET
GET (sX1,sY1) - (sX2,sY2), frame(1,crntS)    'put into frame
RETURN

AffectBits:                                    affect bits until button is released
WHILE status < 0                                'while button is still held down
    IF NOT FN lnBits(x,y) THEN OutSide
    cX = INT((x - btX1) / fatSize)  'button down in fatbit area at (cX,cY)
    cY = INT((y - btY1) / fatSize)
    PSET (sX1 + cX, sY1 + cY), clr
    x = btX1 + cX * fatSize : y = btY1 + cY * fatSize
    LINE (x,y) - (x + fatSize - 2, y + fatSize - 2), clr, bf
OutSide:
    status = MOUSE(0) : x = MOUSE(1) : y = MOUSE(2)
WEND
GET (sX1,sY1) - (sX2,sY2), frame(1,crntS)    'save new frame
RETURN

5999  ENC OF OVERLAY 1

5000  ENC OF OVERLAY 2: File I/O

ON item GOSUB OpenFileOvr, SaveFileOvr, SaveAsOvr
ON MENU GOSUB MenuEvent : MENU ON
GOSUB AfterMenu
GOTO MainLoop

OverlayData:                                     'number of this overlay
DATA 2
OpenFileOvr:

```
' read frame file and sequence file
f$ = FILES$(1,"TEXT")
IF f$ = "" THEN NoLoad
IF UCASE$(RIGHTS$(f$,-4)) = ".SEQ" THEN f$ = LEFTS$(f$, LEN(f$) - 4)
GOSUB ClearAll

OPEN f$ FOR INPUT AS #1
pict = 0
WHILE NOT EOF(1)
  INPUT #1, bytes
  pict = pict + 1
  pict$ = INPUT$(bytes,1)
  IF pict > maxFrames THEN NoMore
  CLS : PICTURE (0,0), pict$
  GET (0,0) - (bitsW - 1, bitsH - 1), frame(1,pict)
NoMore:
  pict$ = INPUT$(1,1)
WEND
CLOSE #1
```

file$ = f$
MENU 6, 2, 1
f$ = f$ + ".Seq"
fileHere = true
ON ERROR GOTO FileError
OPEN f$ FOR INPUT AS #1
ON ERROR GOTO 0
IF NOT fileHere THEN NoSeqLoad
j = 0
WHILE NOT EOF(1)
  j = j + 1 : INPUT#1, seq(j)
WEND
CLOSE#1
NoSeqLoad:
NoLoad:
RETURN

SaveFileOvr:

```
' save frames and sequence to file$
OPEN file$ FOR OUTPUT AS #1
FOR n = 1 TO maxFrames
  CLS
  PICTURE ON : PUT (0,0), frame(1,n) : PICTURE OFF
  pict$ = PICTURES$
  PRINT #1, len(pict$)
  PRINT #1, pict$
  pict$ = ""
NEXT n
CLOSE #1
```

f$ = file$ + ".Seq"
OPEN f$ FOR OUTPUT AS #1
FOR n = 1 TO maxSeq : PRINT#1, seq(n) : NEXT
CLOSE#1
RETURN
SaveAsOvr: ___________________________ save to new file name
f$ = FILES$(0, "Save as what file?")
IF f$ = "" THEN RETURN
file$ = f$ : MENU 6, 2, 1
GOSUB SaveFileOvr
RETURN

File Error: ________________ fileHere = false, if error is "File not Found"
IF ERR <> 53 THEN ON ERROR GOTO 0
fileHere = false
RESUME NEXT

5999 **************************** END OF OVERLAY 2

5000 **************************** OVERLAY 3: INSTRUCTIONS
GOSUB InstOvr
ON MENU GOSUB MenuEvent : MENU ON
GOSUB AttrMenu
GOTO MainLoop

Overlay Data:
DATA 3

InstOvr: ___________________________ show instruction screen
WINDOW 2, (16,55) - (495,325), 2
PRINT "MacAnimate Menu"
PRINT
PRINT "Open: Loads a previously saved frame and sequence file if available."
PRINT "Save: Saves frame set and sequence under current filename."
PRINT "Save As: Saves frame set under a filename the user specifies."
PRINT "Animate: Shows an animation of the current frame set."
PRINT "Set Sequence: Allows you to specify an animation sequence."
PRINT "Frame Edit: Edit individual frames with a MacPaint FatBits-like editor."
PRINT "Clear All: Resets all frames to blank."
CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1 : WINDOW CLOSE 2
RETURN

5999 **************************** END OF OVERLAY 3
II. GAMES

- MacDart
- MacBandit
- MacMouse
- MacBall
- MacGurkha
- MacSpace
You're having a relaxing evening with your friends, standing around with mugs of your favorite brew, your darts, and your... Macintosh? Well, why not? Thanks to the Mac's wizardry, the local pub's time-honored dart game can now be played in the comfort of your computer screen. And no one will end the fun with the closing-time call of "Time, gentlemen, please!"

MacDart
How to play MacDart

When you first start MacDart, the title screen shown in Figure 1 appears. It shows a big dart on the right and a fat, happy balloon on the left. You then hear a brief sweep of sound. A moment later, the initial playing screen appears as shown in Figure 2. It shows a little balloon sitting in the lower left corner. Along the bottom of the screen you'll see your current score and the speed that the current dart will have once it is thrown (more about that later). On the right is a stack of nine darts and one that is ready to be thrown. Your goal in MacDart is to pop the balloon before it reaches the top of the screen. The MacDart menu contains these options:

FIGURE 1. The MacDart title screen
If you move the mouse a bit, you'll discover that the top dart moves up and down but not sideways. Think of yourself standing on the throwing line, jiggling the dart to get a feel for its weight. You can do this as long as you like; nothing else will happen until you double-click the mouse button. When you do, the balloon starts to rise toward the top of the screen, as shown in Figure 2. When the balloon rises, sometimes it will float up very slowly, and sometimes it will rocket toward the top of the screen. Fast or slow, the balloon always rises straight up.

After launching the balloon, you can throw the dart at it by single-clicking the mouse button. You will notice that the dart does not move in a straight line. Just like in your favorite pub, MacDart's darts drop with the effect of gravity. To get really good at the game, you must learn to estimate how far above the balloon to aim in order to break it. With practice, you'll learn how the trajectory, or path, the dart takes toward the target is related to the dart speed shown at the bottom of the screen.

If the dart misses the balloon, or if the balloon rises past the top of the screen, you score no points for that dart. When you do manage to find
your range and your dart hits the balloon (or its string), you'll hear a sound and see the balloon disintegrate in a pattern of lines. One to six points will be added to your score, depending on how fast the balloon was moving: The faster it was going, the more points you get. If the dart misses the balloon and there is still time to throw another dart, just aim and press the mouse button again—the next dart will shoot off.

Let's consider what's happening as the dart moves toward the target. Faster darts (speeds of about 80 up to the maximum of 100 mph) fly in almost a straight line, dropping only a little as they go across the screen. Slower darts (from the minimum of 50 to about 70 mph) tend to fall in an arc and drop much farther before they reach the left side of the screen. Indeed, if the dart is slow enough and wasn't released from a high enough altitude, it will fall through the bottom of the screen before it even reaches the area of the target!

If you think about the difference between lobbing a baseball with a nice, slow underhand motion and coming over the top with a blazing fast ball, the behavior of the dart should begin to make sense. The MacDart program includes simulated "gravity," so the darts will fall (move down the y axis) as they move left along the x axis. As with real gravity, the longer the dart falls, the faster its rate of falling will be and the farther it will fall in a given unit of time.

Do slow darts fall faster than fast ones? If you have a stopwatch around the house, you can experiment for yourself, releasing darts of different speeds from the same height and measuring how far each has fallen after a few tenths of a second. In fact, as you may remember if you've had high school physics, the horizontal speed of an object has no effect on how fast or far the object falls. The only thing that matters (discounting friction and air resistance, of which there's none on the Mac screen unless you add it to your simulation) is how long the object has been falling. While falling the same distance, the fast dart moves farther at every moment across the screen, so it gets to the target before it's had much time to fall very far.

How does this physics lesson help with the job of balloon-busting? You can see that you have to "lead" the target (that is, position the dart higher than the balloon) much more for slow darts than for fast ones. You also have to judge the balloon speed so as to pick a rendezvous point where both the dart and the balloon will arrive at the same time. Not only that, for very slow darts you have to pick an intersection point low enough to accommodate the fact that the slow dart will have dropped quite a ways by the time it reaches the left border of the screen. If you've got a real slow
dart and a real fast balloon, things can get very sticky! It might be comforting for you to know that this aiming and "leading" the target is the same problem fighter pilots face shooting in dogfights, multiplied by three dimensions instead of just two!

How MacDart works

MacDart shows several interesting programming ideas. At the beginning, the random-number generator is reseeded with a value from the timer, certain constants are set, and the window is sized. The dart and the balloon images are formed, the MacDart menu is created, and then the initial title screen and instructions are displayed on the screen. The main program loop, which makes use of nested WHILE...WEND loops, is called next. Following the main program loop is the section that handles menu events, and then a section that contains various subroutines for moving the balloon, the dart, and both at the same time, as well as checking to see if the balloon has popped. The structure of the program is shown in outline form in Figure 3.

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Init 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize variables</td>
<td></td>
</tr>
<tr>
<td>create output window</td>
<td></td>
</tr>
<tr>
<td>make dart and balloon shapes</td>
<td></td>
</tr>
<tr>
<td>draw title screen</td>
<td></td>
</tr>
<tr>
<td>install MacDart menu and enable menu event-trapping</td>
<td></td>
</tr>
<tr>
<td>show instructions (Instructions)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Program</th>
<th>Main 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE true (endless loop until interrupt by menu event)</td>
<td></td>
</tr>
<tr>
<td>WHILE there are darts left</td>
<td></td>
</tr>
<tr>
<td>redraw initial screen (InitScreen)</td>
<td></td>
</tr>
<tr>
<td>until double-click (balloon release)</td>
<td></td>
</tr>
<tr>
<td>move dart along yaxis only (UpdateDart)</td>
<td></td>
</tr>
<tr>
<td>WHILE balloon not at top of screen, popped, or no more darts</td>
<td></td>
</tr>
<tr>
<td>move balloon (MoveBal)</td>
<td></td>
</tr>
<tr>
<td>if balloon reaches screen top, go to next dart (SubDart)</td>
<td></td>
</tr>
<tr>
<td>if dart is released (single-click), move both objects (MoveBoth)</td>
<td></td>
</tr>
<tr>
<td>WEND</td>
<td></td>
</tr>
<tr>
<td>WEND</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Main 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>game over:</td>
<td></td>
</tr>
<tr>
<td>update score (GiveScore)</td>
<td></td>
</tr>
<tr>
<td>display 'game over' message</td>
<td></td>
</tr>
<tr>
<td>idle loop until another menu choice, to play again or quit</td>
<td></td>
</tr>
<tr>
<td>WEND</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 3. Program outline
Menu Events

MenuEvent: deactivate menu and branch to appropriate subroutine  
          ME 0

PlayMenu: reset game if user wants to play  
          ME 1

Instructions: show instructions  
            ME 2

BasQuit: quit to BASIC  
         ME 3

DeskQuit: quit to Mac Desktop  
       ME 4

Subroutines

InitScreen: initialize screen to launch new balloon  
            SR 1

MoveBal: move only balloon until double-click  
       SR 2

MoveBoth: move both objects until either balloon reaches top,  
          dart reaches left side, or balloon is popped  
       SR 3

SubDart: subtract one dart from total left and redisplay remaining  
       SR 4

GiveScore: print points and dart speed  
        SR 5

UpdateDart: update dart position  
            just return if no movement  
            show cursor if dart above window  
        SR 6

UpdateBal: update balloon position  
        SR 7

PopTest: test for dart and balloon in same place; pop balloon if hit  
        SR 8

Figure 3. Program outline (continued)

Initialization and drawing objects

This program starts by reseeding the random-number generator with a  
number from the timer, dimensioning arrays for the dart (dart%) and bal-  
loon (bal%) images [Init 1], and setting a somewhat arbitrary constant  
representing our simulated gravity. Next comes the sizing of the window  
in which the game will be played. In [Init 2] the dart shape is drawn and  
then a GET statement stores it in the array dart%. We then get rid of the  
shape on the screen with a PUT statement. (Remember that a PUT state-  
tment, unless otherwise specified, does an XOR with the shape existing at  
the specified location; this has the effect of erasing the previously drawn  
shape if the same shape is PUT on top of itself.)  

The same procedure is followed in creating the balloon shape.  
Note how the two Ms that are part of the balloon shape are drawn using  
built-in QuickDraw routines. The TEXTMODE is set to 3. Mode 3 is the
MAC DART

BIC or “black is changed” mode, which means that when the two Ms are printed, they appear white against the black balloon (this is another example of how text and graphics can be made to interact in interesting ways on the Macintosh).

Title screen, instructions, and MacDart menu

Next the program displays the title screen and instructions for playing the game. The title screen procedure [Init 3] starts out by drawing a big balloon and the word MacDart! with the big dart superimposed on it, and a “sound sweep” caused by the FOR...NEXT loop produces a series of very brief sounds with regularly rising pitch.

The Instructions subroutine [ME 2] shows the player a screen with text that describes how to play the game. It also sets up a button labeled Continue to allow the player a chance to read the instructions before continuing on with the game.

The MacDart menu [Init 4] is installed next. It is set to include the title, MacDart, and the options Play Again, Instructions, Quit to BASIC, and Quit to DeskTop.

The main program

The set of WHILE...WEND loops, [Main 1] through [Main 5], handles the playing of the game until the player decides to quit (by moving to the menu and choosing Quit to BASIC). The game starts at [Main 1] where the value for the number of darts available is set to 10 at the beginning of each game round. The key to understanding this program is realizing that each “turn” (dart throw) breaks down into three states or conditions. Each state contains a loop that includes a test to see if the event triggering the next state has occurred. These three states are:

1. Player may be moving dart up and down; neither balloon nor dart has been released.
2. Player has released balloon; may still be moving dart up and down; dart not yet released.
3. Both balloon and dart have been released; player has no more input until next dart.

Keep these states in mind, and refer to the program outline while you read the following explanation of how MacDart works.
Initialize balloon and dart positions and speeds

First, the initial positions of the balloon and dart are set by a call at [Main 2] to InitScreen [SR 1]. These initial values are always set the same. The balloon position is \((xBal, yBal)\), and the active dart will start at \((xDart, yDart)\). Corresponding variables that will hold the “old” balloon and dart coordinates are assigned the same starting values: These will be used to erase the old images in animating the balloon and dart. The dart speed along the x axis \((xDartStep)\) is set to a random integer between 25 and 50. The balloon speed is set to a random integer between 1 and 6. We use the statement RANDOMIZE TIMER to ensure that the program will generate a new series of random numbers each time. Appropriately for a Midnight Madness program, this statement uses the number of seconds that have elapsed since midnight as the random “seed.”

A call to GiveScore [SR 5], while still within InitScreen, updates the display of the game situation. GiveScore prints the current number of points and the speed of the current dart times 2 as “MPH,” just to make it sound a bit more impressive.

A FOR...NEXT loop draws the darts the player has left, other than the one “in hand.” The dart in play is then drawn (with a PUT) at \((xDart, yDart)\), and the balloon is drawn at \((xBal, yBal)\).

State 1: Waiting for balloon release

The WHILE...WEND segment at [Main 3] is a loop that checks to see that the mouse has not been double-clicked \((MOUSE(0) <> 2)\), then assigns the result of the MOUSE(2) function to yDart. This value is the value of the mouse y coordinate the last time MOUSE(0) was called. A call is then made to UpdateDart [SR 6], which first checks to see if either the x or the y coordinate of the mouse pointer has changed since last time (that is, differs from \(xDartOld\) or \(yDartOld\), respectively). If neither coordinate has changed, UpdateDart simply returns. If either has changed, the y position is checked to see if it is above the window. If it is, the cursor appears. If it isn’t, the dart hasn’t been released and only the y position is changed. The dart in hand is updated by drawing it in a new y position and erasing the old dart. The current x and y values are then saved in \(xDartOld\) and \(yDartOld\) for future comparison.

This is a good place to take a short trip back to the WINDOW statement in [Init 1], where we define the window to be a little wider than the screen. Why? The operation of windows in BASIC does not allow clicking in an area outside the current output window; if this is done, the Mac will beep and do nothing. Since the pointer is hidden to prevent confusion
with the dart we’ve created, we can’t see exactly where the pointer is and it is possible to click the button with the invisible pointer outside the window. If that were to occur, all that would happen for the player would be a beep and nothing else.

We’ve taken care of the motion of the mouse pointer in the vertical direction by testing to see whether the arrow pointer should be displayed instead of the dart. We prevent the pointer from being outside the window in the horizontal direction by making the window a little wider than the screen.

After returning from UpdateDart, the program is still in the WHILE...WEND loop which means the MOUSE(0) function is checked again. This accomplishes two things. First, if MOUSE(0) = 2, a double-click has taken place, and the program will go to State 2 (balloon released). The call to MOUSE(0) also updates the value that will be read the next time MOUSE(2) is used. If there was no double-click, the program repeats the loop section by calling UpdateDart.

One important insight for writing event-driven routines is knowing when you need an interrupt like ON MOUSE GOSUB. Here, the WHILE...WEND construct is used since there are no calculations or other activities that the player might want to interrupt.

State 2: Balloon released

The code segment in [Main 4] handles the state in which the player has released the balloon but has not yet single-clicked to release the dart. This, too, is handled by a WHILE...WEND loop. First, a check is made to see that the value of yBal is greater than zero (the balloon hasn’t yet reached the top of the window), that the balloon hasn’t been popped, and that there are darts left for the player to throw. Then a call is made to MoveBal [SR 2]. MoveBal consists of a WHILE...WEND loop that checks that the balloon still hasn’t reached the top, and that the mouse has not been clicked once (MOUSE(0) <> -1) to release the dart. The yDart position is set to MOUSE(2), and UpdateDart is called to draw the new position of the dart if necessary. After the dart is redrawn, a call is made to UpdateBal [SR 7], which works much like UpdateDart. The new balloon position is calculated by subtracting yBalStep from the balloon’s y coordinate in yBal (the y coordinate decreases as you move toward the top of the screen), the new balloon is drawn, the old is erased, and the new coordinates are saved in xBalOld and yBalOld for future reference. Remember that only the y position of the balloon actually changes, because the balloon rises straight up.
After updating the balloon's position, the WHILE...WEND loop in MoveBal continues to check whether the balloon has reached the top, and whether the mouse button has been pressed. If the button hasn't been pressed, the loop continues and moves the balloon. If the player doesn't click the mouse to release the dart soon enough, the balloon will eventually reach the top of the window. If the player does press the mouse button, control will move to the IF statement at the next line in [Main 4] (more on what happens there later). If the player doesn't press the button and the balloon does reach the top (yBal <= 0), control will pass to the line that reads xDart = 450 : yDart = MOUSE(2), which simply means that the dart position is somewhere along the right edge of the window.

The program continues in the [Main 2] WHILE...WEND loop while there are darts left. When there are no more darts, the program goes to the series of commands at [Main 5] (these will be discussed later). For now, we'll show you what happens after the player throws the dart.

**State 3: Dart thrown**

In the middle of the program loop at [Main 4] there is a call to MoveBal [SR 2]. In this subroutine, the balloon is moved until there is a single-click to signal the launch of a dart. When this single-click occurs, control passes to the IF statement in the next line in [Main 4] that checks to see that the balloon hasn't hit the top of the window. If it hasn't, the program calls SubDart [SR 4]. SubDart subtracts one dart from the total left, and if there are any darts left, blanks one of the darts in the stack at the right edge of the screen by using a PUT statement in XOR mode.

Control then moves back to the main program which next calls MoveBoth [SR 3]. The idea behind this subroutine is to move both the balloon and the dart until the balloon reaches the top, the dart reaches the left side of the screen, or the balloon is popped.

Initially in this subroutine, the value of dartTime is set to zero. This is used in the simulation of gravity, which we'll get to in a moment. A WHILE...WEND loop is set up to operate while the balloon is still below the top, and the dart hasn't reached the left, and popped is false. The value of dartTime is increased by one, and xDartStep is subtracted from the current xDart position to get the new dart's x position (since the dart is moving from right to left, the x coordinate is decreasing).

The change in the dart's y position is more complicated, since it represents the effect of gravity. The y value will be increasing as the dart falls toward the bottom of the screen, but by how much? From high school
physics (or from Newton's run-in with the apple—a McIntosh apple no doubt!), you might remember the equation for the effect of gravity on a falling body:

\[ y = gt^2 \]

The distance, \( y \), traveled by a falling body in a period of time, \( t \), is equal to the rate, \( g \), at which it is falling times the square of the time that it falls. In this case, the value of \( \text{dartTime} \) (representing the length of time the dart has been falling) is substituted for \( t \), and is multiplied by itself. Then this value is multiplied by the constant variable \( \text{gravity} \) (.08). This means, for example, that at \( \text{dartTime} = 2 \) the dart will fall \((0.8 \times 2 \times 2)\) or 3.2 pixels (or 3, since the drawing routines convert these values to integers), at \( \text{dartTime} = 4 \) it will fall \((0.8 \times 4 \times 4)\) or 12 pixels, and so on. (You might try changing the value of \( \text{gravity} \) in [Init 1] to see the effect of different gravitational constants.) In this subroutine, the distance that the dart has dropped is added to the current \( y\text{Dart} \) value, which gives a new value to \( y\text{Dart} \). Having calculated the new dart position, the program first calls UpdateDart [SR 6] to draw the dart in its new position; next, the program calls UpdateBal [SR 7] to make the balloon rise another step. Finally, the dart’s x position is checked to see if it’s less than the balloon’s x position plus 20. If it is, the dart may have reached the balloon, and PopTest [SR 8] is called to find out. Otherwise, the routine goes back to the top of the WHILE…WEND loop to keep moving the dart and the balloon until either the balloon gets to the top of the window, or the dart hits the left side of the screen and disappears.

PopTest [SR 8] checks to see whether the dart’s y position is within the boundaries of the balloon’s body (including the string—if that’s hit, the balloon presumably dies of fright!). If a hit has been made, the value \( \text{popped} \) is set to true, and the SOUND statement makes the sound of the expiring balloon. The outer FOR…NEXT loop then causes the inner loop (the explosion effect) to be executed twice, once with black lines and once with white. The inner FOR…NEXT loop draws pairs of lines that converge upon the center of the balloon, decreasing constantly, representing the exploding, punctured balloon. After the first set of lines is drawn, the color is changed to white and the outer loop executes the line-drawing loop once more. Finally, PopTest adds the speed of the erstwhile balloon to the player’s total score as a measure of the difficulty of hitting it.

When PopTest returns to MoveBoth [SR 3], the value of \( \text{popped} \) is true, and control then returns to the main program at [Main 4] where the
dart is set back to the right side of the screen. Control then goes back up to [Main 2], where everything is ready for the next balloon launch.

When the tenth dart has been played, control goes to [Main 5], where GiveScore [SR 5] is called one more time to give the final score. Next, it's time to find out whether the player wants to play again.

After printing the message Game Over in the middle of the playing screen, the program returns the cursor to its normal shape with a CALL INITCURSOR. That way, the poor player isn't trying to make menu selections with an invisible pointer. Forgetting to take care of details like this produces results embarrassing to the programmer and irritating to the user. For example, someone might be asked to make choices about exiting a "bouncing vegetables" game by trying to figure out how to push a button with a tomato! This exemplifies the essential programming philosophy of always making sure that a user who leaves your program (or is about to) is returned to the normal Macintosh environment.

**Suggestions for MacDart**

You could make the game harder by having an obstacle (say a net to snag darts) moving up the middle of the screen. If a dart gets close enough, it is caught in the net and you miss! The net would begin moving as the balloon is released, and it would have a random speed perhaps 2 or 3 times that of the balloon (experiment with different values). To draw the net, base your routine on the dart- and balloon-drawing routines. To move the net, use steps similar to the balloon steps and a routine similar to UpdateBal [SR 7]. To check for a dart being caught in the net, check for the dart x coordinate being close enough to the net x coordinate. If it is, go to a routine called CaughtTest, similar to PopTest [SR 8]. (You can also embellish things with an appropriate sound if you wish.) A caught dart would, like other kinds of misses, take control back to the top of the main dart loop.

You might experiment with different shapes and sizes of nets by using MacPaint to draw them, and then using MacInterface to incorporate them into your program.

Another thing you could do is assess the player a penalty for caught darts, perhaps reducing the current score by the amount that would have been awarded for a hit on the current balloon. Have fun ... and if you go crazy adding paddle wheels, flippers, black holes, flying-dart teleporter boxes, and so on, don't blame your friendly authors. We warned you that tinkering with games is addictive!
MacDart program listing

MacDart
Throw the dart and see if you can pop the balloon!

INITIALIZE

RANDOMIZE TIMER
DIM dart%(20), bal%(70)
gravity = .08
false = 0 : true = NOT false

WINDOW 1, "", (0.34) - (512,324), 2

CIRCLE (20,10), 8, ,,, 3
LINE (22,8) - (30,6) : LINE (30,6) - (28,10)
LINE (22,12) - (30,14) : LINE (30,14) - (28,10)
LINE (12,10) - (8,10)
GET (8,6) - (30,14), dart% : PUT (8,6), dart%

FOR size = 0 TO 10
  CIRCLE (20,20), size, ,,, 1.5
NEXT size
LINE (20,30) - (17,33) : LINE (17,33) - (20,35)
CALL TEXTMODE(3)
CALL MOVETO(15,22) : PRINT "M"
CALL MOVETO(17,28) : PRINT "M"
GET (10,10) - (30,35), bal% : PUT (10,10), bal%

PUT (30,30) - (230,280), bal%
CALL TEXTFACE(1) : CALL TEXTMODE(1)
CALL TEXTSIZE(18)
CALL MOVETO(270,250) : PRINT "MacDart!"
PUT (200,200) - (420,280), dart%
FOR n = 20 TO 80 STEP 3
  SOUND n, 2
NEXT
FOR t = 1 TO 3000 : NEXT

MENU 6,0,1,"MacDart"
MENU 6,1,0,"Play Again"
MENU 6,2,0,"Instructions"
MENU 6,3,0,""
MENU 6,4,1,"Quit to BASIC"
MENU 6,5,1,"Quit to DeskTop"
ON MENU GOSUB MenuEvent : MENU ON

GOSUB Instructions
MAIN PROGRAM

WHILE true
  CALL HIDE_CURSOR: cursFlag = false
  points = 0: dart = 10

WHILE dart > 0
  WHILE MOUSE(0) <> 0: WEND
  GOSUB InitScreen

WHILE MOUSE(0) <> 2
  yDart = MOUSE(2)
  GOSUB updateDart
  WEND

WHILE yBal > 0 AND NOT popped AND dart > 0
  GOSUB MoveBal
  IF yBal > 0 THEN GOSUB SubDart
  GOSUB MoveBoth
  xDart = 450: yDart = MOUSE(2)
  WEND
  FOR t = 1 TO 1000: NEXT
  WEND

GOSUB GiveScore
  CALL MOVETO(160,100): PRINT "Game Over"
CALL INIT_CURSOR
  MENU 6,1,1
  MENU 6,2,1
  playAgain = false
WHILE NOT playAgain: WEND
  MENU 6,1,0
  MENU 6,2,0

MENU EVENTS

MenuEvent: 'on menu event, deactivate menus and branch to subroutine
  MENU 6,0,0
  item = MENU(1)
  ON item GOSUB PlayMenu, Instructions, , BasQuit, DeskQuit
  MENU 6,0,1
  RETURN

PlayMenu: 'reset game if user wants to play again
  playAgain = true
  RETURN
Instructions:

'Show instructions

WINDOW 2, (16,55) - (495,325), 2
PRINT
PRINT "Instructions for MacDart"
PRINT
PRINT "See if you can pop the balloon with the dart."
PRINT "Double-click the mouse button to release the balloon."
PRINT "Then single-click the button to throw the dart."
PRINT "If the dart hits the balloon, the balloon will explode."
CALL INITCURSOR

BUTTON 1, 1, "Continue", (190,240) - (280,260) 'create button
WHILE DIALOG(0) <> 1 : WEND 'and wait for button click
BUTTON CLOSE 1 : WINDOW CLOSE 2
CLS
RETURN

BasQuit:
FOR n = 1 TO 2 : WINDOW CLOSE n : NEXT
MENU RESET
END

DeskQuit:
MENU RESET
SYSTEM

'Bas Quit:
FOR n = 1 TO 2 : WINDOW CLOSE n : NEXT
MENU RESET
SYSTEM

Desk Quit:
MENU RESET
SYSTEM

------------------------------- SUBROUTINES-------------------------------
InitScreen: 'initialize screen to launch new balloon
xBal = 20 : xBalOld = xBal
yBal = 260 : yBalOld = yBal
xDart = 450 : xDartOld = xDart
yDart = 100 : yDartOld = yDart
xDartStep = INT(25 * RND(1) + 25) 'dart speed
yBalStep = INT(6 * RND(1)) + 1 'balloon speed
CLS
popped = false
GOSUB GiveScore
FOR d = 1 TO dart - 1 'display remaining darts
  PUT(450, 280 - 6 * d), dart%
  'in lower right corner
NEXT
PUT(xDart,yDart), dart% : PUT(xBal,yBal), bal% 'draw objects
RETURN

MoveBal: 'move only balloon until double-click
WHILE yBal > 0 AND MOUSE(0) <> -1
  yDart = MOUSE(2) : GOSUB updateDart
  GOSUB updateBal
WEND
RETURN
MoveBoth: 'move both until either balloon reaches top, dart reaches left side, or balloon is popped

dartTime = 0
WHILE yBal > 0 AND xDart > -20 AND popped = false
  dartTime = dartTime + 1
  xDart = xDart - xDartStep
  yDart = yDart + (gravity * dartTime * dartTime)
GOSUB updateDart
GOSUB updateBal
IF xDart <= xBal + 20 THEN GOSUB PopTest
WEND
RETURN

SubDart: 'subtract one dart from total left

dart = dart - 1
IF dart > 0 THEN PUT(450, 280 - 6 * dart), dart%
RETURN

GiveScore: 'print points and speed
LINE (50,250) - (430,330), 30, bf
CALL MOVETO(50,280)
PRINT points; "points, dart speed"; xDartStep * 2; "MPH"
RETURN

updateDart: 'update dart position; show cursor if dart above window

IF xDart = xDartOld AND yDart = yDartOld THEN RETURN

IF (yDart < -5) = cursFlag THEN CursorOK
IF cursFlag THEN CALL HIDE_CURSOR ELSE CALL SHOW_CURSOR

cursFlag = NOT cursFlag

CursorOK:
  PUT(xDart,yDart), dart%
  PUT(xDartOld,yDartOld), dart%
  xDartOld = xDart : yDartOld = yDart
RETURN

updateBal: 'update balloon position

yBal = yBal - yBalStep
PUT(xBal,yBal), bal%
PUT(xBalOld,yBalOld), bal%
xBalOld = xBal : yBalOld = yBal
RETURN

PopTest: 'test for dart and balloon in same place, pop balloon if hit

IF NOT (yDart < yBal + 20 AND yDart > yBal - 5) OR xDart < 5 THEN RETURN
popped = true
FOR z = 0 TO 5
  SOUND 120, 3, 255 - (50 * z)
NEXT z
xExp = xBal + 10 : yExp = yBal + 10
color = 33
FOR n = 0 TO 1
    FOR r = n TO 25 STEP 3
        rfac = r/2 + 12
        LINE(xExp - rfac, yExp + r) - (xExp + rfac, yExp - r), color
        LINE(xExp - rfac, yExp - r) - (xExp + rfac, yExp + r), color
    NEXT
color = 30
NEXT
points = points + yBalStep
RETURN
MacBandit

Have you ever won it big (or even little) at Las Vegas? No? Well, try your luck on the world's most sophisticated and user-friendly slot machine: your Mac plus MacBandit! We offer a penny slot (try and find that in Vegas these days!), and scrupulously fair odds (ditto!). You can see and hear the symbols roll in the windows, and hear a cheery jingle when you win.
Sorry, we can’t offer a real payoff! You’ll still have to go to Nevada for the bountiful buffets and the floor shows, too. Still, we think you’ll find MacBandit a lot of fun. Next time you have a party, leave this game up on your Mac and we bet you’ll soon find half your guests clustered around it.

How to play MacBandit

Playing MacBandit is simplicity itself. When the program starts running, the slot machine is drawn on the left side of the screen, and on the right side you’ll see four piles of five coins each, as shown in Figure 1. Below the coins are given the totals of coins won in the current play (0 to start) and the coins in your possession (20 to start). As the instructions above the slot machine explain, simply grab any coin with the mouse pointer, drag it over to the machine’s deposit slot, and release it to start the machine.

After you “deposit” your coin, the word MAC briefly appears in the first of the machine’s three windows. Then the windows seem to flicker as the various images (Mac, cherry, bell, lemon, orange, plum) appear, accompanied by a clicking sound from the Bandit’s “mechanism.”

![Diagram of MacBandit playing screen and menu](image)
FIGURE 2. The payoff schedule

What happens next depends on how lucky you were. The payoff depends on whether two or three symbols matched, and which symbols they were. To see these payoff amounts (while the machine isn't in play), pull down the MacBandit menu and choose the PayOffs item: The display in Figure 2 appears with the symbol combinations and their payoffs. Note that matching three symbols pays more than matching two, and that the Mac symbol pays the most in both cases. If you match three Macs you win the "Macpot," and you not only get the 40-coin payoff, you also get all coins deposited since the last Macpot (or the beginning of the game, if there is no previous Macpot).

When you get a jackpot, the machine appears to disgorge a pile of coins comprising your winnings, and a musical selection plays. A two-match win plays "Pop Goes the Weasel," while a three-match win calls more of the Mac's musical resources into play for a rendition of "Happy Days Are Here Again."

Unfortunately, you can't take your winnings down to the local computer store and get that second disk drive you've always wanted for your Mac: All you can do is use the extra coins to play some more. (Come to think of it, that's even better house odds than in Nevada, because no
matter what it gives you, sneaky MacBandit *always* gets all its money back!) After the jackpot has spilled out of the machine, MacBandit picks it up and carefully stacks it with your other coins, and updates the total for that jackpot and your total coins.

How to beat MacBandit? You’re kidding: You could easily rig the machine by altering the program code, but it would be like cheating at solitaire. We don’t recommend jiggling the Mac either; although no bouncer will escort you to the door, it may not be good for your computer’s innards. Some things in life just have to be played where they lie.

### How MacBandit works

As you can see in the program outline in Figure 3, the program’s first task is to initialize a number of constants and arrays. The arrays contain pictures of the fruit, the results for each window, the payoff rates for the fruit, two views of the coins, the coin-slot rectangle, the coin-stacks rectangle, and the values for the pile size for the payout coins. Subroutines called in the early stage of the program form the image arrays for the coins and the fruit. The MacBandit menu is set up with the four options: Play Again, PayOffs, Quit to BASIC, and Quit to Desktop.

---

**Initialization**

- define size of graphics arrays and constants
- define "point in rectangle function"
- create playing window
- get coin image arrays (*FormCoins*)
- display playing screen (*DrawScreen*)
- save fruit symbols (*SaveItems*)
- install MacBandit menu, enable menu event-trapping

**Main Program**

```basic
WHILE true (endless loop interrupted by menu events)
  set up initial stack of coins (*ShowStack*)
  WHILE coins left
    wait for player to deposit a coin (*DepositCoin*)
    erase machine windows
    'spin' the fruit by showing symbol sequences (*SpinFruit*)
    put random final fruit in each window
    check for matches
    branch to appropriate subroutine (*MatchTwo, MatchThree*)
    update player's score (*UpdateCount*)
  WEND
  wait for menu event to restart game
WEND
```

FIGURE 3. *Program outline*
**Menu Events**
- **MenuEvent**: branch to appropriate subroutine  
  
- **PlayAgain**: set flag to play another game  
  
- **PayOffMenu**: set flag to show payoffs while waiting to deposit coin  
  
- **BasQuit**: quit to BASIC  
  
- **DeskQuit**: quit to Mac Desktop  

**Subroutines**
- **DepositCoin**: wait for player to deposit coin  
  - WHILE coin not deposited  
    - WHILE waiting for button press  
      - if payoff flag set, display payoffs (Payoffs)  
    - WEND  
  - if pointer is in stack area:  
    - hide pointer  
    - subtract coin (SubCoin)  
    - animate dragging of coin  
    - show pointer  
    - if coin not dragged to slot, add coin back (AddCoin)  
  - WEND  
  - increment number of coins deposited  

- **SpinFruit**: simulate spinning of fruit  
  - show random fruit in each window  
  - choose and show random final fruits  

- **Payoffs**: show game payoffs  
  - open second window  
  - display two-match payoffs (ShowFruit)  
  - display three-match payoffs (ShowFruit)  
  - create Continue button, wait for button click  
  - close second window, redraw screen (DrawScreen)  
  - show stack of coins (ShowStack)  

- **ShowFruit**: display specified fruit  
  
- **MatchTwo**:  
  - determine payrate  
  - pile coins up at bottom of machine (PileCoins)  
  - play Pop Goes the Weasel (Weasel)  
  - transfer coins to stack (UnpileCoins)  

- **MatchThree**:  
  - determine payrate  
  - pile coins up at bottom of machine (PileCoins)  
  - play Happy Days Are Here Again (HappyDays)  
  - transfer coins to stack (UnpileCoins)  

- **PileCoins**: make pile of coins below machine  

*(continued)*
MacBandit uses a WHILE...WEND loop structure with calls to subroutines and a subprogram to perform most of the work of the game. The outer WHILE...WEND loop sets up the stack rectangle and continues with an inner WHILE...WEND loop that lasts as long as you have money and have not quit the game. This loop calls DepositCoin [SR 1a], which goes into a WHILE...WEND loop that waits for you to deposit your coin. Within this loop, your mouse movements are checked to see if you picked up a coin, where you moved it, and if you dropped it in the slot.

Once that happens, the program exits DepositCoin and returns to erase the windows in the slot machine in the main program. The main program then calls SpinFruit [SR 2a], which does just that: spins the fruit in their windows. After returning to the main program, the results are checked, and appropriate payoffs are made by calling MatchTwo [SR 5] or MatchThree [SR 6] and UpdateCount [SR 9]. This ends the inner loop. The Play Again menu item is enabled, and the program sits patiently waiting for your next move.
Initialization (and some programming tips)

There are a few things to notice about [Init 1]. Since this program uses a number of arrays to hold graphic objects (including two views of a coin), a CLEAR statement is used to reserve enough space in BASIC's data segment for the program text, arrays, and other variables. It's always a good idea to look at your program's data structures and reserve a generous amount of space for them. By reserving space this way, you won't get an Out of Memory error the first time you run the program—or worse, someone else won't get one due to some peculiar combination of inputs that you hadn't anticipated.

Also note the definitions of false and true. If you're used to programming in a language such as Pascal that has these values built in, be warned that you have to define them in Microsoft BASIC. Defining these logical values and using them in loop-control conditions makes your code more understandable.

There are several other touches here that make for programs that are easier to understand and run faster. You probably use many of them yourself, but it's worth reviewing them:

- When all your variables are going to be integers, say so with DEFINT. Integer variables take up less space than other variable types (such as SNG and DBL) and are calculated much faster.
- Name all the values you're going to use—even fixed constants—as variables.
- Use the variable names to dimension arrays involving the appropriate quantities. This makes the code easier to understand and therefore easier to maintain or modify. For example, even a beginning programmer who wants to start with 50 coins rather than 20 can change startCoins to 50 and expect reasonable results.

In [Init 2] we define the function PtinRect. This simply checks whether point (x,y) is within a rectangle with given side coordinates. We'll use it to see if the player has managed to get a coin into the deposit slot and also to check whether the player has started to drag the pointer in the coin-stack area. We also define the function Minimum, which is simply a crafty way of finding which of two numbers is smaller.

There are two good reasons for using functions like these. First, it obviously makes sense to have a complex condition defined as a function so a lengthy segment of code need appear only once in the program and the function can be called with any appropriate variables. However, in this program the function PtinRect will only be used twice, so is it really worth
it? It is, because the other reason to use a function is clarity. In this case, it allows us to read statements beginning with IF PtInRect (...) in the code and understand what they do without the details of their definition confusing us. It's something like writing a report that requires a lot of citations and supporting data: You use footnotes to make these details accessible while tucking them out of the way so the reader can concentrate on the thrust of your argument. In general, the more the main loop of a program and as many of the major subroutines as possible look like English (which is referred to as “self-documenting” code), the better the structure of your program and the easier it will be to modify and use.

In [Init 3], we make sure that the MacBandit window limits are set to the proper size. Next, the payoff values for each fruit are set. This is done by assigning a payoff to payRate(n,1) of 1 coin if two fruits match. This value is increased to 2 coins if the two fruits that match are from the first three items: Macpot, cherry, or bell. For three matches, the payoff of 10 coins is assigned, unless the items are among the first three, which have a payoff of 20 coins. One more value is set, for the case of matching Macpot items. For these, the payoff for two matching—payRate(1,1)—is 4 coins, and the payoff for three matching—payRate(1,2)—is 40 coins.

Initializing, displaying, and saving graphic objects

Next, [Init 4] calls FormCoins [SR 19a], which checks to see if the file MacBandit.graphics is on the disk. Since no disk volume is specified, this file must be on the same disk BASIC was started from, or it won't be found. If the file is found, FormCoins opens it and gets the count of bytes-per-image. A FOR...NEXT loop handles the two coin views. We input the graphics commands for each coin view as a string and use a PICTURE statement to both display the coin as an animated attention-getter, and enable us to capture the image and put it into our coin array with a GET statement for later use. Note that a two-dimensional array is used, since we want to store and access more than one related image. (By the way, this technique is described in detail in Chapter 17, “Animation,” in another book of ours, Macinations, Microsoft Press, 1985.) We then erase the coin with PUT and go on. This technique of checking to see if a graphics file exists takes advantage of the MacInterface program (also in this book) to modify MacPaint files for incorporation into programs that use graphics, without the bother of keying in long strings of DATA statements. Once you've learned how to use MacInterface, you can then create your own coin symbols and generate appropriate DATA statements for drawing them, and incorporate the statements in MacBandit. (Perhaps you want a dollar slot machine?)
With the use of PUT and GET, we can produce animation effects when we have more than one view of a particular figure. Remember, when you PUT the same figure in the same location on the screen in the default XOR mode, you effectively erase it.

If the graphics file wasn't found, we go to the label ReadPictures [SR 19b], which first does a RESTORE to set the DATA pointer to the appropriate DATA statements for the ReadArray subprogram [SP 1] to first READ our two coin views, then draw them, and finally save them.

The program returns to [Init 4], and we are now ready to draw the remaining items that are needed to make this Bandit work: the slot machine and the fruit. The program first calls DrawScreen [SR 12], which in turn calls DrawMachine [SR 14].

**Drawing the slot machine**

DrawMachine draws all of the slot machine except for the fruit that will “spin” in the windows. This drawing is straightforward and is well commented in the listing, but we'd like to point out a handy technique used here: building proportions into the drawings by defining coordinates in terms of previously defined coordinates.

Consider the coin slot, for example. The bottom is defined in terms of the top \( (slotR(1) + 15) \), and the right side \( (slotR(4)) \) is defined as the coordinates of the left side \( (slotR(2)) \) plus twice the width of a coin (that is, \( coinSep \times 2 \)). This means that if you decide to move the slot down a bit, the bottom of the slot will remain the correct distance from the top. If you decide to make larger coins, the slot will expand to accommodate them! Things will still get out of kilter if you go too far, but building proportions into your graphics by defining relationships between variables makes the graphics much more flexible and intuitively understandable.

**Drawing and saving the fruit**

DrawItems [SR 15] is called next from DrawScreen [SR 12] and draws the Macpot jackpot bar and the fruit symbols — six items in all. Each is a simple figure, and for the lemon, orange, plum, and cherry, the built-in BASIC command, CIRCLE, makes the job easier. After returning to DrawScreen, the instruction: “Drag a coin to the deposit slot to play” is displayed on the screen.

After returning to [Init 4], SaveItems [SR 16] GETs the items into the fruit array. This is another example of making a bit of program code do double duty: First, the program draws the items to decorate the machine
and then gets them for use in the machine windows later. When designing games, it's a good idea to think of something useful to do with images that you draw, even if your main purpose is simply to GET them for future use.

Although we don't really save any space by using three separate subroutines to draw the machine and draw and get the symbols, the code is broken into three logical segments: draw fixed image (the machine); draw movable images (the fruit); and save movable images. This is an example of letting the logical structure of what you're doing determine how you segment your code.

**Setting up the MacBandit menu**

The next section of code [Init 4] sets up and enables the MacBandit menu. You will find that with most systems, creating a menu and checking for appropriate input is a tedious task that has to be repeated for each program or group of related functions within a program. As you can see in [Init 4], creating a menu on the Mac only takes one statement per menu item (plus one for the menu title). We create a pull-down menu at menu-bar position 6 with the title MacBandit, and include five items in the menu.

We enable the various items by putting a 1 as the last parameter in the MENU statement. This item will appear in black print when the menu is pulled down and will be “enabled”; that is, will be read as a valid user selection. Since Play Again isn't applicable when the player hasn't played at all yet, this item will appear as a “background” item (dimmed print) and won't be enabled the first time the game is played. In fact, the menu as a whole isn't quite ready to use yet: Menu event-trapping still needs to be activated. The next couple of lines of code do just that. An ON MENU GOSUB calls MenuEvent [ME 0] if the player makes selections from the MacBandit menu at any time.

**The main program loop**

This is a good time to glance back at our program outline to see what's on the agenda for the game (the outer WHILE...WEND loop, [Main 1]) and for the play of each coin (the inner WHILE...WEND loop, [Main 2]).

First, we set up (but don't display) a rectangle around the coin stack area so that we will have the reference points for placing our four coin stacks. A call to ShowStack [SR 13] sets the coin count to 0 and begins a FOR...NEXT loop that calls AddCoin [SR 10] to place each of the 20 coins that make up the player's starting supply. This subroutine uses the
MOD function to make sure that coins are distributed as evenly as possible among the stacks. Since the first and third stacks are closer to the bottom of the screen, we use another handy MOD to offset the second and fourth stacks by subtracting 10 from their y coordinates. (Because of the first MOD, think of the stacks as numbered 0, 1, 2, 3.) In all cases, coins are added by “erasing” the “top of coin” image with a PUT of the old image, and then a PUT of the new image, a coin-edge image stored in coin(1,2).

As we add coins, we have to watch for the situation where all the stacks are filled evenly with coins and a new coin is going to start a new “layer” of coins. We can tell that this is going to happen when coinCount (not counting the coin being added) is divided evenly by the number of stacks. When this happens, we jump to the AddEmptyStack label, which accommodates the new layer of coins.

Back in ShowStack, we call UpdateCount [SR 9] to print the starting totals of coins won in the last play and total coins owned by the player. Since we haven’t played yet, these totals are 0 and 20, respectively. Then control returns to the main program and the main playing loop, which handles the playing of the individual coins until the player either quits or runs out of money.

Main playing loop

After setting up a WHILE...WEND loop that makes sure the player hasn’t lost all the coins (WHILE coinCount <> 0), we then call DepositCoin [SR 1a]. This starts out with the variable deposited set to false, then uses its own WHILE...WEND loop to wait for a successful coin deposit to be made by the player. Since this requires a mouse drag, a nested WHILE...WEND loop waits for the player to hold the mouse button down.

When MOUSE(0) is less than or equal to −1, we know the mouse is being dragged. If the menu-option PayOffs is chosen, payOffFlag is set to true, and PayOffs [SR 3a] is called. PayOffs sets payOffFlag to false, and then displays the fruit symbols [SR 3b], [SR 3c] (by calling ShowFruit [SR 4]) and their appropriate pay rates. Remember that at the beginning of this program we dimensioned the payRate array to hold the various pay rates. By setting up that array of values and then referencing it using the index, it’s a simple matter to change the payoff schedule in [Init 3] without having to remember to change all of the occurrences of the values, which we’d have to do if each one had been given a specific value in different sections of the program.

The DIALOG expression enables the player to click a button to continue with the program after the payoff schedule has been read. This
technique makes the game much easier for users to play, since they can go at their own pace. It's easier for you as the programmer, too, because you don't have to code a timing loop based on your guess of the average player's speed-reading abilities. Instead, one set of instructions handles it all. After the user clicks the Continue button, the screen is redrawn with a call to DrawScreen [SR 12], which we discussed earlier.

If the PayOff menu selection was not made, the PtinRect function checks to see if the mouse activity is within the coin-stack area. If it is, we assume that the player wants to drag a coin to the slot machine (since it's the only game in town!). We then call SubCoin [SR 11], which manages the coin stacks much like AddCoin [SR 10], except that it takes a coin from the appropriate stack and reduces coinCount by 1.

Back in DepositCoin, we draw the coin over the (now hidden) arrow pointer. Another WHILE...WEND loop draws and redraws it while MOUSE(0) continues to be less than or equal to −1 (that is, while the player continues to drag it). Once the coin is "dropped"—the player has let go of the mouse button—we restore the pointer to normal and use our PtinRect function to see if the coin was released within the rectangle representing the deposit slot. If it wasn't, we call AddCoin [SR 10], which gives the effect of our Bandit indignantly snatching the coin away from the player and putting it back in the player coin stack. If the coin was properly deposited, the function sets the logical variable deposited to true, and we then fall through the bottom of DepositCoin's main WHILE...WEND loop, increase the value of totalDeposit by 1, and RETURN to the main playing loop.

The money is in the slot, so we're ready to give some action for the player's coin. After erasing the slot-machine windows [Main 2] (remember that a LINE statement with the $bf$ parameter defines and fills a rectangle), we give the appearance of the wheels spinning within the windows (by flashing different images) with a call at [Main 3] to SpinFruit [SR 2a]. The heart of SpinFruit is a pair of nested FOR...NEXT loops. The outer loop gives us spinCount spins; the inner loop cycles through the list of six symbols in rapid succession, drawing and then erasing the next image in each of the three windows.

What we're doing here is somewhat different from the operation of a real slot machine. A real machine spins the windows separately and uses a mechanism to vary the point at which each wheel stops spinning,
thus providing random symbols in the windows. We spin all the windows 
\textit{spinCount} times and then use another FOR...NEXT loop [SR 2b] to draw 
random symbols in the windows. (We also save the symbol numbers in the 
\textit{result} array so we can check for winning combinations.) This process is a 
lot easier than trying to simulate the mechanical process of an actual slot 
machine, and it gives about the same effect for game purposes.

Next, the program returns to [Main 3] and checks to see if we’ve 
got a winner. There’s a bit of clever coding here. The straightforward (and 
lengthy) way would be to use an IF...THEN test for each possible matching 
pair of symbols (does the result in window 1 equal the result in window 2; 
does window 2’s result equal the result in window 3; and so on) as well as 
checking for a three-match (does the result in window 1 equal the result in 
window 2 and the result in window 3). We would then need additional 
IF...THEN statements to branch to the appropriate subroutines for han­
dling a no-match, a two-match, and a three-match. Instead, we define a 
variable called \textit{same} that can have three possible values: 0 if no match at 
all, 1 for a two-match, and 2 for a three-match. We check results in win­
dows 1 and 2 and make \textit{same} equal to 1 if they match. If windows 2 and 3 
match, we add 1 to \textit{same}. (If 1 and 2 matched, \textit{same} is already 1, so it be­
comes 2, which makes sense, since if 1 and 2 match and 2 and 3 match, we 
have a three-match.) If nothing matched, \textit{same} remains at 0, due to the 
\textit{ELSE} clause in the first IF...THEN statement.

Now that we’ve got a numeric result, we use the ON...GOSUB 
structure to branch to the appropriate subroutine to handle the outcome 
found. If \textit{same} was 0 (nothing matched), the ON...GOSUB doesn’t get exe­
cuted. We then call UpdateCount [SR 9], which prints the message that no 
coins were won that play and subtracts one coin from the stack. The pro­
gram then returns to the top of the main playing loop at [Main 2] to play 
the next coin (if there is one).

If either two or three symbols matched, we have a winner. We 
then follow this general procedure: (1) Determine the number of coins 
won; (2) show a pile of coins representing the jackpot; (3) play some mu­
ic; and (4) stack the coins in the player’s pile. For a two-match we call 
MatchTwo [SR 5], which sets the value of \textit{prize} equal to the \textit{payRate} for 
the particular symbol. MatchTwo then calls PileCoins [SR 7], which draws 
a number of coins equal to the prize in random positions at the bottom of
the slot machine to simulate the coins pouring out. Next, Weasel [SR 18] is called to play "Pop Goes the Weasel," using a simple series of SOUND statements with appropriate values for the tune. Finally, UnPileCoins [SR 8] is called. This starts with a delay, so the player can gloat over the winnings for a moment, and then uses a FOR...NEXT loop to erase each coin (whose coordinates are in the pile array) and add it to the player's stack via a call to AddCoin [SR 10].

MatchThree [SR 6], which handles the big jackpots for three-matches, works the same way as MatchTwo. The prize is bigger, and if you are lucky enough to hit the "Macpot"—three Mac bars in a row—you also collect all of the coins deposited in the machine since the last Macpot (or the beginning of the game, if this is the first Macpot of the game). The other difference from MatchTwo is that the tune is played through a call to HappyDays [SR 17]. This produces a longer and louder tune (it uses the louder default volume of 127, while Weasel uses a volume of 20).

Having handled the jackpot, we call UpDateCount [SR 9] to summarize the good news in the player's totals, then we go back to the top of the main playing loop to play the next coin.

When the player runs out of coins, we fall into the bottom of the outermost (main game) loop [Main 4] and enable the Play Again menu item. We then loop until the player chooses Play Again (or Quit, which is already enabled). While we're here, let's look in a little more detail at the way menu events are handled.

Handling menu events

As mentioned earlier, we've enabled only the PayOffs, Quit to BASIC, and Quit to Desktop menu selections while coins are being played. If a menu selection is made, MenuEvent [ME 0] is automatically called, due to our use of the ON MENU GOSUB earlier. The MENU(1) function tells us which selection was made. The ON mItem GOSUB statement calls the appropriate subroutine according to which item is selected. PlayAgain [ME 1] sets the pAgainFlag to true, and deactivates the Play Again menu item (MENU 6,1,0) before it returns. The BasQuit [ME 3] and the DeskQuit [ME 4] subroutines each clean out the menu (MENU RESET) and then BasQuit ends the program, while DeskQuit exits to SYSTEM.

When at least one game (that is, one batch of player coins) has been played, the Play Again menu option is enabled. Selecting this option while we're in the loop following a game calls MenuEvent, which in turn calls PlayAgain because mItem is 1. PlayAgain sets pAgainFlag to true and
uses a MENU statement to disable Play Again. We do this because we want Play Again to be an active option only when a game has just been completed; trying to start a new game in the middle of an existing one would play havoc with the sequence of events and probably break the bank with a barrage of range errors. PlayAgain then returns to MenuEvent, which in turn returns to just after the post-game loop and thence to the top of the main game loop, where we set up the new game.

Suggestions for MacBandit

It would be easy for you to specify and check for your own winning combinations, determine their payoff, and choose a tune to serenade the winner. You could add a few more symbols to the symbol list (perhaps a nice McIntosh apple?). You can, of course, draw your own coins with MacPaint and use MacInterface to create the graphics file. You could animate the handle to simulate pulling it down. You could also rig the slot machine to be more like a Vegas slot (that is, give lower payoffs than the odds would justify) — but why be sadistic? After all, what could the Mac do with its accumulation of imaginary money?
MacBandit program listing

MacBandit
A game of almost chance

IF FRE(0) < 22000 THEN CLEAR, 22000
DEFINT A-Z
OPTION BASE 1
false = 0 : true = NOT false
fruitCount = 6
spinCount = 12
windCount = 3
windY = 108
stackNum = 4
coinSep = 32
stacksX = 300 : stacksY = 200
startCoins = 20
maxPileSize = 50
DIM fruit(100,fruitCount)  'pictures of different fruits
DIM result(windCount)  'results for each window
DIM payRate(fruitCount,2)  'rate at which fruit is paid off
DIM coin(95,2)  'contains 2 views of coin
DIM slotR(4)  'coin slot rectangle
DIM stR(4)  'coin Stacks Rectangle
DIM pile(maxPileSize,2)  'holds X and Y for payout coins
DIM penLoc(2)
RANDOMIZE TIMER  'return true if (x,y) is in rectangle (t,l,b,r)
DEF FN PtnRect(x,y,t,l,b,r) = (x >= l AND x <= r AND y >= t AND y <= b)
DEF FN Minimum(x,y) = -(x < y) * x + -(x > y) * y

WINDOW 1, "MacBandit", (2,40) - (508,337), 1

FOR n = 1 TO fruitCount
  payRate(n,1) = 1
  IF n < 4 THEN payRate(n,1) = 2
  payRate(n,2) = 10
  IF n < 4 THEN payRate(n,2) = 20
NEXT
payRate(1,1) = 4 : payRate(1,2) = 40  'MacPot pays off highest

GOSUB FormCoins  'form coin image arrays
GOSUB DrawScreen  'draw initial screen
GOSUB SaveItems  'get fruit pictures from screen
'install MacBandit menu
MENU 6, 0, 1, "MacBandit"
MENU 6, 1, 0, "Play Again"
MENU 6, 2, 1, "PayOffs"
MENU 6, 3, 0, "-
MENU 6, 4, 1, "Quit to BASIC"
MENU 6, 5, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON

** MAIN PROGRAM **
totalDeposit = 0
WHILE true
    stR(1) = stacksY - 10 : stR(3) = stacksY + 25
    stR(2) = stacksX : stR(4) = stacksX + stackNum * coinSep
    stackSize = startCoins : GOSUB ShowStack
    WHILE coinCount <> 0
        GOSUB DepositCoin
        LINE(55,105) - (95,138), 30, bf
        LINE(105,105) - (145,138), 30, bf
        LINE(155,105) - (195,138), 30, bf
        GOSUB SpinFruit
    WEND
    MENU 6, 1, 1
    pAgainFlag = false
    WHILE NOT pAgainFlag : WEND
    WEND
END

** MENU EVENTS **
MenuEvent:
    MENU 6, 0, 0
    mItem = MENU(1)
    ON mItem GOSUB PlayAgain, PayOffMenu, BasQuit, DeskQuit
    MENU 6, 0, 1
    RETURN

PlayAgain:
    ------------
    pAgainFlag = true
    MENU 6, 1, 0
    RETURN
PayOffMenu:                  user wants to see payoff
   payOffFlag = true
   RETURN

BasQuit:                   quit to BASIC
   MENU RESET
   END

DeskQuit:                  quit to Desktop
   MENU RESET
   SYSTEM

DepositCoin:               stay here until coin is deposited
   deposited = false
   WHILE NOT deposited         'loop until coin is deposited
      WHILE MOUSE(0) > -1     'wait until mouse button down
         IF payOffFlag THEN GOSUB PayOffs
            'PayOffs menu item chosen
      WEND
      x = MOUSE(1) : y = MOUSE(2)  'is button down in stack rectangle?
      IF NOT FN P.InRect(x, y, stR(1), stR(2), stR(3), stR(4)) THEN NotPick

       CALL HIDE_CURSOR: GOSUB SubCoin
      PUT(x - 15, y - 15), coin(1,1)  'draw new view
      WHILE MOUSE(0) <= -1         'loop until button is released
         IF MOUSE(1) = x AND MOUSE(2) = y THEN NotMoved
            'erase old view
            PUT(x - 15, y - 15), coin(1,1)
            x = MOUSE(1) : y = MOUSE(2)
            PUT(x - 15, y - 15), coin(1,1)  'draw new view
       WEND
       PUT(x - 15, y - 15), coin(1,1)  'erase old view
       CALL SHOW_CURSOR
         deposited = FN P.InRect(x, y, slotR(1), slotR(2), slotR(3), slotR(4))
      IF NOT deposited THEN GOSUB AddCoin
         'no, add back to stack

NotPick:
   WEND
   totalDeposit = totalDeposit + 1
   RETURN

SpinFruit:                  spin wheel
   FOR wind = 1 TO windCount : result(wind) = wind - 1 : NEXT
   FOR spin = 1 TO spinCount
      'spin wheel spinCount times
      FOR wind = 1 TO windCount
         'show something in each window
         result(wind) = (result(wind) MOD windCount) + 1
         x = 50 * wind + 10
         PUT(x,windY), fruit(1,result(wind))            'draw item
      NEXT wind
      SOUND 30, .7
      FOR wind = 1 TO windCount
         x = 50 * wind + 10
         PUT(x,windY), fruit(1,result(wind))            'erase item
      NEXT spin
      NEXT spin

FOR wind = 1 TO windCount 'place random items in each window
    x = 50 * wind + 10
    item = INT(fruitCount * RND + 1)
    PUT(x,windY), fruit(1,item)
    result(wind) = item
NEXT wind
RETURN

PayOffs: show user payoffs
payOffFlag = false
WINDOW 2, , (10,55) - (505,325), 2
PRINT SPACE$(48) + "Payoffs for MacBandit"
PRINT
FOR fruitNum = 1 TO 6
    FOR i = 1 TO 2 : GOSUB ShowFruit : NEXT i
    IF fruitNum MOD 3 = 0 THEN PRINT : PRINT
NEXT fruitNum
PRINT
FOR fruitNum = 1 TO 6
    FOR i = 1 TO 3 : GOSUB ShowFruit : NEXT i
    IF fruitNum MOD 3 = 0 THEN PRINT : PRINT
NEXT fruitNum
PRINT "If you get 3 Macs, you also get all the coins put in so far."
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1 : WINDOW CLOSE 2
GOSUB DrawScreen
RETURN

ShowFruit:
    CALL GETPEN(VARPTR(penLoc(1))))
    PUT (penLoc(2),penLoc(1) - 15), fruit(1,fruitNum)
    CALL MOVE(40,0)
RETURN

MatchTwo: 'two items matched on spin
    prize = payRate(type,1) 'type is type of item that matched
    GOSUB PileCoins
    GOSUB Weasel
    GOSUB UnPileCoins
RETURN

MatchThree: 'three items matched on spin
    prize = payRate(type,2) 'type is type of item that matched
    IF type = 1 THEN prize = prize + totalDeposit : totalDeposit = 0
    GOSUB PileCoins
    GOSUB HappyDays
    GOSUB UnPileCoins
RETURN
PileCoins:

```
FOR n = 1 TO FN Minimum(prize, maxPileSize)
    pile(n,1) = INT(140 * RND + 51)
    pile(n,2) = INT(40 * RND + 212)
    PUT(pile(n,1), pile(n,2)), coin(1,1)
NEXT
RETURN
```

UnPileCoins:

```
FOR delay = 1 TO 5000 : NEXT delay
FOR n = 1 TO FN Minimum(prize, maxPileSize)
    FOR delay = 1 TO 1000 : NEXT delay
    PUT(pile(n,1), pile(n,2)), coin(1,1)
    GOSUB AddCoin
NEXT
IF prize <= maxPileSize THEN UnPile1
    FOR n = 1 TO (prize - maxPileSize)
        GOSUB AddCoin
    NEXT
    UnPile1:
    RETURN
```

UpdateCount:

```
CALL MOVETO (stR(2), stR(3) + 20)
PRINT "Coins won = "; prize; "Total = "; coinCount; SPACE$(14)
prize = 0
RETURN
```

AddCoin:

```
stackX = (coinCount MOD stackNum)
stackY = stacksY - 5 * INT(coinCount / stackNum)
IF stackX MOD 2 = 1 THEN stackY = stackY - 10
IF INT(coinCount / stackNum) = 0 THEN AddEmptyStack
    PUT(stackX * coinSep + stacksX, stackY + 5), coin(1,1)
    PUT(stackX * coinSep + stacksX, stackY + 5), coin(1,2)
AddEmptyStack:
    PUT(stackX * coinSep + stacksX, stackY), coin(1,1)
    coinCount = coinCount + 1
    IF stackX = 0 THEN stR(1) = stR(1) - 5
    RETURN
```

SubCoin:

```
coinCount = coinCount - 1
stackX = (coinCount MOD stackNum)
stackY = stacksY - (5 * INT(coinCount / stackNum))
IF stackX MOD 2 = 1 THEN stackY = stackY - 10
    PUT(stackX * coinSep + stacksX, stackY), coin(1,1)
    IF INT(coinCount / stackNum) = 0 THEN SubEmptyStack
        PUT(stackX * coinSep + stacksX, stackY + 5), coin(1,2)
        PUT(stackX * coinSep + stacksX, stackY + 5), coin(1,1)
SubEmptyStack:
    IF stackX = 0 THEN stR(1) = stR(1) + 5
    RETURN
```
MAC BANDIT

DrawScreen: set up initial screen
CLS: GOSUB DrawMachine: GOSUB DrawItems
CALL TEXTFONT(0)
CALL MOVETO(10,20): PRINT "Drag a coin to the deposit slot to play."
RETURN

ShowStack: show stackSize number of coins
coinCount = 0
FOR n = 1 TO stackSize: GOSUB AddCoin: NEXT
GOSUB UpdateCount
RETURN

DrawMachine: draw machine except for fruit
slotR(1) = 80: slotR(2) = 55
slotR(3) = slotR(1) + 15: slotR(4) = slotR(2) + coinSep * 2
CALL TEXTSIZE(9): CALL TEXTFONT(1)
CALL MOVETO(slotR(4) + 8,slotR(1) + 12): PRINT "Deposit Coin"
CALL TEXTSIZE(12): CALL TEXTFONT(0)
CALL MOVETO(54,68): PRINT"$$"
CALL MOVETO(179,68): PRINT"$$
LINE(50,75) - (200,250), b 'main box
LINE(51,76) - (199,249), b 'make it thicker
CALL PENSIZE(2,2)
CALL FRAMERECT(VARPTR(slotR(1)))
CALL PENNORMAL

LINE(75,40) - (175,75), b: CIRCLE(125,53), 7 'box with rays, sun
LINE(125,75) - (75,40) - (175,75) - (75,75), b
LINE(125,75) - (175,75) - (125,75) - (175,40)
LINE(125,75) - (100,40) - (150,40)

LINE(50,50) - (75,75), b: LINE(175,50) - (200,75), b 'left, right $$ box
LINE(50,100) - (200,102), b: LINE(99,100) - (103,140), b 'l,l window
LINE(149,100) - (153,140), b: LINE(50,140) - (200,142), b 'r,b window
LINE(220,105) - (230,200), b: LINE(200,200) - (240,220), b 'arm
CIRCLE(225,90), 15 'handle
LINE(50,210) - (200,212), b: LINE(50,245) - (200,250), b 't,b of tray
RETURN

DrawItems: draw items on machine
CIRCLE(75,190), 12,...,7: CIRCLE(62,191), 1: CIRCLE(88,189), 1 'lemon
CIRCLE(125,190), 15,...,85: CIRCLE(113,195), 2 'orange
FOR n = 0 TO 10: CIRCLE(175,190), n,...,8: NEXT
LINE(162,185) - (175,190)
CALL MOVETO(60,163): PRINT"MAC"
CALL TEXTFONT(1)
LINE(58,150) - (90,165), b 'jackpot bar
CIRCLE(120,155), 5 : CIRCLE(125,165), 5 : CIRCLE(135,160), 5 'cherry
LINE(124,152)-(140,145) : LINE(128,160)-(140,145)
LINE(137,155)-(140,145)
CIRCLE(175,148), 2 'bell
LINE(170,150)-(180,150) : LINE(170,150)-(167,155)
LINE(180,150)-(183,155) : LINE(183,155)-(183,165)
LINE(167,155)-(167,165) : LINE(183,165)-(188,170)
LINE(167,165)-(162,170) : LINE(162,170)-(188,170)
RETURN

SaveItems: save images from screen
GET(58,146) - (90,165), fruit(1,1) 'Motor
GET(115,145) - (140,170), fruit(1,2) 'cherry
GET(162,146) - (190,170), fruit(1,3) 'bell
GET(60,180) - (90,200), fruit(1,4) 'lemon
GET(110,175) - (140,205), fruit(1,5) 'orange
GET(160,180) - (185,200), fruit(1,6) 'plum
RETURN

HappyDays: play 'Happy Days Are Here Again'
SOUND 262,6 : SOUND 330,3 : SOUND 330,9 : SOUND 392,6 'ceceg
SOUND 392,6 : SOUND 523,3 : SOUND 523,9 : SOUND 659,6 'gcece
SOUND 659,6 : SOUND 523,6 : SOUND 523,6 : SOUND 392,6 'ecce
SOUND 392,6 : SOUND 330,3 : SOUND 330,6
RETURN

Weasel: play 'Pop Goes the Weasel'
SOUND 523,4,20 : SOUND 523,2,20 : SOUND 587,4,20 'cdcd
SOUND 587,2,20 : SOUND 659,2,20 : SOUND 784,2,20 'deg
SOUND 659,2,20 : SOUND 523,4,20 'ec
RETURN

FormCoins: get coin images
'if file MacBandit.graphics exists, then read it, else read from DATA
fileHere = true ON ERROR will change this to false if file isn't here
ON ERROR GOTO FileError set up error trapping
OPEN "MacBandit.graphics" FOR INPUT AS #1 is file present?
ON ERROR GOTO 0 disable error trapping
IF NOT fileHere THEN ReadPictures no, read from DATA

FOR n = 1 TO 2 'get 2 coin images from file
INPUT #1, bytes 'get byte count for image
image$ = INPUT$(bytes,1) 'input it into string
noUse$ = INPUT$(1,1) 'read past end of line marker
PICTURE (0,0), image$ 'display image
GET (0,0) - (32,30), coin(1,n) 'transfer bit image into array
PUT (0,0), coin(1,n) 'erase it from window
NEXT
CLS
CLOSE 1
GOTO FormPicEnd
ReadPictures: 'MacBandit.Graphics isn't on disk; read from DATA
  RESTORE SlantHeadData: CALL ReadArray(coin(),1)
  RESTORE EdgeData: CALL ReadArray(coin(),2)
  FormPictEnd:
  RETURN

FileError: return fileHere = false if file not found
  IF ERR <> 53 THEN ON ERROR GOTO 0
  RESUME NEXT

SUB ReadArray(array(2),index) STATIC ' _______ read given array from DATA
  READ size
  FOR n = LBOUND(array) TO LBOUND(array) + size -1
    READ num : array(n,index) = num
  NEXT
END SUB

SlantHeadData: ' __________ data for picture of slanted coin
  DATA 95
  DATA 33, 31
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 15, -1024, 0, 240, 960, 0, 1795, -4040, 0
  DATA 2124, 3204, 0, 12832, -3565, 0, 8226, -23551, 0, 16433
  DATA 3072, -32768, 22222, -31388, -32768, 16388, 26648, -32768, 8249, 1793
  DATA 0, 12352, -16253, 0, 10240, 5, 0, 5888, 58, 0
  DATA 2288, 964, 0, 1807, -968, 0, 240, 960, 0, 15
  DATA -1024, 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0

EdgeData: ' __________________________ data for picture of edge of coin
  DATA 95
  DATA 33, 31
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, -32768, 16384, 0, -32768, 16384, 0, -32768, 8192, 1
  DATA 0, 12288, 3, 0, 10240, 5, 0, 5888, 58, 0
  DATA 2288, 964, 0, 1807, -968, 0, 240, 960, 0, 15
  DATA -1024, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
  DATA 0, 0, 0
MacMouse is an ideal program for introducing young children to the Mac. When you think about it, many of the things that appeal to us about the Mac—its easy-to-use menus, buttons, and windows, the visual richness of its graphics capabilities, and the gratifyingly tactile response of the mouse—are all things that reach the child in us. Children want to see, touch,
manipulate, and explore the world around them, whether they’re hiking in the woods or sitting in front of a computer. They have even less patience with unfriendly machines than we do. With the Mac and Macintosh BASIC, we can write programs that let kids play directly with the images on the screen. As kids play with MacMouse, they’ll learn how to use the mouse to start an exciting mouse race. As the mice race neck and neck down the track, kids can cheer “their” mouse on, then watch a photo finish. We’ll also suggest some ways that you can modify this program to make it interesting and useful to older kids.

How to play MacMouse

When MacMouse starts running, the screen shown in Figure 1 appears. A “banner” across the top of the screen tells you that you’ve arrived at the MAC & TOSH RACEWAY. Instructions just below tell how to start the race. On the left side of the screen, behind the starting gate, are two mice. The black one is Mac, and his opponent, the white one, is called (surprise!) Tosh. The finish line is on the right side of the screen. Each mouse’s total wins are tallied in the big box or “scoreboard” at the top of the screen. Right now, of course, the score is tied at 0-0.

FIGURE 1. The MacMouse title and playing screen
To start the race, simply click the mouse button (it doesn’t matter where the mouse pointer is, as long as it’s on the screen). You’ll hear a familiar bugle call (you can yell “They’re off!” if you wish). The mice will appear to run across the screen, their little legs pumping away. Most of the time they will run “neck and neck,” with one seldom getting more than a head in front of the other. When a mouse reaches the finish line, the message “[Mouse’s name] wins!” appears in the center of the scoreboard. You’ll hear a triumphant refrain: Each mouse has its own victory tune, a snippet of notes from “Pop goes the weasel.” The finish will usually be a close one, since random numbers are used to determine how far each mouse moves with each “stride,” and random numbers tend to average out over time. Occasionally, the race will be a tie, and a message to that effect will appear on the scoreboard.

After each race, 1 is added to the winning mouse’s score on the scoreboard. MacMouse will then set up the racetrack again, with the mice back at the starting gate, and wait for you to click the mouse button to start a new race. To stop MacMouse and return to BASIC, pull down the MacMouse menu and choose Quit to BASIC or Quit to DeskTop.

There aren’t any options or strategies for playing MacMouse, but after you’ve read our discussion of the code and tinkered with the program a bit, you’ll probably think of a number of things that could be added to make the game more interesting. We’ll suggest some fairly elaborate projects using MacMouse at the end of the chapter.

How the program works

The program outline in Figure 2 shows the main events that take place in MacMouse and the way they are structured. The main program consists of two WHILE...WEND loops: an outer one to set up each new race and start it when the mouse is clicked, and an inner one to move the mice until one mouse crosses the finish line. When that happens, the outer loop takes over again to proclaim the winner and update the score.

<table>
<thead>
<tr>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimension arrays for images</td>
</tr>
<tr>
<td>set constants for mouse size and mouse paths</td>
</tr>
<tr>
<td>set window size</td>
</tr>
<tr>
<td>install MacMouse menu</td>
</tr>
<tr>
<td>draw and store mouse pictures (MakeMice)</td>
</tr>
</tbody>
</table>

 FIGURE 2. Program outline
Main Program

WHILE true
    set mouse starting positions
    draw raceway banner (DrawBanner)
    draw start and finish lines
    draw two mice in starting positions
    wait for mouse button to be clicked
    start race: make bugle sound and erase starting line (StartSound)

WHILE neither mouse has reached or passed finish line:
    calculate how far each mouse will go this time
    erase old mice
    draw mice in new positions
    switch to alternate view for animation
WEND (repeat until a mouse reaches finish line)

test for winner
    if x positions of mice are equal, race is tie: print message
    if not, play appropriate victory tune, update score, display message
WEND

Menu Events

MenuEvent: transfer control to appropriate subroutine
    BasQuit: quit to BASIC
    DeskQuit: quit to Mac desktop

Subroutines

DrawBanner: draw raceway banner
    StartSound: bugle call to start race
    MacWinSound: Mac's victory tune
    ToshWinSound: Tosh's victory tune

MakeMice:
    enable error trap if file not found
    open graphic file
    if file not found
        read images from DATA statements (ReadPictures)
    if file found
        get both views of Mac and save to array

PicToClip:
    convert image array to DATA statements
    put DATA statements into Clipboard for pasting at end of program

Subprogram

ReadMArray: read image array from DATA statements

Figure 2. Program outline (continued)
Dimension arrays and set up constants

In [Init 1], the random-number generator is “seeded” with a random number from the timer so that a new series of numbers will be obtained when RND is called. Only simple arithmetic with integers will be needed for this game, so we DEFINT A-Z. The arrays mac and tosh will each hold two views of their respective mice; the two images will be alternated on the screen to produce a rudimentary animation that makes the legs of the mice appear to move as they run.

The constant variables are defined next: false is given the value zero, and true is defined as NOT false. These are set this way because Microsoft BASIC lets any nonzero value—positive or negative—be interpreted as “true” in any conditional statement. Two other constants to note are macY and toshY, which specify the y coordinates of our two mice. The y coordinates can be specified as constants because the mice race in a straight line across the screen, each on its own “track,” so only their x coordinates change during the course of the race. The constant finish will be used both in drawing the finish line and in testing to see if either mouse has won the race.

The line of code at [Init 2] ensures that the MacMouse window is the right size. Next, we install the MacMouse menu. This menu has only two options, Quit to BASIC, and Quit to DeskTop. The menu handler is coded at [ME 0] and will be called automatically by menu events.

The call to MakeMice [SR 5] sets the stage for some interesting code. The object of MakeMice is to generate the mouse picture-arrays. It accomplishes this by reading the mouse pictures from the disk-file MacMouse.graphics. A FOR...NEXT loop is used for each mouse to INPUT, PICTURE, GET, and PUT (erase) the two pictures of that mouse [SR 5a, SR 5b]. If the disk file is not available, then control goes to the label ReadPictures [SR 5c] which calls the subprogram ReadMArray [SP 1] for each of the two views of the mice. ReadMArray makes use of the appropriate set of data statements that describe the mice images: Mac1Data [Data 1], Mac2Data [Data 2], Tosh1Data [Data 3], and Tosh2Data [Data 4]. MacMice then continues with the same processing after the [Init] section that it would have done had there been a MacMouse.graphics file.

Main program

The main program is an endless loop. Races will be run until the user decides to quit by pulling down the MacMouse menu and choosing Quit to
BASIC or Quit to DeskTop. To get ready for a new race, the first lines in the main loop [Main 1] set the x positions of the two mice (macX and toshX) to 10. (Remember that the y positions are constant and don't ever have to be changed.) [Main 1] calls DrawBanner [SR 1] to display all the text needed for the racecourse. This includes the banner message “MAC & TOSH RACEWAY,” the instruction for starting the race by pressing the mouse button, and the total wins for Mac and Tosh. Next, the starting gate and finish line are also labeled, and the name of each mouse is placed beneath it at the starting line. Finally, a LINE statement with the box option is used to frame the scoreboard.

After control returns to the main program loop, the finish line is drawn, using the constant finish for the x coordinate. Next, the starting line is drawn at an x coordinate of macX + mouseLen + 2: The mouse starting position, plus the mouse length, plus 2. The extra two pixels make sure the mouse's nose is not quite touching the starting line (we wouldn't want any cheating, would we?). The same y coordinates are used for the finish and starting lines. Using mouseLen rather than a specific number for placing the starting line makes it easy to change the size of the mouse (or substitute another creature of different size) and still place the starting line correctly. You'll appreciate this when you write an elephant-race program (which would be very easy if you use Maclnterface, another program in this book, to incorporate the elephants that you might draw in MacPaint).

The first of the two images for each mouse is drawn in its starting position just behind the starting line using PUT statements. Then a WHILE...WEND loop is used to wait for the mouse button to be pressed.

Once the button is pressed, it's time to start the race. StartSound [SR 2] is called at [Main 2] to play a traditional bugle call. The subroutine ends with a brief delay loop to add suspense.

Next, the starting line is erased so it won't be visible through the white mouse as the mouse is drawn over it. Note that the last parameter in the LINE statement, the color, is given as 0. Microsoft BASIC treats 33 as black and 30 as white, as noted in the manual, but it turns out that any even color number (including zero) comes out as white, while any odd number comes out as black. We've used 0 here to draw a white line over the black line to erase it. One thing to look out for if you run this program with later versions of BASIC is whether the color numbers have been redefined.
The WHILE...WEND loop at [Main 3] moves the mice until one crosses the finish line. Note that the actual condition is, "While the x coordinate of each mouse plus its length is less than the x coordinate of the finish line, keep moving the mice."

The first item inside this loop is a short delay loop. Next, the distance in pixels that Mac will move (macInc) and the distance for Tosh (toshInc) are determined. These increments are assigned a random number between 1 and 7. The mice are then erased from their old position with PUT statements specifying the "old" coordinates: that is, the x coordinates without the increments.

Next, the program draws the mice in their new positions, which have as their x coordinates macX + macInc for Mac, and toshX + toshInc for Tosh. The y coordinates do not change. The new x coordinates are re­saved in macX and toshX, respectively. These new x coordinates are used both as the starting point for the next move, and in the next move so the mice in these positions can then be erased.

The value of view must now be changed so that the next mouse drawn will use the alternate image. Since view should have a value of either 1 or 2 (recall that there are two images for each mouse), a neat little trick can be used to do this: view is assigned the value of 3 minus its current value. If view is 1, this gives 3 – 1, or 2, while if view is 2, the result is 3 – 2, or 1. An alternative way to do this might be:

\[
\text{IF view = 1 THEN view = 2 ELSE view = 1}
\]

This might be easier for you to understand than the way we do it in MacMouse. The method in the program results in more compact code, but it does take a moment or two to figure out. Which do you prefer?

Control now goes back to the top of the mouse-moving loop, and the test for the end of the race is made again. When a mouse crosses the finish line, control falls through the bottom of the loop to [Main 4].

**In the winner's circle**

The first thing to check for is whether there really is a winner and not a tie. At this point, we know that at least one mouse (but possibly both) has reached or crossed the finish line. The first IF statement determines whether the x coordinates of the two mice are the same. Since both mice
move each time through the loop, it's possible, though not too likely, that they'll end up in a dead heat. Note that both mice can cross the finish line during the same “turn” but with one ahead of the other: This isn't a tie, but a win for the mouse with the greater x coordinate. If the x coordinates are not equal, control goes to NotTie [Main 5], where the winner will be determined. If they are equal, it's a tie, so a message to that effect is printed and control jumps past the various “win” routines to the label PastWin.

If the race is not a tie, the IF...THEN statement in [Main 5] determines the winner. If macX is less than toshX, then Tosh won and control is passed to ToshWins [Main 6]. Otherwise, Mac won and the code following the IF...THEN statement is executed. The blocks of code for Mac winning and for Tosh winning do exactly the same things: (1) add one to the winning mouse's score, (2) play the tune for the winning mouse (handled by MacWinSound [SR 3] or ToshWinSound [SR 4]), and (3) print the winning message with the mouse's name. The tunes and messages are different for the two mice, but the order of events is the same, and in either case control ends up at PastWin. If you use a structure like this in your own programs, don't forget the GOTO at the end of the first block of code (in [Main 4]). If you leave it out, both of the alternative paths will be taken.

After a moderately long delay, control returns to the top of the outer loop, which is, as previously noted, an infinite loop. The racetrack is redrawn, and the next race will start when the mouse button is pressed.

**Suggestions for MacMouse**

You can start tinkering with this program by adding small embellishments or “bells and whistles.” These might include a flag that waves in the breeze and a “winner’s circle,” placed just past the finish line, for the winning mouse to enter.

A more elaborate (and perhaps adult) version of the game could be created by making it possible for several players to bet (in imaginary money) on the outcome of the race. An array could store the players' current money (they might all start with, say, $100 and bet $5 per race), and the money would be updated after the race. Alternatively, you could let players bet any amount they wish up to their total cash, and store the amounts bet in another array.

To make the race more realistic, you could change the odds of a given mouse winning from the present 50-50 by giving that mouse a larger or smaller range of random numbers for its increment. Bets could pay off
according to these odds, which would be posted before the race begins. If you go in that direction, consider adding some more mice (perhaps a total of four). You could use MacPaint to create four distinctive images (plus alternates) for the mice, then put them in a graphics file using the MacInterface program (see that chapter for details).

You might also use MacPaint and MacInterface to create more than two images for each mouse. Flickering back and forth between two images of the same object provides a sense of movement, but movement will be smoother and truer to life if more images are used.

Finally, you could take an entirely different tack and make the game an educational quiz for kids. To do that, you would need to either write a separate program or include in this one an option to have the parent type in questions and answers, which would be stored in a string array quiz. This array would be saved to disk as a file and opened when the child ran the game. During a race, one mouse (say Tosh) would move a random distance each “turn” as at present, but the child’s mouse, Mac, would move only when a question was answered correctly. Since the child probably won’t get all the answers right, the mouse should probably move up to twice as far as the average move of the computer mouse when a question is answered correctly.
MacMouse program listing

```basic
INIT

RANDOMIZE TIMER
OPTION BASE 1
DEFINT A-Z
DIM mac(100,2), tosh(100,2)
false = 0 : true = NOT false
mouseLen = 48
macY = 141 : toshY = 191
finish = 425

WINDOW 1, "MacMouse", (4,30) - (508,325), 4

MENU 6, 0, 1, "MacMouse"
MENU 6, 1, 1, "Quit to BASIC
MENU 6, 2, 1, "Quit to DeskTop"
ON MENU GOSUB MenuEvent : MENU ON

GOSUB MakeMice

MAIN

WHILE true
  loop once per race, until quit by menu event
  'starting positions
    macX = 10 : toshX = 10

GOSUB DrawBanner
  'draw raceway banner
LINE (finish,125) - (finish,300)
  'finish and starting lines
LINE (macX + mouseLen + 2, 125) - (macX + mouseLen + 2, 300)
view = 1
PUT (macX,macY), mac(1,view)  
  'show mice sitting and waiting
PUT (toshX,toshY), tosh(1,view)
WHILE MOUSE(0) > -1 : WEND
  'wait for button press

  ____________________________ and they're off!
GOSUB StartSound
  'erase start line
LINE (macX + mouseLen + 2, 125) - (macX + mouseLen + 2, 300), 0

  ____________________________ loop, moving mice until one crosses line
WHILE macX + mouseLen < finish AND toshX + mouseLen < finish
  calculate how far they
FOR t = 1 TO 50 : NEXT
  will move this time
  macInc = 7 * RND(1)
  toshInc = 7 * RND(1)
```

MacMouse
A mouserace featuring Mac and Tosh
MAC MOUSE

PUT (macX, macY), mac(1, view)  'erase old Mac
PUT (toshX, toshY), tosh(1, view)  'and Tosh images
PUT (macX + macInc, macY), mac(1, 3 - view)  'draw new images
PUT (toshX + toshInc, 191), tosh(1, 3 - view)

macX = macX + macInc

toshX = toshX + toshInc

view = 3 - view  'flip view to other state

WEND

we have a winner!

CALL TEXTFONT(2) : CALL TEXTSIZE(24)
CALL MOVETO(180, 110)
IF macX <> toshX THEN NotTie
PRINT "Tie Race!"
GOTO PastWin

NotTie:
IF macX < toshX THEN ToshWins
macScore = macScore + 1  'Mac wins
GOSUB MacWinSound
PRINT "Mac Wins!"
GOTO PastWin

ToshWins:
toshScore = toshScore + 1  'Tosh wins
GOSUB ToshWinSound
PRINT "Tosh Wins!"

PastWin:
FOR t = 1 TO 10000 : NEXT
WEND  'loop back to race again

***************************************************************************** MENU EVENTS

MenuEvent:
MENU 6, 0, 0
item = MENU(1)
ON item GOSUB BasQuit, DeskQuit
MENU 6, 0, 1
RETURN

BasQuit:  ' ______________________________________________ quit to BASIC
         MENU RESET
         END

DeskQuit:  ' ______________________________________________ quit to Mac desktop
         MENU RESET
         SYSTEM

***************************************************************************** SUBROUTINES

DrawBanner:  ' ______________________________________________ draw raceway banner
CLS : CALL TEXTFONT(0)  'Chicago font
CALL TEXTSIZE(24)
CALL MOVETO(100, 28) : PRINT "MAC & TOSH RACEWAY"
CALL TEXTSIZE(12)
CALL MOVETO(120,50) : PRINT "Press Mouse Button to Start Race"
CALL MOVETO(40,80) : PRINT "Mac's Total Wins:"; macScore
CALL MOVETO(320,80) : PRINT "Tosh's Total Wins:"; toshScore
CALL MOVETO(35,105) : PRINT "Starting Gate"
CALL MOVETO(400,105) : PRINT "Finish Line"
CALL MOVETO(5,175) : PRINT "Mac"
CALL MOVETO(5,225) : PRINT "Tosh"
LINE (20,35) - (480,120),, b
"frame scoreboard
RETURN

StartSound: __________________________ bugle call starts race
SOUND 392,2 : SOUND 523,2 : SOUND 659,2
SOUND 784,4 : SOUND 659,2 : SOUND 784,6
FOR t = 1 TO 600 : NEXT
"wait for bugle call to finish
RETURN

MacWinSound: ________________________ play Mac's victory tune
SOUND 523,2 : SOUND 587,2 : SOUND 659,2
SOUND 587,4 : SOUND 659,2 : SOUND 523,2
RETURN

ToshWinSound: ________________________ play Tosh's victory tune
SOUND 523,2 : SOUND 0,2 : SOUND 523,2
SOUND 587,2 : SOUND 0,2 : SOUND 587,2
SOUND 659,2 : SOUND 784,2 : SOUND 659,2
SOUND 523,2
RETURN

MakeMice: __________________________ make mouse picture arrays
"if file MacMouse.Graphics exists, read images,
'else read from data statements
fileHere = true
'ON ERROR will change it to false if not found
ON ERROR GOTO FileError
'is file present?
OPEN "xMacMouse.graphics" FOR INPUT AS #1
'enable error trapping
ON ERROR GOTO 0
'no, read from data
IF NOT fileHere THEN ReadPictures

FOR view = 1 TO 2
INPUT #1, bytes
image$ = INPUT$(bytes,1)
noUse$ = INPUT$(1,1)
PICTURE (0,0), image$
GET (0,0) - (mouseLen,19), mac(1)view
PUT (0,0) - (mouseLen,19), mac(1)view
NEXT

FOR view = 1 TO 2
INPUT #1, bytes
image$ = INPUT$(bytes,1)
noUse$ = INPUT$(1,1)
PICTURE (0,0), image$
GET (0,0) - (mouseLen,19), tosh(1,view)  'save Tosh in array
PUT (0,0) - (mouseLen,19), tosh(1,view)  'erase it
NEXT view
image$ = ""
CLOSE 1
RETURN

ReadPictures:  'MacMouse.Graphics isn't on disk: read from data
RESTORE Mac1Data : CALL ReadMArray(macro(),1)
RESTORE Mac2Data : CALL ReadMArray(macro(),2)
RESTORE Tosh1Data : CALL ReadMArray(tosh(),1)
RESTORE Tosh2Data : CALL ReadMArray(tosh(),2)
RETURN

FileError:  returns fileHere = false, if error is "File not Found"
IF ERR <> 53 THEN ON ERROR GOTO 0  'unknown error, print and halt
fileHere = false
RESUME NEXT

PicToClip:  generate DATA statements in Clipboard
OPEN "Clip:Text" FOR OUTPUT AS #20
na$ = "Tosh1"
PRINT#20, na$; "Data: "; STRING$('_',33);" data for picture "; na$
low = LBOUND(tosh,1) : high = UBOUND(tosh,1)
PRINT #20, "Data": high - low + 1
PRINT #20, "Data": tosh(low,1); "; "; tosh(low + 1, 1)
FOR n = low + 2 TO high
  IF (n - (low + 2)) MOD 10 <> 0 THEN GOSUB NewLn ELSE PRINT #20, "; ";
temp$ = STR$(tosh(n,1))
  IF LEFT$(temp$,1) = " " THEN temp$ = RIGHT$(temp$,LEN(temp$) - 1)
  PRINT#20, temp$;
NEXT n
PRINT #20,"
CLOSE 20
STOP

NewLn:  start new line, print DATA with indent
PRINT #20," ; PRINT #20," Data ";
RETURN

SUB ReadMArray(array%(), index%): Read size
READ size
FOR n = LBOUND(array%,1) TO LBOUND(array%,1) + size - 1
  READ num : array%(n,index%) = num
NEXT
END SUB

Mac1Data:  data for picture Mac1
DATA 100
DATA 49,20
DATA 0,0,0,0,4096,0,896,0,8192,0
DATA 1984, 0, 8192, 2040, 2032, 0, 8192, 16383, 1016, 0
DATA 12288, -1, -15368, 0, 6145, -1, -12304, 0, 3075, -1
DATA -30, 0, 1539, -1, -76, 0, 775, -1, -5, 0
DATA 511, -1, -4, 0, 255, -1, -2, 0, 127, -1
DATA -1008, 0, 3, -18429, -29152, 0, 3, 12291, 1536, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0

Mac2Data:  """" data for picture Mac2
DATA 100
DATA 49, 20
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 3968, 0, 0, 60, 4064, 0, 0, 126, 2032, 0
DATA 0, 511, -30736, 0, 0, 1023, -24608, 0, 0, 2047
DATA 63, -1, -8, 0, 487, -1, -4, 0, 312, -1
DATA -2016, 0, 0, 1911, 7232, 0, 0, 1638, 3072, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0

Tosh1Data:  """" data for picture Tosh1
DATA 100
DATA 49, 20
DATA 0, 0, 0, 0, 2048, 0, 448, 0, 4096, 0
DATA 544, 0, 4096, 1020, 568, 0, 4096, 7171, -32444, 0
DATA 6144, 24576, 24900, 0, 3072, -32768, 5688, 0, 1537, 0
DATA 2065, 0, 769, 0, 42, 0, 386, 0, 5, -32768
DATA 252, 0, 2, 0, 64, 0, 511, 0, 62, 9214
DATA 14856, 0, 1, 21505, 17680, 0, 1, -27623, -32000, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

Tosh2Data:  """" data for picture Tosh2
DATA 100
DATA 49, 20
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 2176, 0, 0, 14, 2272, 0, 0, 57, 1296, 0
DATA 0, 193, -31472, 0, 0, 256, 22752, 0, 0, 512
DATA 8260, 0, 0, 512, 168, 0, 2040, 1024, 22, 0
DATA 1055, -2048, 8, 0, 7, -32768, 2044, 0, 1, -954
DATA -6112, 0, 0, 681, 5184, 0, 0, 816, 3072, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0
MacBall

Ball games of various sorts have probably been played since the very dawn of civilization. From the Mayas of Central America to bored pages in medieval castles to descendants of Abner Doubleday who glue themselves to the TV for the World Series, hitting something with a stick or racquet and getting it to go where you want it to has provided continuing fascination.
MacBall lets you bounce a video "ball" off a mouse-controlled "paddle" and guide it into a collection of video apples on the screen. It works much like the popular games Breakout and Pong.

While you may have bought your Mac for serious purposes like spreadsheeting and producing high-quality business charts, you will find that MacBall reveals some interesting programming techniques (as well as tests your hand-eye coordination!). Among the programming techniques you will discover are paddle-movement control, animation, and how to make an object appear to "bounce." So limber up your wrists, put a little English on your mouse moves, and play MacBall!

How to play MacBall

MacBall starts by giving you the title screen shown below in Figure 1. The next screen, the playing screen shown in Figure 2, presents you with the various elements of the game.

FIGURE 1. The MacBall title screen
On the left side of the playing screen is your paddle (or racquet, if you prefer). On the right side are three columns of nine apples each (McIntoshes, we assume). When you start to play, a little ball whizzes out from in front of the apples. The object of the game is to keep batting the ball back across the screen with the paddle. When the ball hits an apple it bounces back, the apple disappears, and you score a point. The goal is to knock out all the apples with the ball while trying not to let the ball get past your paddle to the left and out of play. Your score (maximum possible is 32,767) is listed at the bottom of the screen, together with the number of balls remaining to be played (not counting the current one), and a ball-speed indicator/control, which we'll discuss shortly. The MacBall menu contains these choices:
The program contains a demonstration which you can run by choosing See Demo from the MacBall menu. When you select demo mode and then move the mouse forward and back, the paddle moves up and down along the left side of the screen. Next, a ball is served from the right side of the screen. When the ball comes, you can hit it by moving your paddle into its path. If you miss and the ball sails past your paddle and hits the left side of the screen, there’s no harm done since this is a demo: The ball will simply bounce off the left wall and go its merry way, committing more mayhem in the apple orchard. (In a normal game, you’d lose one ball if you missed.) The demo game will run, all by itself, with no effort from you.

When you do hit the ball, it bounces off the paddle at an angle that depends on how squarely you hit it. Figure 3 illustrates the angles the ball will take after hitting the paddle at different points on its surface.

The real fun begins when you’ve cleared a path to the far-right column of apples. When a ball hits one of those apples, it’s liable to bounce off, hit an apple in one of the front columns, bounce off to the back wall, then bounce back to another apple, then... well, you get the idea. If you

---

**Figure 3.** Rebound angles from different points on the paddle
have the skill (or the luck) to break through the apple barrier while there are lots of apples left, you can rack up an impressive score. At certain angles you can go through several apples before the ball can break free. When you’ve knocked out all the apples, a new supply of 27 is provided.

You can stop the demo game by choosing Quit to BASIC from the MacBall menu. To actually play a game, choose Start from the Run menu and then Play Game from the MacBall menu. When the title screen reappears, choose Play Game from the MacBall menu. Now when you play and miss a ball with the paddle, the ball goes out of play and one of the little balls in the supply shown below the playing screen disappears. Also, if you don’t pick a speed to start with, the next ball served up by your Mac travels a little faster than the last. You’ll find that each “serve” is lobbed from a different place. Serves can range from a flat drive to a ball that comes flying down at a steep angle toward the lower left side of the screen.

For a more strenuous game, click the mouse pointer in the long “speed bar” below the playing screen before you start your next game. You’ll see a little indicator appear where you clicked. The closer to the right end of the bar you set the indicator, the faster the balls will travel. By using the indicator, you can “handicap” yourself.

You’ve probably figured out that knocking out a row of apples so that you can hit a ball behind the apples and off the back wall is the best way to rack up a high score in MacBall. Aside from that strategic consideration, becoming a pro at MacBall mostly involves being able to control where the ball goes so you can knock out a column of apples or pick off the odd apple or two when most of the apples are gone.

**How MacBall works**

MacBall is a good example of a simple program structure. At the beginning, it performs a number of initialization steps. It draws the apple, ball, and paddle pictures and saves them in arrays for future use. It installs the MacBall menu, draws a title screen, shows a screen of instructions, then draws the playing screen. After setting up these initial parameters, it goes to the main game loop. The program is outlined in Figure 4, which you should refer to while reading the following sections.
**Initialization**

- Dimension and initialize variables, constants, and patterns
  
- Draw and save pictures for future use
  
  *(FormApple, FormBall, FormPaddle)*
  
- Install MacBall menu
  
- Show title, instruction, and playing screens
  
  *(TitleScreen, Instructions, DrawScoreArea)*

**Main Program**

WHILE true (interrupted by menu event choice to quit)

- Enable Play Game and See Demo items in MacBall menu
  
- Set up ball, paddle and apples for new game *(StartGame)*
  
- Disable Play Game and See Demo menu items until end of game
  
  WHILE there are balls left to play
    
    - Give player a chance to move paddle,
      
      - Then start a new ball in a random direction *(NewBall)*
      
      - Move paddle, ball, check for apple hit or ball loss *(PlayBall)*
      
      - If ball speed not user-set, increase for next ball *(IncSpeed)*

  WEND

WEND

**Menu Events**

MenuEvent: branch to appropriate subroutine

- GameMenu: reset and play game
  
- DemoMenu: play demo game
  
- BasQuit: quit to BASIC
  
- DeskQuit: quit to Mac Desktop

**Subroutines**

- **StartGame:**
  
  - Set up initial screen
  
  - Draw new speed bar *(DrawSpeed)*
  
  - Set up orchard *(DrawApples)*
  
  - While waiting for choice of Play Game from MacBall menu
    
      - Allow input of ball speed *(CheckSpeed)*
    
  WEND

- **NewBall:**
  
  - Subtract ball from those remaining
  
  - Show current speed *(DrawSpeed)*
  
  - Allow movement of paddle before ball release *(MovePaddle)*
  
  - Set random direction of ball movement

- **IncSpeed:** increase speed if not user-set

---

FIGURE 4. Program outline (continued)
PlayBall:
    WHILE player hasn't missed ball, or in demo mode
       move the ball (UpdateBall)
       move the paddle (MovePaddle)
    if an apple has been hit:
       remove apple, increase score, bounce ball (CheckApple)
       if ball hits paddle, bounce ball, make a sound
       if ball hits wall, bounce ball, make a sound
    WEND
    if ball has missed paddle, make a sound and remove image of ball

CheckApple:
    if apple was hit, make a sound and remove apple
    increase score
    if last apple was hit, draw a new set of apples

UpdateBall:
    draw ball at new position of ball
    remove ball at old position

DrawScoreArea: draw initial score area

CheckSpeed:
    set ball speed based on location of pointer in speed box
    display new speed (DrawSpeed)

DrawSpeed: display new speed

MovePaddle:
    read position of mouse pointer
    if pointer in game window, show paddle, else show pointer
    if new paddle position is different than last:
       display paddle at new position
       erase paddle at old position

Instructions: show instructions

TitleScreen: show first title screen

DrawApples: display new set of apples, reset apple array

FormPaddle: define shape of paddle and save in array

FormApple: define shape of apple and save in array

FormBall: define shape of ball and save in array
Initialization and drawing objects

First, arrays for a number of purposes are dimensioned [Init 1]. One use of arrays here is to specify images. We dimension arrays of bits for three images: the ball (ballPic), the paddle (padPic), and the apple (applePic). Since an array is the logical data structure for dealing with characteristics of a number of identical items, several arrays are used to handle the three columns of apples on the screen. There will be up to 27 apple images on the screen at a time, and each is subject to being hit and removed by the ball. The array apHere is used to indicate whether a given apple is still present: It is essentially an array of logical (true/false) values. Two more arrays (apC and apR) hold the coordinates of the apple’s column (x) and row (y), respectively. A single two-dimensional array could be used here, producing code that would be more compact but a little harder to understand. The choice is a matter of programmer preference or style. Finally, we have a pen pattern for the ball (grayPat) and paddle netting (padPat).

After the arrays comes the initialization of constants. The significant ones to note are those that define important areas of the screen. maxX and maxY are the horizontal (x) and vertical (y) boundaries within which the ball can travel. minApX is the x coordinate of the first column of apples: We’ll need this to know when to start checking to see if the ball has hit an apple. (We don’t check for a hit until we know that the ball is in the vicinity of the apples.) The maximum number of balls, the minimum speed, and the boundaries of the speed-selection bar are also given.

In [Init 2] we define the size and style of the window. Then the subroutines FormApple, FormBall, and FormPaddle are called to draw and save the apple, ball, and paddle images. As usual, all three subroutines GET their images to store them in their respective arrays, then use PUT to erase them on the screen.

Using QuickDraw’s PAINTOVAL function, FormApple [SR 15] draws the apple as two adjacent ovals and then draws a third, inverted oval to take the “bite” out of the right side of the apple—a nice trick. Finally, the stem of the apple is drawn as three lines and a single point. FormBall [SR 16] simply fills an oval with the gray pattern for the ball and then draws a circle around the pattern to give the ball a definite boundary (you wouldn’t want to bat around a fuzzy blob, would you?). FormPaddle [SR 14] draws the paddle in three parts. The paddle itself is an oval like the ball but filled with a different pattern. The handle looks like it contains five lines, but there are actually only three. The first line, from coordinates (10,30) to (9,50) appears broken because at some
arbitrary point the routine used by the LINE statement has to "jump" one pixel to the left to go from an x coordinate of 10 to one of 9. The other side of the handle is drawn in the same way, with the "jump" in the same place. This is a trick that you can exploit in your own programs.

The MacBall menu is set up and turned on with the lines of code at [Init 3]. TitleScreen [SR 12] is called in [Init 4]. Note the use of LOCATE to position the pen for drawing the two character strings MacBall and Knock out the Apples. This handy statement automatically moves to a preconfigured column and row position corresponding to the size of the font used, making it unnecessary to use pixel coordinates. The paddle, ball, and three nice, large apples are then drawn to complete the title screen.

The preliminary instruction-screen subroutine, instructions [SR 11] is called next. The window area is set, and the instructions are printed on the screen along with a Continue button, which lets you proceed from the instruction screen to the game.

The initial game-screen subroutine, DrawScoreArea [SR 7], is called next. The area at the bottom of the screen that will give the score and show the current ball speed is first set up. The initial score is 0, and the speed bar is empty. When the player clicks the mouse pointer in the speed box to select the speed, the pointer’s (x,y) coordinates are read and converted into the distance in pixels that the ball will travel at every update. After the score area is drawn, we set the initial paddle y coordinate and PUT the paddle there. Note, however, that the paddle x coordinate (10) never changes, so the paddle can be moved only up and down.

A generic game structure

Before we get to the main part of MacBall, let’s step back for a moment and look at some general aspects of the structure of the kind of games we present in this book. If you’ve studied several of these games, you’ve no doubt observed similarities in their structure. With minor variations, all of the games follow this basic pattern:

WHILE player wants to play
    set up new game
    WHILE there are more (balls, spaceships, whatever)
        WHILE not out of play
            update positions of screen objects
            check for scoring events
        WEND (one play)
    WEND (one game)
WEND (program)
You can compare the preceding outline with the program outlines for MacGurkha, MacDart, and the other games to see how they fit into this pattern. The use of the WHILE...WEND statement makes it easy to specify what characterizes a given stage of the game and thus what must happen for the program to go on to the next stage. This structure is obscured in older forms of BASIC that have to use leapfrogging IF...THEN GOTO statements and don't have named labels for subroutines. Most games break down into a multilayered WHILE...WEND structure because the player has relatively few options and is usually doing only one thing at a given stage of play. Special events like firing a laser can be handled by setting mouse- or keyboard-event interrupts and checking to see whether they cause a termination of the current WHILE...WEND condition.

Actual games will usually have several more layers of “nested” WHILE...WEND loops and occasionally other kinds of structures. For example, the “one play” loop in the MacBall program (PlayBall) uses its own WHILE...WEND loop to move the ball and paddle on the screen. These additional layers are simply further refinements that break the code down into discrete events.

By now you should be able to outline a game of your own design, following this same pattern. Even complex video games that have to keep track of many objects can be outlined in this way.

Main program

Now let's get back to MacBall. The main part of the program starts by setting cursFlag to true, then starts an endless loop which is interrupted only if the player decides to quit from the MacBall menu. (By the way, all menu events are handled by the MenuEvent subroutine [ME 0]; see “MacCur sor” for a description of menu event-handling.) Then, the Play Game and See Demo menu items are enabled. StartGame [SR 1] is called to set up the new game. This subroutine initializes the ball velocity (which has two components, xVel and yVel) and the number of balls already played (which is 0 at the beginning of a new game). The variable userSpd is a logical variable that indicates if the speed of the balls will be controlled by the program (userSpd = false) or by the player (userSpd = true). It is initially set to false at the beginning of this subroutine.

Next, DrawSpeed [SR 9] is called. This subroutine, which draws the current speed bar, begins by figuring the value of newSX based on the values of xVel, minXVel, and sBoxX. It then erases the old speed bar, draws a new one, and then sets the value of oldSX to newSX.
After setting the score to 0, the paddle is drawn and DrawApples [SR 13] is called. It draws an “orchard” of 27 apples in a 3 column by 9 row relationship on the right side of the screen. This subroutine stores each apple’s x and y coordinates in the appropriate arrays and sets all the elements of the apHere array to true to indicate that all the apples are present.

A FOR…NEXT loop in StartGame then draws five balls at the bottom of the screen by taking advantage of CHR$(165), a special character consisting of a small filled circle, or “bullet.” These ball symbols represent the number of balls remaining to the player, not counting the ball about to be in play (thus the total of six balls per game).

Now it’s time to find out what the player wants to do. The program uses TEXTMODE(1) to prevent printing blanks at the end of each printed line, and then asks the player to choose an option from the MacBall menu, or to set the speed in the speed bar, if that hasn’t already been done. The next few lines check the logical value of cursFlag and the program sits in a WHILE NOT…WEND idle state until the player decides to choose a speed (see the IF MOUSE(0) < 0 line) or the Play Game or Demo options from the MacBall menu. If the MOUSE(0) < 0 condition is fulfilled, the mouse button has been pressed, so CheckSpeed [SR 8] is called.

CheckSpeed first obtains the mouse x and y coordinates via MOUSE(1) and MOUSE(2). The coordinates are then checked to see if they are within the speed box area. If they aren’t, the player was presumably doing something other than trying to set the speed, so the subroutine returns. Otherwise, the flag userSpd is set to true, and xVel is scaled down to an appropriate value for the ball speed (a ball traveling at 300+ pixels per jump, for instance, would appear to teleport about the screen and make for a very frustrating game!). DrawSpeed [SR 9] is used to erase the old speed bar and draw the new one. The new value of newSX is saved as oldSX so that the current speed bar can be erased when a new speed is set.

It’s important to realize that this program involves two concepts of motion: speed and velocity. Speed is the magnitude of the motion regardless of direction: It is expressed as the number of pixels the object (in this case, the ball) is moved each time it is updated. Velocity is speed plus direction: A velocity \((xVel)\) of \(-30\) is motion at the rate of 30 pixels per update toward the left side of the screen, while an \(xVel\) of 30 is motion at a rate of 30 pixels per update to the right side of the screen. (We’re concerned only with velocity along the x axis here; we’ll show later how the y velocity is derived.)
Back in StartGame, the program is still waiting for the player to either choose an option from the MacBall menu, or change the speed. Once Demo or PlayGame is chosen, a 0 is moved into position to show that the initial score is zero and the little message box is erased. Control now returns to the main program loop, where the PlayGame and Demo menus are now disabled. The program checks to see if there are any balls left to play and, if there are, moves into the playing phase.

**Play loop**

Assuming the player wants to play, the program now enters the inner game loop [Main 2]. This play loop will be executed as long as the player hasn't run out of balls.

The play loop begins by calling NewBall [SR 2] to put a new ball in play. First, the representation of the previous ball beneath the playing screen is erased by printing a blank in the appropriate place. The variable *ball* holds the number of balls played so far; the first time around, that is 0, so the blank is printed far enough to the right so that none of the five balls is erased. As the player misses the balls with the paddle and they go out of play, the expression $190 - ball \times 10$ decreases (gives a lower x coordinate), and successive blanks are printed toward the left side of the screen at the proper intervals to erase each successive ball symbol (which, you will remember, is a regular ASCII character and so can be erased by printing a blank over it).

A quick call to DrawSpeed [SR 9] puts the speed bar at the bottom of the screen with the current speed of the ball shown.

A FOR...NEXT loop then calls MovePaddle [SR 10] 100 times, giving a reasonable time delay for the player to adjust the paddle position. This subroutine starts by reading the mouse y position. The curious statement $dummy = MOUSE(0)$ shouldn't make you feel like a dummy—it is used here because, although the value returned by MOUSE(0) is unimportant (the mouse button probably isn't being pushed, and we're not checking for a click now, anyway), MOUSE(0) has the additional function of updating all the values that will be returned by calls to MOUSE(1), MOUSE(2), and so on. Since the mouse may have just been moved, it's important to get the latest news on the value of MOUSE(2) (the current mouse y coordinate), when it is asked for in the next line of code. (Since the paddle can be moved only up and down, the x coordinate is unimportant.)

MovePaddle first checks to see whether the logical value of $MOUSE(2) > 0$ is the same as the value of *cursFlag*. This is used to check on
the cursor position to see if it is on the playing screen, or in the menu area. If the cursor isn't on the playing screen, then the base of the paddle is set to the nearest boundary of the playing screen (top or bottom). This new y coordinate ($padY1$) is then checked to see if it differs from the old position stored in $padY$. If it doesn't, the paddle isn't being moved, and control returns to NewBall [SR 2]. If the values are different (in other words, the paddle is or was moved), the new paddle is drawn, the old one is erased, and the current paddle y coordinate is saved for future use.

NewBall sets the initial y axis velocity to a random value between 2 and 7. This means that the ball will always drop, but its rate of descent will vary. $yVel$ makes use of both the INT (integer) and the RND (random number) functions, to generate a random value between 2 and 7.

Next, NewBall sets the x axis velocity by taking the negative of the absolute value of $xVel$ to ensure that the x component of the velocity is negative, since the ball will be coming toward the paddle from the apple area on the right side of the screen.

By setting these two velocity values when the ball is launched, it will always move down and toward the left side of the screen, but the steepness of the slope will vary due to the random variation in the y component of the ball's velocity.

Finally, the starting ball's x and y coordinates are set: The x position is always just to the left of the first row of apples, but the y position is randomized, so we don't know where the ball is going to come from. Control is then returned to the main program loop.

Now, as the umpire would say, it's time to play ball — so PlayBall [SR 4] is called by the main program loop. This subroutine starts by setting the hit flag to false (the ball is just starting toward the paddle and hasn't hit any apples yet). $oldXB$ and $oldYB$ are set to $xBall$ and $yBall$ respectively for future reference.

A WHILE...WEND loop is used to handle the play of the ball. The ball remains in play as long as its x coordinate ($xBall$) remains greater than 15 (if it's less than 15, the ball's gotten by the paddle) or the program is in Demo mode (where the ball is bounced off the left screen boundary — more on this later — and stays in play even if it misses the paddle).

The real key to the entire game comes in the next few lines. The ball position is updated by adding the current x velocity to the ball's x coordinate and the y velocity to the y coordinate. UpdateBall [SR 6] is then called to draw the ball in its new position, and MovePaddle [SR 10] moves the paddle to the new mouse y coordinate if the player has moved the mouse in the meantime.
Next a check is made to see if the ball has entered the area of the “apple orchard.” If it has, it’s time to check to see whether an apple has been hit. Since the ball will often hit a number of apples in succession after hitting the first one, the check is also automatically done once the first apple has been hit by the current ball (thus the AND NOT hit specification).

The check is done in CheckApple [SR 5] by dividing the ball’s x and y coordinates by the number of columns and rows respectively to find which column and row the ball has entered. This is used to look up a value in the apHere array. If the value in the apHere array at this column and row is true, it means an apple is present and has been hit by the ball. In this case, a sound is produced, the apple is erased, and apHere(col,row) is set to false to indicate that there is no longer an apple at that spot. The X velocity is reversed (xVel, multiplied by −1), making the ball appear to bounce. The hit flag is set to true (it may, of course, already be true), and 1 is added to the score, which is updated on the display. Every 27 apples (score MOD 27 = 0) a full batch of apples has been cleaned out, so DrawApples [SR 13] is called to refill the orchard.

Eventually the ball is going to bounce off either a wall or an apple and head back toward the paddle, so PlayBall next checks for the ball hitting the paddle. If this check is negative, the ball didn’t hit the paddle and the routine drops to the label NoPadHit. If the ball is close enough to hit the paddle, its x velocity is reversed in order to bounce it, and its x position (xBall) is set to 40. The y velocity will depend where on the paddle the ball hit: yBall is subtracted from the y coordinate of the center of the paddle (padY−15) and multiplied by the constant 0.6. This means that the farther from the center of the paddle the ball hit, the higher the y velocity (which will be positive or negative depending on whether the ball hit toward the top or bottom of the paddle), the steeper the angle at which the ball will move away from the paddle, and the faster the ball will travel. If you have trouble visualizing this, refer to Figure 3, or hit the ball a few times off different parts of the paddle to really experience the effect.

Next, there’s a check for the ball having hit a wall. If the ball is close enough to the right wall (or the left if we’re in Demo mode), the ball’s x velocity is reversed. If the ball has gotten up to the top wall or down to the bottom one, the y velocity is reversed.

After completion of this section, if the user hasn’t quit, control goes back to the top of PlayBall's WHILE...WEND loop. If the game is a Demo, or xBall is greater than 15, play continues. But if xBall isn’t greater than 15, the ball has gotten by the paddle and control falls through
the bottom of the loop, which means that the ball goes out of play. The ball is erased, and a beep informs the player that he or she has flubbed it.

Control returns to the main program loop after the ball has gone out of play, and IncSpeed [SR 3] is then called. If the ball speed was set initially by the player, control returns to the main program routine. If the speed wasn’t set by the player, the absolute value of $x\text{Vel}$ is incremented by 3, and control returns to the main program routine. Then the next ball (if there is one) is played. When all the balls have been played, a new game is started by Start Game (with the score reset to 0). This continues until the player quits, via the MacBall menu.

Suggestions for MacBall

You could modify MacBall so the player gets a new ball every time a full set of apples is knocked out. This would allow good players to play longer and get higher scores, but it might make the game too easy.

Another possibility would be to have one or more “poison apples” randomly selected. These apples (which would be filled with a pattern different from that of the regular apples) would cause the player to lose the ball if hit from any direction except behind. This could make the game considerably more challenging. Other ideas you might consider include adding gravity to the ball (like MacDart), or allowing the score to grow as large as you wish, instead of resetting it each time. Or, you could also add a fourth row of apples, or even design a subroutine that places a magic apple in the orchard that would turn into several apples when hit by a ball.
MacBall program listing

MacBall

Knock the apples down with the ball.

INIT 1

DEFINT A-Z
DIM ballPic(50), padPic(120), applePic(150) 'bitmap arrays for images
DIM apHere(3,9) 'true if this apple is present
DIM apR(9), apC(3) 'coordinates of apple rows & columns
DIM grayPat(3), padPat(3) 'gray & paddle grid pen patterns

FOR n = 0 TO 3 : grayPat(n) = &HAASS : NEXT 'initialize set constants
padPat(0) = &H8855 : padPat(1) = &H2255
padPat(2) = &H8855 : padPat(3) = &H2255 'initialize gray pattern

false = 0 : true = NOT false
maxX = 465 : maxY = 250 'max horiz, vertical boundaries for ball
minApX = 341
maxBalls = 6
boxX = 80 : boxY = 100 'x location of the first row of apples
minXVel = 10
sBoxX = 300 : sBoxY = 265 'maximum number of balls per game

INIT 2

WINDOW 1, "MacBall", (5,30) - (505,330), 4 'open window and form all pictures
GOSUB FormApple
GOSUB FormBall
GOSUB FormPaddle

INIT 3

MENU 6, 0, 1, "MacBall" 'install MacBall menu
MENU 6, 1, 0, "Play Game"
MENU 6, 2, 0, "See Demo"
MENU 6, 3, 0, ".”
MENU 6, 4, 1, "Quit to BASIC"
MENU 6, 5, 1, "Quit to DeskTop"
ON MENU GOSUB MenuEvent : MENU ON

INIT 4

GOSUB TitleScreen
GOSUB Instructions
GOSUB DrawScoreArea
padY = 40 'set initial paddle location
PUT (10,padY), padPic

'draw initial screen

'and show paddle
**MAIN LOOP**

cursFlag = true
WHILE true
  'play until Quit selected from menu
  'enable Play Game and Demo menus
  MENU 6,1,1
  MENU 6,2,1
  GOSUB StartGame
  'get Game or Demo, initialize balls etc.
  'disable Play Game and Demo menus
  MENU 6,1,0
  MENU 6,2,0
  WHILE ball < maxBalls
    GOSUB NewBall
    GOSUB PlayBall
    GOSUB IncSpeed
  WEND
WEND

**MENU EVENTS**

MenuEvent:
  MENU 6,0,0
  item = MENU(1)
  ON item GOSUB GameMenu, DemoMenu, BasQuit, DeskQuit
  MENU 6,0,1
  RETURN

GameMenu:
  playGame = true
  RETURN

DemoMenu:
  demo = true
  RETURN

BasQuit:
  MENU RESET
  END

DeskQuit:
  MENU RESET
  SYSTEM

**SUBROUTINES**

StartGame:
  'start new round of balls
  xVel = minXVel : yVel = 7
  ball = 0
  userSpd = false
  GOSUB DrawSpeed
  score = 0
  PUT (10,padY), padPic
  'no balls played yet
  'userSpd true if user has set speed
  'show initial speed
  'erase old paddle

  'reset and play game
  RETURN

  'play demo
  RETURN

  'quit to BASIC
  RETURN

  'quit to Mac Desktop
  RETURN
PadY = 100: PUT(10, padY), padPic
GOSUB DrawApples
CALL MOVETO(100, 275): PRINT "BALLS: ";
FOR n = 1 TO 5: PRINT CHR$(165); " ": NEXT
LINE (boxX, boxY) - (boxX + 230, boxY + 35), , b
CALL TEXTMODE(1)
CALL MOVETO(boxX + 5, boxY + 13)
PRINT "Choose an option from the MacBall menu"
CALL MOVETO(boxX + 5, boxY + 30)
PRINT "or set the speed in the speed bar."

IF NOT cursFlag THEN CALL SHOWCURSOR
  cursFlag = true
demo = false: playGame = false
WHILE NOT demo AND NOT playGame
  IF MOUSE(0) < 0 THEN GOSUB CheckSpeed
WEND
LINE (boxX, boxY) - (boxX + 230, boxY + 35), 30, bf
CALL TEXTMODE(0)
CALL MOVETO(45, 275): PRINT "0";
RETURN

SR 2

NewBall: set up to play new ball
CALL MOVETO(190 - ball * 10, 275): PRINT " ";
ball = ball + 1
GOSUB DrawSpeed
FOR n = 1 TO 100
  GOSUB MovePaddle
NEXT
yVel = INT(RND(1) * 6) + 2
xVel = ABS(xVel) + 3
xBall = 335: yBall = 250 - RND(1) * 220
oldXB = xBall: oldYB = yBall
RETURN

SR 3

IncSpeed: increment ball speed if not user set
IF userSpd THEN NoSChange
  xVel = ABS(xVel) + 3
NoSChange:
  RETURN

SR 4

PlayBall: play until user misses ball
hit = false 'means we haven't hit an apple yet
oldXB = xBall: oldYB = yBall: PUT(xBall, yBall), ballPic
WHILE demo OR xBall > 15
    xBall = xBall + xVel : yBall = yBall + yVel
    GOSUB UpdateBall
    GOSUB MovePaddle
    IF (xBall >= minApX) AND NOT hit THEN GOSUB CheckApple
        'hit an apple?
    IF xBall >= 30 OR (yBall > 30 + padY OR yBall < padY - 10) THEN NoPadHit
        xVel = -xVel : xBall = 40
        yVel = yBall - padY - 15 : yVel = d *.6
        SOUND 500,1
        hit = false
    ENDWHILE
    'end while ball in play
FOR n = 800 TO 500 STEP -50
    SOUND n,1
NEXT
PUT (xBall,yBall), ballPic
    'remove last image of ball before next ball
RETURN

CheckApple:
    '____________________________________ check to see if apple was hit
    col = INT((xBall - minApX) / 31) + 1 : IF col > 3 THEN col = 0
    row = yBall / 27 + (yBall => 257)
    IF NOT apHere(col,row) THEN NoApple
    FOR n = 1 TO 4 : SOUND 800,1 : NEXT
    PUT (apC(col),apR(row)), applePic
        'erase apple, mark as gone, incr. score
    apHere(col,row) = false
    xVel = -xVel : hit = true : score = score + 1
    CALL MOVETO(45,275) : PRINT STR$(score);
        'deliver new crop of apples
    IF score MOD 27 = 0 THEN GOSUB DrawApples : PUT (xBall,yBall), ballPic
NoApple:
    RETURN

UpdateBall:
    '_____________________________________ update ball to new position
    PUT (xBall,yBall), ballPic
        'draw new ball
    PUT (oldXB,oldYB), ballPic
        'erase new ball
    oldXB = xBall : oldYB = yBall
    RETURN

DrawScoreArea:
    '_____________________________________ draw initial score area
    CLS
    LINE (5,8) - (maxX + 20, maxY + 10), , b
        'draw border & speed box
LINE (sBoxX - 1, sBoxY - 1) - (sBoxX + 164, sBoxY + 10), , b
CALL TEXTFACE(32)  
CALL MOVETO (230,275) : PRINT "SPEED"

CALL MOVETO(10,275) : PRINT "Score: 0"
RETURN

CheckSpeed: change speed based on mouse x pos
x = MOUSE(1) : y = MOUSE(2)
IF x < sBoxX OR x > sBoxX + 160 OR y < sBoxY THEN NoCS
userSpd= true
xVel = (x - sBoxX) / 8 + minXVel
GOSUB DrawSpeed
NoCS:
RETURN

DrawSpeed: draw current speed bar
newSX = (ABS(xVel) - minXVel) * 8 + sBoxX
LINE (oldSX,sBoxY) - (oldSX + 3, sBoxY + 9), 30, bf  
erase old speed bar
LINE (newSX,sBoxY) - (newSX + 3, sBoxY + 9), 33, bf  
draw new speed bar
oldSX = newSX
RETURN

MovePaddle: move paddle based on where pointer location
dummy = MOUSE(0) : padY1 = MOUSE(2) - 45
IF (MOUSE(2) < 0) = cursFlag THEN CursorOK
IF cursFlag THEN CALL HIDECURSOR ELSE CALL SHOWCURSOR
cursFlag = NOT cursFlag
CursorOK:
IF padY1 < 0 THEN padY1 = 0 ELSE IF padY1 > 235 THEN padY1 = 235

IF padY1 = padY THEN NoPadMove
PUT (10,padY1), padPic : PUT (10,padY), padPic  
draw new, erase old pos
padY = padY1  
set new paddle location

NoPadMove:
RETURN

Instructions: show instructions
WINDOW 2, , (16,45) - (495,315), 2
PRINT
PRINT "Instructions for MacBall"
PRINT
PRINT "Hit the ball with the paddle to knock out as many apples as you can."
PRINT "If you don't set the speed yourself, each time you miss a ball the speed"
PRINT "is increased."
PRINT "At the beginning of the game you make a choice of what to do from the"
PRINT "MacBall menu. Demo just shows you how the game works. The paddle"
PRINT "doesn't have to hit the ball; the ball just never disappears off the left"
PRINT "side of the screen."
PRINT "Pull down the MacBall menu and quit the game any time you want."
PRINT
CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1: WINDOW CLOSE 2
RETURN

TitleScreen: ____________________________ draw title screen
  CLS
  CALL TEXTFONT(0) : CALL TEXTSIZE(36)
  LOCATE 1,2 : PRINT "MacBall!"
  LOCATE 2,2 : CALL TEXTSIZE(12)
  PRINT "Knock out the Apples!"
  CALL TEXTFONT(1)
  PUT (10,150) - (60,250), padPic
  PUT (250,90) - (270,110), ballPic
  FOR n = 0 TO 2
    PUT (350, 10 + n * 80) - (430, 82 + n * 80), applePic
  NEXT
  FOR t = 1 TO 10000 : NEXT
  RETURN

DrawApples: ____________________________ draw and initialize Apple array
  'set up apple array in (column, row) order
  'set up x,y coordinates for apples and draw apples
  FOR row =1 TO 9
    apR(row) = row * 27 - 13
    FOR col = 1 TO 3
      apHere(col,row) = true
      apC(col) = (col - 1) * 31 + minApX
      PUT(apC(col), apR(row)), applePic, PSET
    NEXT col
  NEXT row
  RETURN

FormPaddle: _____________________________ form paddle and save in padPic
  r(0) = 0 : r(1) = 0 : r(2) = 32 : r(3) = 25
  CALL PENSIZE(2,1)
  CALL FILLOVAL(VARPTR(r(0)), VARPTR(padPat(0)))
  CALL FRAMEOVAL(VARPTR(r(0)))
  LINE (10,30) - (9,50)
  LINE -(15,50)
  LINE -(14,30)
  GET (0,0) - (25,50), padPic : PUT (0,0), padPic
  RETURN
FormApple:

```
form apple shape and put into Apple
r(0) = 19 : r(1) = 17 : r(2) = 39 : r(3) = 31
CALL PAINTOVAL(VARPTR(r(0)))
r(1) = 27 : r(3) = 41
CALL PAINTOVAL(VARPTR(r(0)))
r(0) = 20 : r(1) = 36 : r(2) = 34 : r(3) = 55
CALL PAINTOVAL(VARPTR(r(0)))
CALL INVERTOVAL(VARPTR(r(0)))
LINE (29,19) - (33,15)
LINE (31,15)
LINE (29,19)
PSET (31,16), 33
GET (14,14) - (44,40), applePic
PUT (14,14), applePic
RETURN
```

FormBall:

```
form ball picture
r(0) = 0 : r(1) = 0 : r(2) = 10 : r(3) = 10
CALL FILLOVAL(VARPTR(r(0)), VARPTR(grayPat(0)))
CALL FRAMEOVAL(VARPTR(r(0)))
GET (0,0) - (10,10), ballPic
PUT (0,0), ballPic
RETURN
```
During World War II a British officer had a tough assignment for the legendary Nepalese Gurkha riflemen: He wanted them to make a parachute drop. The officer was somewhat surprised when only about half of the ten thousand Gurkhas volunteered for the mission, since these troops were known for their absolute fearlessness and their ability to carry on under any
conditions. But later, after he explained to them that each man would be provided with a parachute before jumping out of the plane, the rest of the Gurkhas quickly volunteered, too.

Well, your modern Gurkha is guaranteed a parachute, and no one is trying to shoot him out of the sky. The job still isn't a piece of cake, though, if you expect to get your Gurkha successfully into the landing zone: There are blimps to watch out for, the parachute tends to drift, and you have to know when to open the chute and how much to guide the Gurkha into the target area. As you'll see, your score for a successful jump will depend on a fine sense of timing, and the willingness to take a chance sometimes—true Gurkha fearlessness!

**How to play MacGurkha**

MacGurkha starts with the title screen shown in Figure 1. A larger version of the plane your Gurkhas will jump from appears and flies across the screen from right to left. In a moment, a little Gurkha figure emerges from the aircraft, and falls headlong to the ground, without his chute ever opening. Woops! don't let that happen to you!

![MacGurkha title screen](FIGURE 1. The MacGurkha title screen)
After a moment, the playing screen shown in Figure 2 appears. Your plane (smaller than the one on the title screen) flies in from the right side of the screen, accompanied by a nice engine sound. At the bottom of the screen, you can see the target area or jump zone, which is designated by two little markers. One or more blimps are also tethered to the bottom of the playing screen. They don't move, but if your Gurkha touches one it's instant death! The score for the last jump and the total score for the game so far are shown at the bottom right, just below the playing screen.

Wait for the plane to fly to the point where you want your Gurkha to jump. You'll have to take the plane's speed into consideration when choosing the release point, because the Gurkha starts out with the plane's forward velocity. This means that the faster the plane is traveling, the more your free-falling Gurkha will tend to move diagonally rather than straight down.

To release the Gurkha, click the mouse button. The plane then disappears, and the Gurkha (with the chute still closed, unless you lingered too long on the mouse button) starts his free-fall descent. When you
think the moment is right, click the mouse once more to open the chute. Once the chute is open, it will tend to drift from side to side randomly. You steer the chute by moving the mouse pointer left or right while holding the mouse button down. As long as you hold the button down, the chute will move toward the side the pointer is on. (While you're steering, don't forget to look out for blimps!) If you succeed in making the Gurkha land in the jump zone, you'll hear a bugle call and see your score for the jump as well as your new total score for the game. Your score for the jump actually depends on three separate factors. The longer you wait before opening the chute, the higher the score. Your score is also higher when the jump zone is narrower. On the other hand, using the mouse to steer the chute costs you points, so you should try to steer as little as possible.

The game gets tricky when several blimps are bunched close to the target zone. Sometimes you can drop the Gurkha far enough from the balloons to avoid them and then drift in over the target in a kind of glide, using the Gurkha's forward velocity set by the plane's motion, but you have to cope with the random-drift factor and the fact that time for maneuvering is limited. Since a free-falling chutist presents a smaller profile to the blimps than one with a chute open, free-falling past a blimp and then opening the chute is often a good tactic, although sometimes the chute gets caught in the guy wires. If the blimp is too low, however, you may not have time to maneuver before the Gurkha hits the ground. The trick is to know how far you can let him free-fall (thus increasing your score) before you have to pull the ripcord. At the same time, you want to minimize steering, and these two objectives are sometimes at odds with each other—but it's decisions like these that make the game so interesting. After some practice you can call yourself an honorary Gurkha!

After a jump, note the number of extra Gurkhas displayed in the lower left corner of the screen. Each time you are unsuccessful (you miss the jump zone, don't open the chute, or hit a blimp) you lose one of these Gurkhas. When none are left, you're playing with the last one—you have only one life left. When you run out of lives, the game is over.

**How MacGurkha works**

The heart of the program is the main game loop at [Main 1]. The outer `WHILE...WEND` loop lets the player play repeated games until choosing one of the standard menu quit options. The inner `WHILE...WEND` loop handles new jumps as long as the player has “lives” left. After the
playing screen is set up for a new jump, various subroutines are called in succession to control the stages of the jump: flying the plane, releasing the Gurkha into free-fall, drifting down with the chute open, steering, checking for collision with a blimp, scoring, and so on. The program outline in Figure 3 makes the structure of the program clear.

**Initialization**

- define size of graphics arrays for plane, chute, man
- set constants for pattern, logical variables, screen locations
- create playing window
- create and save shapes *(MakePictures)*
- install MacGurkha menu, enable menu event trapping
- display title screen *(TitleScreen)*
- show instructions *(Instructions)*

**Main Program**

```plaintext
WHILE true (endless loop interrupted by menu events)
    initialize lives, score, jumps
    WHILE lives left
        set up playing screen *(SetUpScreen)*
        set jump points to zero, establish random plane speed
        fly plane until button pressed *(FlyPlane)*
        start Gurkha's free-fall *(FreeFall)*
        save free-fall distance for score
        fall with open chute *(ChuteFall)*
        if Gurkha hit blimp or ground, show hit *(ShowHit)*
        if Gurkha landed in drop zone, show message *(MadeIt)*
        if Gurkha outside of drop zone, show message *(Casualty)*
    WEND
    announce end of game *(Announce)*
    wait for menu event to restart game
WEND
```

**Menu Events**

- **MenuEvent**: branch to appropriate subroutine
- **PlayMenu**: play another game
- **Instructions**: display instructions
  - create second window
  - print instructions
  - create Continue button and wait for button click
- **BasQuit**: quit to BASIC
- **DeskQuit**: quit to Mac Desktop

**Subroutines**

- **ShowHit**: do sound effect
- **Announce**

---

*FIGURE 3. Program outline*
subtract a life

**MadeIt:**
- give points (mouse usage, accuracy and length of free fall)
- do sound effect
- announce success *(Announce)*

**Casualty:** subtract a life, do sound effect

**SetUpScreen:**
- clear playing area
- set initial plane location
- draw landing zone and ground
- update score and lives left *(UpdateStatus)*
- draw plane in starting position
- draw one blimp in random location for each successful jump

**Fly Plane:**
- increment plane location by speed factor
- make sound proportional to speed
- keep moving plane until button press

**Free Fall:**
- WHILE button not pressed, blimp or ground hit
  - erase Gurkha in old position
  - calculate new position, add random drift
  - draw Gurkha in new position
  - check for blimp hit *(CheckBlimp)*
- WEND

**Chute Fall:**
- WHILE Gurkha above ground and not hitting blimp
  - erase Gurkha in old position
  - if mouse button pressed, alter direction of fall
  - add random drift to fall
  - draw Gurkha in new position
  - check for blimp hit *(CheckBlimp)*
- WEND

**CheckBlimp:** determine if Gurkha has struck blimp

**Clear Mouse:** clear mouse events

**Update Status:**
- display points made this jump, total points in game
- display Gurkhas remaining

**Title Screen:**
- display title and other text
- fly plane until at drop point
- drop Gurkha until he hits ground or button clicked

*Figure 3. Program outline (continued)*
Initialization

In [Init 1] all variables from a-z are declared to be integers by the DEFINT statement, except variables whose names begin with q, which are declared to be single-precision real numbers by the DEFSNG statement. In general, variables holding screen coordinates are best set to be integers, because you can’t have half a screen pixel anyway. On the other hand, probabilities need to be real numbers, as do variables that involve complex motion or draw curved surfaces. Other processes where fractions often occur and accuracy needs to be preserved through repeated calculations are also best done with real numbers.

Three sets of statements are used to dimension the arrays that will hold the graphics data for the game’s three movable objects: the plane (airPlane), the Gurkha with an open chute (openChute), and the Gurkha with closed chute (clsChuteP). The same procedure is used in each case. First, the horizontal dimension or width (planeH) of the object is set. Next, the vertical dimension or height (planeV) is set. A formula defined by the ArraySize function is then used to determine the number of integer elements that the array will need to hold the graphics image.

As you probably recall if you’ve worked with BASIC graphics, a graphic image is actually an array of bits (or “bit map”) in which each bit corresponds to one screen pixel of the image. The general formula for the number of bytes (8 bits) needed to store a graphics image with GET is given in the Microsoft BASIC documentation under the GET statement. We’ve modified the variables in that formula to reflect our MacGurkha program:

\[ \text{bytes} = 4 + (YB - YT + 1) \times 2 \times \text{INT} \left( \frac{(XR - XL + 16)}{16} \right) \]

Let’s see how this formula is used to determine how the first of our images, the airplane, is stored in an array.
YB is the y coordinate of the bottom of the image rectangle; YT is the y coordinate of the top. \((YB - YT + 1)\) is equal to the height of the rectangle in pixels, or planeV in our code. (The coordinate of the top is subtracted from the bottom because y coordinates increase as you move from top to bottom.)

XR is the x coordinate of the right side, and XL is that of the left. \((XR - XL + 1)\) is the width of the rectangle in pixels, or planeH in our code. We add 16 to it, divide by 16, and take the INT in order to round it up to the next full 16-bit word. Graphics memory is organized in 16-bit words laid end-to-end across the screen, and if you have part of a word left over you have to use the whole word. We now have the total in 16-bit words, but because this formula gives us bytes, the quantity is therefore multiplied by 2 to get 8-bit bytes. Finally, four bytes are added for overhead involved in the array representation.

The actual expression in our code is basically the same except it gives a result in words (or 16-bit integer elements), not bytes. Thus the above formula is divided by 2 and the result is represented in simplified form in the function ArraySize. This same procedure is followed for dimensioning the arrays for the other two objects.

Three other constants are defined near the end of [Init 1]: the number of lives a player gets, the altitude the plane flies at, and the location of the ground (the latter two both as y coordinates).

In [Init 2] the title window is created. Here the actual images to be stored in our picture arrays are obtained, and MakePictures [SR 12a] is called. As with MacSpace and some of our other games, we provide the option of using a graphics file from which the program will get its images. This allows you to create your own "custom-made" graphics and have your games use them. If you're interested in doing this, read the chapter on Maclnterface to see how that program can be used to create graphics files and data statements for use with other programs.

To implement this option, MakePictures first attempts to load the file MacGurkha.graphics. Before attempting to open the file, an ON ERROR statement enables trapping of errors by FileError [SR 12b]. If the file is not found, an error is generated and the flag fileHere will be set to false to inform MakePictures that the file doesn't exist.

If the file is found, each of the three images is read into the appropriate array. First, the count of the number of bytes in the file is obtained. The string returned by the INPUT$ function is assigned to the variable image$. A PICTURE statement displays the image on the screen. This is followed by a GET statement that stores the image from the screen in the
array. You can't directly store the contents of the graphics file in the array, though, because the file contains a sequence of encoded graphics instructions. The image must first be drawn with PICTURE, and then it can be stored by GET as a bit map in the array.

If there is no graphics file, control goes to the label ReadPictures. The subprogram ReadArray [SP 2] is called. It is a subprogram rather than a subroutine so it can simply be given the name of the array to read DATA statements into. This makes it a general-purpose tool you might find handy in other programs that must initialize several arrays from DATA statements. The size of the array is read in first, and then a FOR...NEXT loop is used to read in the array elements.

Note that the limits for this FOR...NEXT loop are given in the form LBOUND(array) and LBOUND(array) + size - 1. The reason is that, depending on the value given in the OPTION BASE statement in a particular program, arrays can start with their first element being either element 0 or element 1. The function LBOUND returns the correct element number, and adding size - 1 gets the last element of the array. This makes the subprogram flexible enough to read arrays starting at either element 0 or element 1 and of any size.

After ReadPictures is finished and returns to MakePictures, the latter returns to [Init 2], where the MacGurkha menu is created. Note that the second and third items (Play Again and Instructions) have a 0 as the last parameter in their MENU statements: These options are deactivated (and dimmed on the screen) when the menu first appears. Since the player hasn't played yet, Play Again is inappropriate at this time. The instructions will be automatically shown following the title screen, so that option, too, is not active now.

Next, the title screen and instructions are shown. TitleScreen [SR 11] is called. After the title text is displayed, a WHILE...WEND loop is used to show the airplane flying across the screen and the unfortunate chuteless Gurkha falling from it. The plane stays at a constant altitude (or y coordinate) of 150, but as the variable pH decreases (10 is subtracted from it each time at the bottom of the loop) the plane is drawn farther and farther to the left of the screen. The usual procedure for moving an object on the screen is followed: PUT the object at a position, calculate the new position, PUT the object again at the old position to erase it (which works because PUT's default is to do an XOR with the existing screen contents), and finally PUT the image again at its new position. We also include a short SOUND statement in the loop to evoke the motor's drone.
The IF...THEN statement in the middle of the loop determines when the falling Gurkha will be shown. As long as $pH$ is greater than 300, the Gurkha doesn't jump: Control goes to the label NoFall. When $pH$ goes below 300, however, the IF statement becomes false and the picture of the "closed-chute" Gurkha is shown. His x coordinate will remain the same as the plane's, so he will be under it as he falls. Each time through the loop, though, his y coordinate is multiplied by 1.03 so he falls a little farther down the screen. At the bottom of the loop, MOUSE(0) is checked to see if the user clicked the mouse button. If so, $pH$ is set to $-1$ and the loop terminates prematurely (presumably leaving the hapless Gurkha in limbo).

Instructions [ME 2] is used to display brief instructions for the player. (These instructions are also available as a menu option between games.) After the instructions are displayed, the Continue button is created, and a WHILE...WEND loop waits for the button to be clicked, at which point the button is removed and the screen is cleared. After returning from the subroutine, we're ready to set up the game itself.

The main program

The main game loop starts at [Main 1]. This sets up a new game when the program is first run, and after each game ends, until the player decides to quit. The form WHILE true...WEND means that the loop will never terminate of its own accord, but instead by some external event, in this case when one of the quit options is chosen from the MacGurkha menu.

The player gets a full set of maxLives lives. (Five lives are more than most of us get, although fewer than those provided for the average cat.) The number of successful jumps is set to 1, which also determines the number of blimps shown at each play, and the total score is set to 0.

The main play loop

The main play loop starts at [Main 2] and handles one jump each time through. As long as the player has at least one life left, there will be more jumps. SetupScreen [SR 4] clears the playing screen and draws the target zone for the new jump. The playing screen is cleared by using a LINE statement with the box-fill option to white out the playing area.

The location of the left "flag" (flag1, the boundary of the target zone) is determined randomly, and the right flag (flag2) is located a random distance (but at least 20 pixels) from the left one. The flags and ground are then drawn. Note a nice way to get thick lines: The ground, for
example, is drawn as a filled box three pixels deep. No color is specified because the default of black is wanted.

UpdateStatus [SR 10] is called to show the points gained for the last jump and the total score thus far for the game (both are 0 the first time through, of course). The number of lives left ("Extra Gurkhas") is displayed by a FOR…NEXT loop, which draws the appropriate number of "open chute" figures in the lower left corner of the screen. Note that the horizontal, or x, coordinate to start each figure is offset successively by multiplying the width of the image (openH) by the loop index and adding 5 for spacing. Note also that the number of figures shown is actually lives – 1, since you have one life left when you have no "extra" Gurkhas. The other thing that has to be done is to erase a Gurkha that's on the screen from the previous jump if that life has now been lost. This is done by finding where the last Gurkha should currently be (offsetting in the way described above), and then using a LINE statement with the white box-fill option to erase the Gurkha past that position (if any).

After control returns to SetupScreen, the plane is drawn at its starting location (the upper right corner of the screen). A FOR…NEXT loop is then used to place the blimps. The number of blimps used is equal to the number of successful jumps the player has made so far in the game, but is always at least one, the starting value of jumps.

The x coordinate for each blimp (obsX) is random between 1 and 400, while the y coordinate (the altitude of the blimp) is random but no higher than 25 pixels below the plane and no lower than 40 pixels above the ground. This is reasonable: Since our plane can fly only in a straight line, we don’t want to put the blimp directly in its path, but on the other hand, a blimp down on the ground doesn’t make sense. The blimp coordinates are placed in rec%. A call to the QuickDraw PAINTRECT routine with coordinates offset to just below the blimp draws the blimp’s gondola. The blimp’s tail fin is drawn by calling another QuickDraw routine, PAINTARC, to draw a small arc segment. (It often takes some experimentation to come up with the best way to draw curved objects.) The propeller is simply a short line, as is the tether. Finally, two more QuickDraw routines, a FILLOVAL and a FRAMEOVAL (for outlining), take care of the "blimp" part of the blimp. After the required number of blimps are placed and drawn, control returns to the main playing loop in [Main 2].

Next, a random plane speed is determined, and it’s time to start flying. FlyPlane [SR 5] is called to start the jump approach. The plane is simply flown in a straight line by the WHILE…WEND loop until the mouse button is clicked, moving the plane across the screen by subtracting
its speed from the x coordinate each time through the loop. If the plane gets within the value of its speed on the left side of the screen (so it would go off the screen next time through the loop), it is “wrapped around” to the right side for another pass. The sound made each time through the loop gives the engine noise, which has a higher pitch when the plane is faster.

Once the mouse button is clicked, it’s time to heave the Gurkha from the plane. FlyPlane returns, and after a short delay loop (time to check the chute pack again and avoid the sergeant’s boot) the Gurkha starts falling. The Gurkha’s x and y coordinates start at the plane’s position at the time of the release. The hitBlimp and hitGround flags are set to false.

FreeFall [SR 6] handles the fall of the Gurkha until the chute is opened. The coordinates of the bottom center of the figure (hotX,hotY) are determined: This is the “hot spot,” analogous to the mouse cursor’s hot spot, whose coordinates are used to check for collisions with blimps or the ground. A WHILE...WEND loop with a rather complicated condition is entered. FreeFall continues until one of the following conditions is met:

1. A blimp is hit (as indicated by hitBlimp being set to true by checkBlimp [SR 8] at the end of the loop);
2. The mouse button is clicked (checked by the MOUSE(0) > -1 condition in the WHILE statement);
3. The Gurkha hits the ground without ever having opened his chute (splat!).

First, a sound whose pitch is a function of altitude is generated. The Gurkha (with closed chute) is drawn at the opening coordinates (manX,manY). As he falls, the SOUND statement’s pitch decreases as the y coordinate increases, producing the familiar “falling bomb” whistling sound (remember, the y coordinate increases as the altitude decreases).

The Gurkha is erased at his “old” position, and his new coordinates are then determined. He will fall a constant 4 pixels (we’re not trying to simulate gravity here). A random-drift factor is calculated: This adds some uncertainty to the game and simulates gusts of wind, the way the chute is manipulated, and so on. The plane’s speed is subtracted from the x coordinate, and then the plane speed is reduced by multiplying it by 0.9. The effect of this will be that the Gurkha starts out with a forward speed equal to that of the plane, but the effect of this speed will become progressively less, a rough simulation of wind resistance canceling the forward speed. The x coordinate of the Gurkha is checked for being less than the left screen boundary, and wrapped around the edge of the screen if necessary. Finally, the Gurkha is drawn in his new position and we then check to see if he has collided with a blimp.
CheckBlimp [SR 8] does this. It first checks to see if the "hot spot" position on the Gurkha is too close to the ground for him to hit a balloon—in that case, the subroutine simply returns. Otherwise the POINT function is checked: If it returns a 33 (corresponding to the color black), the Gurkha has collided with a blimp. Why? Because the blimps are the only objects (other than the Gurkha himself) drawn on the screen more than 30 pixels from the ground, and thus being reported as black by the POINT function. So, instead of having to record the positions of all the blimps in an array and having to check them individually for collisions, we can simply check to see if the Gurkha's hot spot (the bottom-center of the rectangle containing the Gurkha image) is black. Note that this also means there's a small chance that a Gurkha's hot spot will happen to be on a blimp's tether when it is checked, and that will also kill him. If the Gurkha hits a blimp, the flag hitBlimp is set to true.

We then return to the top of the loop and the Gurkha continues to fall, unless he hits a blimp, reaches the ground, or has his chute clicked open. Normally, the loop is exited by the player clicking on the mouse button. After the loop ends, the program checks to see if the ground was hit, and if so, sets hitGround to true. But if neither a blimp nor the ground was hit, the Gurkha is simply erased pending drawing him again with the chute open. We then return from FreeFall to [Main 3] in the main playing loop. The number of pixels the trooper has free-fallen is stored as fDist, which will be part of the score the player will receive for a successful jump. The variable mCntrl, which will be used to record the number of times the player uses the mouse button to steer the Gurkha, is initialized to 0.

However, before continuing the jump, we have to deal with the possibility that the Gurkha has already bought the farm by hitting a blimp or the ground, as indicated by the flags hitBlimp or hitGround being true. In either case, ShowHit [SR 1] is called. An "alarming" sound is generated, and the appropriate announcement is made via the Announce subprogram [SP 1]. We again use a subprogram rather than a subroutine in order to make a general-purpose tool: Any string can simply be provided to the subprogram as a parameter, without having to assign it to a global string variable first. Note the SHARED clause indicates that altitude from the main program will be used by the subprogram: It's used to offset the rectangle where the message will be printed. That way, if you change the height at which the plane flies, you don’t have to fix anything at all in this subroutine. The QuickDraw ERASERECT and MOVETO routines are used to clear out anything in the area and then position the string far enough to
the left to fit its length. After returning from Announce, ShowHit sub-
tracts one life from the player’s total. After returning to the playing loop, a
jump to the label NextJump aborts the jump.

If the Gurkha has avoided the blimps or a chuteless landing, ChuteFall [SR 7] is called. A new “hot spot,” similar to that used for the
closed-chute Gurkha, is established for collision-detection purposes. Now
the “open chute” Gurkha is drawn. The mouse button is checked, and, if it
has been or is being held down ($MOUSE(0) < 0$), the Gurkha’s x coordinate
is adjusted according to the mouse pointer’s x coordinate (returned by
$MOUSE(1)$), and using the $SGN$ (sign function) to find out if this quantity is
positive or negative. If it is positive, the mouse pointer’s x coordinate
is greater than the Gurkha’s. The Gurkha should be moved to the right, so 3
is added to the Gurkha’s x coordinate. If the quantity is negative, 3 is sub-
tracted from the x coordinate to move the Gurkha toward the left. The
count of mouse control usage $mCntrl$ is increased by 1, thus lowering the
potential score for the jump.

Next, random drift is calculated, similar to that used in free-fall,
except that it’s greater, since an open chute drifts much farther than a
closed one. This is randomly added to or subtracted from the x coordinate.
A constant fall of 3 pixels is then added to the y coordinate. The x coordi-
nate is wrapped around if the left boundary of the screen was reached.
The Gurkha is drawn in his new position, and a short delay loop is exe-
cuted. This delay loop both simulates the slow fall of a parachute and gives
the player time to do steering with the mouse if desired. CheckBlimp is
then called again to check for a collision.

When ChuteFall is exited, control returns to [Main 3] in the main
playing loop. The fall is at an end because one of three things has hap-
pened: (1) The Gurkha hit a blimp ($hitBlimp$ is true), (2) the Gurkha is on
the ground in the target zone, or (3) the Gurkha is on the ground but has
missed the target zone.

We’ve already described what happens when a blimp is hit. If a
blimp wasn’t hit, the Gurkha is on the ground and it’s time to see whether
he’s landed in the target zone. The hot spot of the Gurkha picture is
checked to see if it’s between the two target markers. If it is, the Gurkha
made a successful jump, so Madelt [SR 2] is called. Madelt calculates the
score for the jump. As we mentioned before, this is based on free-fall dis-
tance minus target size minus mouse control. Note that negative scores are
converted to 0. You can adjust the other constants used to suit yourself,
perhaps multiplying everything by a thousand to get suitably arcade-like
scoring! The score for the jump is added to the total score for the game. A “bugle call” is played, and the versatile Announce subprogram is used to print the message “You made it.” Control then goes to NextJump.

If the jump wasn’t successful, control drops through to a call to Casualty [SR 3]. Casualty subtracts one of the player’s lives (just as in the case of hitting a blimp or the ground), makes a sound somewhat like a Bronx cheer, and calls Announce to print “You missed!” Once again we arrive at the NextJump label.

Regardless of which of the three paths was taken, we have now arrived at the label NextJump. This section of code calls UpdateStatus (which we’ve already discussed) to update the player’s score and number of lives. Then, after a delay, the program calls ClearMouse [SR 9], which clears out any leftover mouse events.

The main game loop continues while there are lives left. When the player’s last Gurkha is lost, control falls through to the end-of-game section [Main 4]. The Play Again and Instructions menu options are then re-enabled, and the program waits for the player to make a selection. If Play Again is selected, the Play Again and Instructions options are again disabled and control returns to the top of the outermost loop, and a new game is set up.

Now that you’ve read through all this, take a break. Why not play a game or two of MacGurkha?

Suggestions for MacGurkha

You can do a number of interesting things with MacGurkha for fun and practice. For a start, change the way the Gurkha falls with the chute closed (free-fall) to make it more realistic. Try using a trajectory like that in MacDart: Give the Gurkha the plane’s forward speed as his horizontal (x) velocity and use gravity (as in MacDart) for his falling (y) velocity. Because of air resistance, the Gurkha shouldn’t continue to accelerate indefinitely, but instead will tend toward a “terminal velocity.” You could make this happen by coding a limit for the y velocity or by using a smaller gravity constant than that in MacDart. Experiment!

Wind has an effect on the way a real parachutist falls, especially when the chute is open. You could modify OpenChute to include a randomly generated wind (which would remain constant for the duration of the jump). The wind would have a 50-50 chance of being to the left or the
right side of the screen and would be added as a positive or negative number to the Gurkha's x velocity. The wind speed and direction could be displayed at the bottom of the screen, or you could draw a waving flag to give a visual indication of the wind's direction and speed.

The interaction between the falling Gurkha and the blimps could be modified according to your expectation of how the game should reflect "reality." For example, you might want the Gurkha with an open chute to snag his chute on a guy wire of a balloon every time he tried to drift past one, instead of only occasionally. You would have to change the code in CheckBlimp in order to do a more extensive level of collision detection. This would slow the program down considerably, which could be offset by the fact that the Gurkha is drifting slowly in the wind. A change in the strategy of the program would be needed, however, since in many cases the landing zone is completely covered by balloons. This would be done by changing SetUpScreen so that balloons would never be drawn within a certain distance from the landing zone, but would not be too far away, since a totally clear landing path wouldn't be too exciting a goal to reach.

You can also, of course, change the way the scoring works. You could have the target zone shrink automatically after each successful jump (say by 10 percent) and the score bonus increase by 10 percent each time. A failed jump would reset the target zone to its normal size.
MacGurkha program listing

MacGurkha
Land the Gurkha in the jump zone!

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

INIT 1

RANDOMIZE TIMER
DEFINT a-z
DEF SNG q

DEF FN ArraySize(xWide,yHigh) = INT(1 + xWide / 16) * (yHigh + 1) + 2

planeH = 50 : planeV = 23
DIM airPlane(FN ArraySize(planeH,planeV))
openH = 23 : openV = 26
DIM openChute(FN ArraySize(openH,openV))
closedH = 14 : closedV = 9
DIM clsChuteP(FN ArraySize(closedH,closedV))

DIM rec(3), grayPat(3)
false = 0 : true = NOT false
maxLives = 5
altitude = 5
ground = 250
FOR n = 0 TO 3 : grayPat(n) = &HA55 : NEXT

WINDOW 1, "MacGurkha", (12,32) - (500,330), 2
GOSUB MakePictures

MENU 6, 0, 1, "MacGurkha"
MENU 6, 1, 0, "Play Again"
MENU 6, 2, 0, "Instructions"
MENU 6, 3, 0, "-"
MENU 6, 4, 1, "Quit to BASIC"
MENU 6, 5, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON

GOSUB TitleScreen
GOSUB Instructions

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

MAIN 1

WHILE true
lives = maxLives
jumps = 1 : score = 0
WHILE lives > 0
GOSUB SetupScreen
points = 0
qPlaneSpd = (RND(1) * 10) + 2
GOSUB FlyPlane

MAIN 2
FOR t = 1 TO 500 : NEXT
manX = plane
manY = altitude + planeV
hitBlimp = false : hitGround = false
GOSUB FreeFall
ffDist = manY - altitude
mCntrl = 0
IF NOT (hitBlimp OR hitGround) THEN GOSUB ChuteFall

NextJump:
GOSUB UpdateStatus
FOR t = 1 TO 5000 : NEXT
GOSUB ClearMouse
WEND

CALL Announce("GAME OVER")
playAgain = false
MENU 6, 1, 1
MENU 6, 2, 1
WHILE NOT playAgain : WEND
MENU 6, 1, 0
MENU 6, 2, 0
WEND

MenuEvent:
MENU 6, 0, 0
item = MENU(1)
ON item GOSUB PlayMenu, Instructions, , BasQuit, DeskQuit
MENU 6, 0, 1
RETURN

PlayMenu:
playAgain = true
RETURN

Instructions:
WINDOW 2, , (16,55) - (495,325), 2
PRINT "Instructions for MacGurkha"
PRINT "Your mission: to bail a Gurkha out of the airplane, avoid the balloon"
PRINT "obstacles, and guide him into the landing zone between the vertical bars.
PRINT "The planes fly across the screen at different speeds. To nudge the"
PRINT "Gurkha out of the plane, click the mouse button once. Since a Gurkha's"
PRINT "forward momentum depends on the plane's speed, you should eject him"
PRINT "sooner the faster the plane is travelling."
Once your Gurkha is out of the plane and free-falling, you can pop open his parachute by clicking the mouse button again. After the chute is open, you can guide the Gurkha by holding down the mouse button and dragging the mouse. Your score is determined by three factors: 1) The width of the landing zone, 2) how close the Gurkha is to the ground when you open his chute, and 3) how much you must use the mouse to guide the Gurkha into the landing zone.

```basic
PRINT "Once your Gurkha is out of the plane and free-falling, you can pop open his parachute by clicking the mouse button again. After the chute is open, you can guide the Gurkha by holding down the mouse button and dragging the mouse. Your score is determined by three factors: 1) The width of the landing zone, 2) how close the Gurkha is to the ground when you open his chute, and 3) how much you must use the mouse to guide the Gurkha into the landing zone."
```

```
PRINT "his parachute by clicking the mouse button again. After the chute is open,"
PRINT "you can guide the Gurkha by holding down the mouse button and dragging" PRINT "the mouse."
PRINT "Your score is determined by three factors: 1) The width of the landing"
PRINT "zone, 2) how close the Gurkha is to the ground when you open his chute, " PRINT "and 3) how much you must use the mouse to guide the Gurkha into the " PRINT "landing zone."
```

```
PRINT CALL INITCURSOR BUTTON 1, 1, "Continue", (190, 240) - (280, 260) WHILE DIALOG(O) <> 1 : WEND BUTTON CLOSE 1 : WINDOW CLOSE 2 CLS RETURN
```

```
Bas Quit: SUBROUTINES

ShowHit: ' gurkha hits blimp
FOR z = 0 TO 5
  SOUND 120, 3, 255 - (50 * z) NEXT IF hitBlimp THEN CALL Announce("You hit a blimp!") IF hitGround THEN CALL Announce("You hit the ground like a rock!") lives = lives - 1 RETURN

Madel: ' hurray! He hit the jump zone!
points = ffDist + 50 - (flag2 - flag1) - mCntrl / 3 IF points < 0 THEN points = 0 score = score + points : jumps = jumps + 1 FOR n = 1 TO 2 : SOUND 440,3 : SOUND 659,6 : NEXT CALL Announce("You made it!") RETURN

Casualty: ' landed outside of jump zone
lives = lives - 1 SOUND 20, 20 : CALL Announce("You missed!") RETURN

SetupScreen: ' set up screen for new jump run
LINE (0,0) - (520, ground + 3), 30, bf plane = 500 'clear playing field 'starting x-loc of plane
```
flag1 = INT(300 * RND(1)) + 25  
flag2 = flag1 + INT(50 * RND(1)) + 20  
LINE (flag 1 - 2, ground - 5) - (flag1,ground), , bf  
LINE (flag2, ground - 5) - (flag2 + 2, ground), , bf  
LINE (1,ground) - (500, ground + 3), , bf  
GOSUB UpdateStatus  
PUT (plane, altitude), airPlane  

FOR n = 1 TO jumps  
  obsX = INT(400 * RND(1)) + 1  
  obsY = INT((ground - altitude - 65) * RND(1)) + altitude + 25  
  rec(0) = obsY + 18 : rec(1) = obsX + 13  
  rec(2) = obsY + 23 : rec(3) = obsX + 27  
  CALL PAINTRECT(VARPTR(rec(0)))  
  rec(0) = obsY + 3 : rec(1) = obsX + 33  
  rec(2) = obsY + 17 : rec(3) = obsX + 45  
  CALL PAINTARC(VARPTR(rec(0)), 45, 90)  
  LINE (obsX - 2, obsY + 6) - (obsX - 2, obsY + 12)  
  PSET (obsX - 1, obsY + 9)  
  LINE (obsX + 20, obsY + 9) - (obsX + 20, ground)  
  CALL FILL OVAL(VARPTR(rec(0)), VARPTR(grayPat(0)))  
  CALL FRAME OVAL(VARPTR(rec(0)))  
NEXT  
RETURN

FlyPlane:  
WHILE MOUSE(0) > -1  
  plane = plane - qPlaneSpd  
  IF plane < -planeV THEN plane = 500  
  PUT (plane, altitude), airPlane  
  SOUND 20 + qPlaneSpd, 1  
  PUT (plane, altitude), airPlane  
WEND  
RETURN

FreeFall:  
WHILE NOT hitBlimp AND MOUSE(0) > -1 AND manY + closedV < ground  
  SOUND 3500 - 10 * manY, 1  
  PUT (manX, manY), clsChuteP  
  manY = manY + 4  
  drift = INT (3 * RND(1))  
  IF RND(1) < .5 THEN manX = manX - drift ELSE manX = manX + drift  
  manX = manX - qPlaneSpd : qPlaneSpd = qPlaneSpd * .9  
  IF manX < 0 THEN manX = 500  
  PUT (manX, manY), clsChuteP  
GOSUB CheckBlimp
MAC GURKHA

WEND
IF manY + closedV >= ground THEN hitGround = true
IF NOT (hitBlimp OR hitGround) THEN PUT(manX, manY), clsChuteP
RETURN

Chute Fall: loop for floating open chute
hotY = openV: hotX = openH / 2
PUT (manX, manY), openChute
WHILE manY + hotY < ground AND NOT hitBlimp
   PUT (manX, manY), openChute
   'draw initial chute
   'erase existing chute
   'add mouse control
   IF MOUSE(0) < 0 THEN manX = manX + SGN(MOUSE(1) - manX) * 3: mCntrl = mCntrl + 1
   drift = INT(7 * RND(1))
   IF RND(1) < .5 THEN manX = manX - drift ELSE manX = manX + drift
   manY = manY + 3
   'go down a little
   IF manX < 0 THEN manX = 500
   PUT (manX, manY), openChute
   FOR t = 1 TO 50: NEXT
   GOSUB CheckBlimp
   'draw new chute
   'small delay to slow chute
WEND
RETURN

CheckBlimp: check to see if man has hit blimp
IF manY + hotY > ground - 31 THEN RETURN
IF POINT(manX + hotX, manY + hotY) = 33 THEN hitBlimp = true
RETURN

ClearMouse: clear mouse event queue
WHILE MOUSE(0) <> 0: WEND
RETURN

UpdateStatus: update status displays
CALL MOVETO(320, ground + 20): PRINT "Points for Jump: " points
CALL MOVETO(320, ground + 36): PRINT " Total Score: " score;
CALL MOVETO(2, ground + 20): PRINT "Extra"
CALL MOVETO(2, ground + 36): PRINT "Gurkhas"
FOR n = 1 TO lives - 1
   PUT(55 + n * (openH + 5), ground + 10), openChute, PSET
NEXT
IF lives < 1 THEN NoneLeft
onePast = 55 + lives * (openH + 5)
LINE(onePast, ground + 10) - (onePast + openH, ground + 10 + openV), 30, bf
NoneLeft:
RETURN

TitleScreen: display title screen
CLS
CALL TEXTFONT(0): CALL TEXTSIZE(36)
LOCATE 1,1 : PRINT "MacGurkha"
LOCATE 2,1 : CALL TEXTSIZE(12)
PRINT "Land in the target zone."
pH = 500 : mY = 150
WHILE pH > 0
    PUT (pH,100) - (pH + 100,150), airPlane
    SOUND 30,1
    IF pH > 300 THEN NoFall
    PUT (pH,mY) - (pH + 20, mY + 20), clsChuteP
    PUT (pH,mY) - (pH + 20, mY + 20), clsChuteP
    mY = mY * 1.03
NoFall:
    PUT (pH,100) - (pH + 100, 150), airPlane
    pH = pH - 10
    IF MOUSE(0) <> 0 THEN pH = -1
WEND
RETURN

MakePictures:
    ' -------------------- make picture arrays
    ' if MacGurkha.Graphics exists, read it, else read DATA

fileHere = true
    ' On error will change to false if not here.
ON ERROR GOTO FileError
    ' set up error trapping
OPEN "MacGurkha.graphics" FOR INPUT AS #1
    ' is file present?
ON ERROR GOTO 0
    ' disable error trapping
IF NOT fileHere THEN ReadPictures
    ' no, read from DATA
CLS
INPUT #1, byteCount
image$ = INPUT$(byteCount,1)
PICTURE(0,0), image$
GET(0,0) - (planeH - 1, planeV - 1), airPlane
CLS
image$ = INPUT$(1,1)
INPUT #1, byteCount
image$ = INPUT$(byteCount,1)
PICTURE(0,0), image$
GET (0,0) - (closedH - 1, closedV - 1), clsChuteP
CLS
image$ = INPUT$(1,1)
INPUT #1, byteCount
image$ = INPUT$(byteCount,1)
PICTURE(0,0), image$
GET (0,0) - (openH - 1, openV - 1), openChute
CLS
CLOSE 1
GOTO BuildPictEnd
ReadPictures:
    ' MacGurkha.Graphics isn't on disk; read from DATA
RESTORE AirPlaneData : CALL ReadArray(airPlane())
RESTORE clsChutePData : CALL ReadArray(clsChuteP())
RESTORE OpenChuteData : CALL ReadArray(openChute())
BuildPictEnd:
RETURN
FILE ERROR: set FileHere to false if file not found

IF ERR <> 53 THEN ON ERROR GOTO 0
fileHere = false
RESUME NEXT

SUB Announce(temp$) STATIC
    display given string
    SHARED altitude
    DIM r(3)
r(0) = 0 : r(1) = 0 : r(2) = altitude + 20 : r(3) = 500
    CALL ERASERECT(VARPTR(r(0)))
    CALL MOVETO(250-(LEN(temp$) / 2 * 8), altitude + 15)
    PRINT temp$
    ERASE r
END SUB

SUB ReadArray(array(1)) STATIC
    read given array from DATA statements
    READ size
    FOR n = LBOUND(array) TO LBOUND(array) + size - 1
        READ num : array(n) = num
    NEXT
END SUB

AirPlaneData: DATA for airplane picture
DATA 95
DATA 50, 23

DATA 511, -2, 0, 0, 512, 12, 14, 0, 511, -16
DATA 18, 0, 32, 1024, 18, 0, 16, 10240, 34, 0
DATA -32760, -19968, 66, 0, -32763, -18936, 130, 0, -30750, -1521
DATA -254, 0, -30689, -16, 2, 0, -26752, 0, 2, 0
DATA -28672, 28, 2047, 0, -2176, 127, 2048, -32768, -28672, 28
DATA 2047, 0, -26752, 0, 48, 0, -30689, -1, -64, 0
DATA -30748, 16384, 48, 0, -32766, -32768, 40, 0, -32767, 0
DATA 16, 0, 3, -32768, 0, 0, 4, 16384, 0, 0
DATA 5, 16384, 0, 0, 4, 16384, 0, 0, 3, -32768
DATA 0, 0, 0

ClsChutePData: DATA for closed chute picture
DATA 12
DATA 14, 9

DATA 8192, 4096, 2168, 28552, -128, 28664, 2056, 4096, 8192, 0

OpenChuteData: DATA for open chute picture
DATA 55
DATA 23, 26

DATA 254, 0, 853, -32768, 3413, 24576, 5461, 20480, 13653, 22528
DATA 21845, 21504, 21845, 21504, -10923, 22016, -10923, 22016, -1, -512
DATA 16384, 1024, 8192, 2048, 4096, 4096, 2048, 8192, 1040, 16384
DATA 568, -32768, 313, 0, 403, 0, 511, 0, 56, 0
DATA 56, 0, 56, 0, 68, 0, 68, 0, 68, 0
DATA 198, 0, 0
MacSpace

Life is tough in today's Space Navy. You've got plush bucket seats, factory air, and a refrigerator crammed with edible and drinkable goodies. Plus, you've got thousands of microdisks filled with the galaxy's best computer games and other entertainment. Trouble is, just when you've settled down for an evening's fun, the blasted Klaxxon alarm goes off and you've got to strap
How to play MacSpace

When you load MacSpace and choose Start from the Run menu, the title screen shown in Figure 1 appears. After a little action on the title screen, the screen of instructions in Figure 2 comes up. After you’ve read them, click the Continue button at the bottom of the screen to get on with the action. As you watch, your Mac becomes the viewscreen of a star cruiser looking out into a sky full of stars, as in Figure 3.

Note that the number of enemies shot is shown at the top of the screen, as is the number of photon torpedoes remaining in your tubes (you start with five, and there’s no replacement service out where you are!). At the top left of the screen you’ll also see a number of little star cruisers representing the number of player ships (that is, “lives”) you have left (not counting the one you’re currently using). At the start of the game, you have five cruisers: the one whose viewport you’re stargazing from, plus the four others shown at the top of the screen.
Welcome to MacSpace

The object is to kill as many enemy Klaxxon ships as you can before they destroy all of your ships. There are two ways to destroy the Klaxxons. The first is by shooting them with your lasers by putting the cross hairs over the Klaxxon ship and clicking the mouse button. Every time you make a hit, the Klaxxon ship will get smaller. Shrink the Klaxxon ship small enough and it explodes. You have an unlimited number of laser shots.

The second way to kill the Klaxxon is to aim with the cross hairs, then fire a photon torpedo by pressing any key. No matter what the Klaxxon's size, if you hit the ship with a torpedo it will explode. Be aware that every time you destroy a Klaxxon ship, the next one is faster and trickier.

FIGURE 2. The MacSpace instruction screen

FIGURE 3. The MacSpace battle screen
Enemy ships appear one at a time (following the provisions of the Aldebaran Treaty of 2243). As each Klaxxon cruiser approaches, it looms larger and larger in your viewscreen. If the enemy cruiser gets too close, you'll hear a warning sound, and soon the foul Klaxxons are likely to be shooting at you. Confederation scientists haven't figured out how the enemy's weapons work, but you know you've been hit when the whole sky flashes. Every time that happens, you lose a player ship. If you lose all five of your cruisers, the game is over.

**Weapons systems**

You have two ways of attacking the Klaxxons: lasers and photon torpedoes. You fire the lasers by positioning the cross-hair cursor over the enemy cruiser and then clicking the mouse button to fire. Two beams from your laser guns (which are in the lower left and right corners of the screen) will converge on the target. If you hit the vile Klaxxon cruiser, it shrinks away in cowardly retreat. Repeated hits make the enemy ship retreat farther, but it keeps trying to come closer between shots. Remember that if it gets close enough it can destroy you.

If you make the enemy cruiser go far enough away, it will be destroyed—you'll see it blow up. To ensure that you don't join the ranks of the unemployed, however, a replacement Klaxxon cruiser will be sent your way immediately at no charge to you. It'll appear and approach you from a random location. To make life even more interesting, each new enemy cruiser moves a little faster and gets bigger more quickly than its predecessor. (Isn't it nice to know that you're helping the Klaxxons develop better ships?)

Luckily, you have a second way of combating the slimy forces of evil. If a Klaxxon cruiser is getting big and close and starting to cause serious trouble, you can fire a photon torpedo at the cross-hair location by hitting any keyboard key. When you do, you'll see the two, spike-covered components of the torpedo emerge from the torpedo tubes (which are the same dual-purpose tubes that fire your lasers) and converge on the aiming point. If the torpedo hits, it destroys the enemy cruiser regardless of the ship's size. But don't waste your torpedoes—remember that you've only got five per game!

**Battle tactic**

As you've probably figured out by now, the object of the game is to destroy as many Klaxxon star cruisers as possible before you run out of ships.
Keep an eye on the top of the screen to note how many player cruisers and photon torpedoes you have left, as well as the number of enemies you have already destroyed.

The best tactic for defeating a slimy Klaxxon is to get the cursor over the enemy cruiser and then click as rapidly as possible, shifting the cursor as necessary to keep the enemy in your sights. Slow shooting just gives the rotten Klaxxons time to recover and grow bigger. As the enemy cruisers get faster, however, there may not be enough time to get the cursor over the Klaxxon ship and keep it there. That's when knowing where to intercept the onrushing enemy cruiser becomes important.

Here's another trick: If you find a big Klaxxon cruiser shooting at you and your cursor is way out of position, fire a photon torpedo anyway. Even though the torpedo will miss, the enemy freezes while it's in flight. You can use that time to get the cursor over the startled Klaxxon and fire lasers or another torpedo at it as soon as the first torpedo has finished its flight. This tactic is wasteful of resources, but it just might save your life.

When the Klaxxon cruisers are relatively slow, an occasional photon torpedo released at the right time can be effective; but as the enemy spaceships get faster, it becomes harder to hit them with the relatively slow-moving torpedoes. You should probably save one or two torpedoes for emergency situations, such as when a Klaxxon has gotten big enough to fill the screen and is shooting at your last ship.

You should also keep in mind another nasty trick the Klaxxons have: When a Klaxxon cruiser comes close enough, it can get in under your viewscreen so that you can't see it or shoot at it. Unfortunately, it can still shoot at you. This doesn't happen too often, but when it does, all you can do is keep the cursor down at the bottom of the screen and shoot when part of the ship pops up.

When you've finished a game, you can pull down the MacSpace menu and choose Play Again. At any time during or after a game, you can choose Quit to BASIC or Quit to Desktop to end the game and return to the selected environment.

How MacSpace works

As you can see in the program outline in Figure 4, after the arrays and constants are initialized, the menu installed, and the mouse interrupt for firing the lasers enabled, the program enters a WHILE...WEND loop and the game action begins. Laser fire is handled by a mouse interrupt. At the end of the game, the Play Again menu option is made available so that the player can play again or quit.
Initialization
- define size of graphics and cursor arrays
- set constants for logical variables, screen locations
- create playing window
- initialize special cross hairs cursor
- create and save ship shapes (MakeShips)
- create and save torpedo shape (MakePhoton)
- display title screen (TitleScreen)
- show instructions (Instructions)
- install MacSpace menu, enable menu event-trapping
- enable mouse event-trapping

Main Program
- WHILE true (endless loop interrupted by menu events)
  - initialize torpedoes, score, ships
  - set target cursor
  - draw playing screen (DrawView)
  - place initial enemy (NewEnemy)
  - WHILE ships left
    - handle mouse events, then queue them (FireLasers)
    - if key pressed, fire torpedo (FirePhoton)
    - calculate enemy's new position (MoveEnemy)
    - display enemy's new position (UpdateEnemy)
    - if enemy close enough, sound warning (Warning)
    - if close enough, randomly determine if enemy fires (EnemyShot)
  - WEND
  - announce end of game
  - wait for menu event to restart game
- WEND

Menu Events
- MenuEvent: branch to appropriate subroutine
- PlayMenu: set flag for playing again
- BasQuit: quit to BASIC
- DeskQuit: quit to Mac Desktop

Subroutines
- UpdateEnemy: update enemy's location on screen
- MoveEnemy: calculate where enemy should be next
- EnemyShot:
  - do sound effect
  - determine if enemy scores a hit
  - if so, remove player's ship (PlayerKilled)
- PlayerKilled:
  - flash entire window, do sound effect
  - subtract a life (DrawExtra)

FIGURE 4. Program outline
Warning: do sound effect for enemy too close

FireLasers:
  if laser hits, make enemy smaller (CheckHit, UpdateEnemy)
  if enemy small enough, destroy it and replace (EnemyKilled)

FirePhoton:
  get current pointer coordinates
  if no torpedoes left, do sound effect and return
  calculate path of torpedoes, animate them, do sound effect
  check for torpedo hit (CheckHit)
  if so, destroy enemy (EnemyKilled)
  display number of torpedoes left (UpdatePhotons)

CheckHit: determine if pointer is at location of enemy

EnemyKilled:
  increment score
  explode enemy (ExplodeEnemy)
  show new score (UpdateScore)
  place new enemy (NewEnemy)

ExplodeEnemy:
  invert oval around enemy
  make four expanding circles, do sound effect
  restore enemy to original condition and erase it

NewEnemy:
  generate random location for enemy ship
  each new enemy grows and moves faster
  establish initial direction

DrawView:
  display ships remaining (DrawExtra)
  display enemy ships destroyed, torpedoes remaining
  draw black sky, fill with stars

UpdateScore: display updated score

UpdatePhotons: display number of torpedoes remaining

TitleScreen: draw title screen

Instructions: display initial instructions

MakePhoton: create and save shape of photon torpedo

MakeShips:
  load shapes from disk, or read from arrays (ReadArray)

Subprograms
  DrawExtra: draw or erase one of player's ships
  ReadArray: read graphics array from DATA statements

Figure 4. Program outline (continued)
Now let's take a look at the structure and operation of the program in more detail.

Initialization and constants

First, arrays are dimensioned [Init 1] and constants are assigned their values. Note here the use of DEFINT and DEFSNG. These statements set all the variables to integer, except for variables starting with a q, which will be single precision. Next, the coordinates of the combination laser guns/photon torpedo tubes and the screen boundaries are initialized. The WINDOW statement sets the boundaries of the Output window. Other constants give the number of stars, player ships, photon torpedoes, factors used in determining hits, and the aspect ratio (ratio of height to width) for drawing the Klaxxon ship.

Since this is a “shoot 'em up” game, the program initializes a new cross-hair gunsight cursor in [Init 2] by using two FOR...NEXT loops and some additional statements that put bit patterns into the array crossHairs. The mask for this cursor (in elements 17 through 32 of the crossHairs array) is set to 0 so the screen pixels under the cursor will be inverted. This will display the cursor as white against the black background. The “hot spot”—the part of the cursor that will be read as the actual cursor location—is set to the center (8,8) of the cross hairs. This is where the gunsight will be aiming when the mouse button is pressed. The actual creation of the new cursor from the initialized pattern will be done when the main game loop begins. (You could change this section to allow use of cursors created with MacCursor.)

GETting the picture

The code at [Init 3] supplies needed bit-mapped picture data to the arrays for the images of the enemy and player ships and the photon torpedo. It begins by calling MakeShips [SR 18]. This subroutine first checks the disk for a file named MacSpace.graphics. If you have created your own custom graphics for MacSpace, they should be in this file. (The MacInterface chapter contains complete instructions for creating graphics in MacPaint and using them in our games and your own BASIC programs, and includes an example of alternate ships for MacSpace.)

After setting up an ON ERROR GOTO statement to trap file errors, the program attempts to open the graphics file. If an error is found, control goes to FileError [SR 18c]. The most likely error that can occur is File Not Found, which has a code number of 53 (see Appendix B of the
BASIC manual for a complete list of error messages and codes). If this error turns up, it means that the MacSpace.graphics file is not on the disk. If any other kind of error is found, the program halts. If the file isn’t present, the fileHere flag is set to false.

If the file was found, the encoded QuickDraw graphics commands for the picture of the enemy star cruiser are read into the string-variable image$. The PICTURE statement is used to execute these graphics commands and draw the picture on the screen. The GET statement then reads the picture from the screen and stores it in the array ePict. The same procedure is followed for obtaining the picture of the player ship, which ends up in the array pPict.

Because the GET statement stores a picture as an array of bits, and the PUT statement will be used later to transfer the bits directly to the screen, fast animation becomes feasible. These statements draw graphics much more quickly than is possible with a series of BASIC graphics statements, or even the PICTURE statement with a string.

If the file is not found, fileHere is set to false and ReadPictures [SR 18b] calls the general-purpose subprogram ReadArray [SP 2] to read the numbers defining the screen bit image directly into the appropriate arrays. We provide this method of getting the pictures so you can just type the program and run it without having to create special graphics.

The player’s star cruiser will merely make a cameo appearance on the title screen and will then be used at the top of the playing screen to indicate the number of ships the player has left during play. The only objects representing the player on the main screen are the view window, the cross-hair cursor, and the guns.

After MakeShips finishes, the code at [Init 3] calls a second subroutine, MakePhoton [SR 17]. This subroutine uses the QuickDraw PaintOval routine to draw the torpedo body, with the two LINE statements providing the four “spikes” protruding from the torpedo. Again, a GET statement stores this picture from the screen to an array, photonPict.

The program next takes care of drawing the title screen by calling two more subroutines. The first, TitleScreen [SR 15], includes a simulated battle sequence resulting in the destruction of the enemy ship. The laser fire is animated by the FOR...NEXT loop, which READs the end points for the lines of the laser beams from DATA statements and gives the effect of fire homing in on the Klaxxon cruiser. The second subroutine, ExplodeEnemy [SR 10], then blows up the enemy. After a short delay, [Main 3] calls the second subroutine, Instructions [SR 16], which displays the on-screen instructions for playing the game.
Next, the MacSpace menu is installed. The MENU statements title the menu and create three possible options: Play Again, Quit to BASIC, and Quit to Desktop. The ON MENU GOSUB statement causes all menu choices to call MenuEvent [ME 0]. This subroutine first makes the whole menu inactive (the last 0 in the MENU statement does this). It’s important to make a menu inactive while a menu event is being handled so the user can’t make conflicting choices, such as choosing an item repeatedly because “nothing has happened yet.” Similarly, particular menu items should be deactivated when they would be inappropriate selections. For example, before play begins, the menu is active but the option Play Again is inactive (note that last 0). After all, you can’t “play again” when you haven’t played at all yet. The two quit options (Quit to BASIC and Quit to Desktop) will always be active when the menu itself is active, since the player might want to quit at any time, even in the course of play.

MenuEvent gets the numeric value of the selection from the MENU function (with a parameter of 1) and assigns the selection number to item. The ON item GOSUB statement then calls the appropriate subroutine to perform the action requested.

Events and interrupts: Keys to game design

An interrupt is an action that suspends whatever process is going on at the time of interruption and sends control to a specified part of the code (usually a subroutine). An event is an action that the system treats as a request for an interrupt. The Macintosh handles three main kinds of interrupts:

- Menu interrupts: The user has chosen an active item from an active menu.
- Dialog interrupts: The user has clicked a button in a dialog window, changed edit fields, or done something with a window.
- Mouse interrupts: The user is performing some sort of action with the mouse button (single- or double-clicking, dragging, and so on).

BASIC provides an ON [event] GOSUB statement to send control to a “handler” subroutine when the appropriate event occurs. (The actual statements would be ON MENU GOSUB, ON MOUSE GOSUB, and so on.) The handler subroutine then uses the appropriate function to determine exactly what kind of selection was made (whether the mouse button was single- or double-clicked, for example). A further ON...GOSUB is usually used to actually perform the action requested. We just saw an example of this structure in our MacSpace menu.
Just before the game action begins in [Init 3], an ON MOUSE GOSUB statement specifies that FireLasers [SR 6] be called if the mouse button is pressed. Since this is the only button press processed in this game, we don’t need an intermediate handling routine to sort out what has happened. But are you wondering why we have set up an interrupt rather than just using an IF...THEN statement in the main game loop to check whether the button was pushed?

The use of an interrupt is crucial for fast games that have a number of statements being executed in the main game loop. If we simply “poll” the mouse button (check it once in a while) with an IF MOUSE(0) statement, looking for a 1 to be returned indicating that the button was clicked, the program can’t respond to the player’s pushing the mouse button until the loop comes around to the IF statement—which might be a while. In the meantime, the enemy is still moving and the game situation continuing to change. The ON MOUSE GOSUB statement sets up the program to “invisibly” and constantly check the mouse button, thereby guaranteeing that wherever the program is in the loop, a mouse click will bring instant response—which is what you’d expect from a laser gun. With the polling approach, you might get an agonizing delay before the gun is fired. The photon torpedo, on the other hand, is activated by an IF statement that finds when a key has been pressed. Since photon torpedoes are used relatively rarely, and are traditionally slower than lasers anyway, an interrupt is not needed here. A good general rule for designing interactive games is to use an interrupt when response to player input must be immediate, regardless of what’s going on at the time. It’s also useful when you don’t want a loop to be slowed down by polling the mouse; for example, when a loop generates a musical tone.

The main program loop

We now come to the heart of the program [Main]. The number of photon torpedoes and the total number of ships allotted to the player is set to the maximum values specified in the constants maxPhotons and maxShips. The SETCURSOR statement turns the cursor into the cross-hair shape that was defined in the array crossHairs. The view of space as seen from the player’s starship viewscreen is created by a call to DrawView [SR 12]. This subroutine first calls DrawExtra [SP 1] to draw the four player ships in reserve. The strings Enemies Shot: and Photon Torpedoes Left: are printed to set up the “scoreboard” at the top of the screen, and the UpdateScore [SR 13] and UpdatePhotons [SR 14] subroutines are called to provide the starting values of 0 and 5, respectively.
The LINE statement with the \textit{bf} option fills the entire window with the black void of space. Since we want objects drawn as white against this black background, a pen transfer mode of XOR (exclusive OR) is used for drawing. This has the desired effect of inverting objects and making them white on black.

The two guns are drawn by setting up temporary rectangles around their coordinates and using calls to the QuickDraw \texttt{PAINTOVAL} routine to draw the gun tubes. Finally, a sky full of stars with random coordinates and sizes ranging from 1 to 3 pixels in diameter is created. After returning from DrawView, the outer loop in the main program calls NewEnemy [SR 11], which draws an enemy ship with a random position and direction, but heading toward the player's ship.

An inner \texttt{WHILE...WEND} loop [Main 2] now resolves the battles between the player and successive Klaxxon ships. As the enemy cruiser approaches, BASIC automatically checks for a mouse interrupt and calls the FireLasers [SR 6] if one occurs. The \texttt{MOUSE STOP} statement following \texttt{MOUSE ON} prevents FireLasers from being executed in the middle of the graphics-initialization stage, which would interfere with the graphics. \texttt{MOUSE STOP} records any mouse event but suspends execution of the appropriate \texttt{GOSUB} until the next time \texttt{MOUSE ON} is executed back at the top of the inner loop. This gives us the benefit of interrupts—being always able to detect player activity—without having to deal with possible interference between an interrupt and another routine that is executing when the interrupt is detected.

If there is a button press, FireLasers [SR 6] then gets the current mouse-pointer (x,y) coordinates (more precisely, the coordinates of the defined hot spot at the center of the cross hairs) using the \texttt{MOUSE(1)} and \texttt{MOUSE(2)} functions and then draws lines representing laser fire from the fixed gun coordinates \((\text{gun1X,gun1Y}), \text{and (gun2X,gun2Y)}\) to this spot, using calls to the QuickDraw \texttt{MOVETO} and \texttt{LINETO} routines. FireLasers then calls CheckHit [SR 8] to check whether the hot spot is within the last recorded boundaries of the Klaxxon spaceship: If so, a hit was made. In this case, FireLasers makes the enemy ship smaller by the current value of \texttt{eSizeDec}. It then calls UpdateEnemy to show the new, more compact model of Klaxxon star cruiser. FireLasers then checks the new size of the enemy; if it's less than the value \texttt{eHitSize}, the enemy has been destroyed (we'll see how this works in a moment).

Next in the loop, \texttt{INKEY$} peeks at the keyboard buffer. If the returned string is anything other than a null string, a key has been pressed.
A photon torpedo will therefore be fired (via FirePhoton [SR 7]), assuming there’s at least one torpedo left in the player’s arsenal.

FirePhoton [SR 7] works in a similar way. The current mouse x and y coordinates are obtained, and the supply of torpedoes is checked. If there aren’t any left, a “plinking” sound tells the player so, and the subroutine is exited. Otherwise, the number of photon torpedoes is reduced by one. Next, the slopes from the gun coordinates to the hot spot are calculated, and the little shapes representing the torpedo are animated (drawn, erased, and redrawn) along these slopes, giving the effect of the torpedo moving toward the target. CheckHit [SR 8] is then called, and if the enemy spaceship is hit it is destroyed via EnemyKilled [SR 9]. Whether the player hits or misses, UpdatePhotons [SR 14] is called to display the number of photon torpedoes remaining.

If either FireLasers or FirePhoton results in the enemy being destroyed, EnemyKilled [SR 9] is called. This subroutine adds one to the total of enemy ships destroyed and then calls ExplodeEnemy [SR 10], which creates the explosion by first using the QuickDraw INVERTOVAL routine to invert the enemy ship and then FRAMEOVAL to make the four expanding circles that represent the explosion. The hapless Klaxxon is erased, and we then return to EnemyKilled.

EnemyKilled next calls UpdateScore [SR 13], which simply displays the new score. Finally, NewEnemy [SR 11] is called, and a new enemy ship is set up with a random starting location and direction (there is an endless supply of enemy ships). To keep the game interesting, the variables eSizeInc and eMoveInc are increased by a fixed amount so that the new enemy ship will both move faster and get bigger faster than previous ships. Since eSizeDec (the amount by which the ship “shrinks” when hit) is inversely proportional to eSizeInc, the number of laser hits needed to destroy the enemy ship at a given size decreases during the game, to compensate a little for the added difficulty of hitting the target.

After checking for and resolving the player’s attacks, the inner play loop moves the enemy ship with MoveEnemy [SR 2] and updates its position with UpdateEnemy [SR 1]. The enemy ship is “jumped” rather than smoothly animated. (Perhaps the Klaxxons discovered that they lost fewer ships if they installed a computer-controlled miniwarp drive along with their impulse engines to make their course less predictable.) If the previous Klaxxon ship was destroyed by the player’s fearless bombardment, this will be the replacement enemy ship. The size of the enemy spaceship is then checked; if it is large enough, Warning [SR 5] is called and a warning sound is made to alert the player.
Once the Klaxxon ship is close enough for the warning to be sounded, there is a chance that it will fire at the player. This chance is determined by the variable \textit{qEFireProb}. Enemy fire is handled by \textit{EnemyShoot} [SR 3], and the chance of a hit is controlled by the variable \textit{qEHitProb} in that subroutine (remember, we’ve defined variables beginning with \textit{q} to be single precision). If the player is hit, \textit{PlayerKilled} [SR 4] flashes the whole window by inverting it (the graphics aren’t erased, just inverted), produces an accompanying sound (a downward sweep of notes), and reduces the count of remaining player cruisers by one. In this case, \textit{DrawExtra} draws a number of ship symbols at the top of the screen equal to the number of player ships remaining.

We now return to the top of the inner game loop. If the player’s last ship has been destroyed, control falls into the bottom of the outer loop [Main 3]. There, the cursor is changed back into the standard arrow pointer, and the \textit{playAgain} flag is set to \textit{false}. The MacSpace menu is activated (note the last 1 in the \textit{MENU} statement) and a \textit{WHILE…WEND} loop waits for a menu interrupt, which at this point can be a selection of Play Again or one of the two quit options. If Play Again is selected, \textit{PlayMenu} [ME 1] is called. The \textit{PlayMenu} routine simply sets \textit{playAgain} to \textit{true}. Then the \textit{WHILE…WEND} loop is exited, control returns to the top of the main loop, and another round of play with a full supply of player ships and photon torpedoes is set up.

\textbf{Suggestions for MacSpace}

Readers who have designed arcade-type games for other machines will notice one task this program \textit{doesn’t} have to do: update the player’s gunsight position and redraw it as it is moved. This task is avoided because the gunsight is just our everyday Mac mouse pointer in disguise, and its display is taken care of by the system as usual without any programmer effort being needed. On most other systems you would have to read joystick or paddle coordinates, possibly convert them to the correct screen coordinates, and erase and redraw the gunsight. You would probably have timing problems as well, caused by the unavailability of interrupts.

We’ve already discussed how the judicious use of interrupts enables us to be constantly aware of player actions and control the timing of the program’s response. MacSpace illustrates how the use of interrupts and powerful graphics routines makes it much easier to write interactive games in the Macintosh environment.
The following are some suggestions for customizing MacSpace. First, the chance of an enemy shot hitting the player doesn't vary with distance. To make the game a bit more realistic, you could have EnemyShot check the current size of the Klaxxon ship (as an indication of relative distance) and increase the chance of its hitting the player as the enemy cruiser gets bigger (and thus closer). You could also randomize the number of player ships and/or photon torpedoes. As an even more ambitious modification, you could have the program create and manage several enemy ships at once. You could also provide some new and more fiendish weapons for the Klaxxon demons if you find that the game's getting too easy. For example, you could design enemy torpedoes that can hit your ship (represented by the cursor position) and have a chance of disabling your laser guns or photon torpedo tubes or blanking out the viewscreen for a random number of seconds.

Finally, really amazing effects could be created by using MacAnimate to create an animated sequence of images to use when the Klaxxon is blown up. Laser hits could be represented by debris flying off the affected part of the ship, and a torpedo hit by total destruction of the craft by blowing it to smithereens!
MacSpace program listing

---

MacSpace
Zap the spaceship before it zaps you!

---

INIT 1

DEFINT a-z : DEF SNG q
OPTION BASE 1
RANDOMIZE TIMER
DIM ePict(150), pPict(150) 'picture of enemy and player ships
DIM photonPict(50) 'picture of photon torpedo
DIM viewRect(4) 'holds coordinates of view rectangle
DIM t1Rect(4), t2Rect(4) 'temporary rectangles
DIM crossHairs(34) 'holds cross-hairs' cursor definition

false = 0 : true = NOT false
penCopy = 0 : penXor = 10
gun1X = 40 : gun1Y = 320
gun2X = 460 : gun2Y = 320
scrnTop = 35
fSteps = 15
stars = 60
maxPhotons = 5
maxShips = 5
eHitSize = 10

qEAspect = .63
qEFireProb = .2
qEHitProb = .2

WINDOW 1, , (1,20) - (511,342), 3 'put current window size in viewRect

viewRect(1) = scrnTop 'top is top of view screen
viewRect(2) = 0 'left is always (0,0)
viewRect(3) = WINDOW(3) 'find out bottom coordinate
viewRect(4) = WINDOW(2) 'and right one

FOR n = 1 TO 16 : crossHairs(n) = &H80 : NEXT 'cursor
FOR n = 6 TO 12 : crossHairs(n) = &H4081 : NEXT

crossHairs(1) = &H7F0

crossHairs(9) = &H7FFF

crossHairs(16) = &H7F0

FOR n = 1 TO 16 : crossHairs(n + 16) = 0 : NEXT 'and mask

crossHairs(33) = 8 : crossHairs(34) = 8 'set hot spot

GOSUB MakeShips 'make two ship picture arrays
GOSUB MakePhoton 'and photon torpedo picture
GOSUB TitleScreen
FOR t = 1 TO 5000 : NEXT
GOSUB Instructions

MENU 6, 0, 1, "MacSpace"  'install MacSpace menu
MENU 6, 1, 0, "Play Again"
MENU 6, 2, 0, "."
MENU 6, 3, 1, "Quit to BASIC"
MENU 6, 4, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON
ON MOUSE GOSUB FireLasers  'all mouse events call FireLasers

' * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * MAIN PROGRAM

WHILE true
    photonsLeft = maxPhotons
    ships = maxShips : score = 0
    eSizeInc = 0 : eMoveInc = 0
    SETCURSOR(VARPTR(crossHairs(1)))  'make cursor a cross hairs
    GOSUB DrawView  'draw player's view
    GOSUB NewEnemy  'place initial enemy
    WHILE ships <> 0  'loop while player has ships left
        MOUSE ON : MOUSE STOP  'handle and then queue mouse events
        IF INKEYS <> "" THEN GOSUB FirePhoton
        GOSUB MoveEnemy
        GOSUB UpdateEnemy
        IF eSize > 50 AND eSize - eSizeInc < 55 THEN GOSUB Warning
        IF eSize > 50 AND RND(1) < qEFireProb THEN GOSUB EnemyShot
    WEND

    CALL INITSIZE : CALL TEXTFACE(9)
    CALL MOVETO(120,140) : PRINT "END OF BATTLE"
    CALL TEXTMODE(O) : CALL TEXTSIZE(12) : CALL TEXTFACE(O)
    playAgain = false
    MENU 6, 1, 1  'enable Play Again menu item
    WHILE NOT playAgain : WEND  'loop until menu interrupt
    MENU 6, 1, 0  'disable Play Again
    WEND

' * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * MENU EVENTS

MenuEvent:
    MENU 6, 0, 0
    item = MENU(1)
    ON item GOSUB PlayMenu, BasQuit, DeskQuit
    MENU 6, 0, 1
    RETURN

PlayMenu:  '________________________________ user has chosen to play again
    playAgain = true
    RETURN
BasQuit: user wants to quit to BASIC

WINDOW CLOSE
MENU RESET
END

DeskQuit: quit to Mac Desktop

MENU RESET
SYSTEM

--- SUBROUTINES ---

UpdateEnemy: update enemy's location on screen

\[
\text{endX} = \text{enemyX} + \text{eSize} : \text{endY} = \text{enemyY} + \text{INT}(\text{eSize} \times \text{qEAspect})
\]

PUT(oldX,oldY) - (oldEndX,oldEndY), ePict  
\quad \text{'place new enemy,}

PUT(\text{enemyX},\text{enemyY}) - (\text{endX},\text{endY}), \text{ePict}
\quad \text{'and erase old}

oldX = \text{enemyX} : oldY = \text{enemyY} : oldEndX = endX : oldEndY = endY

RETURN

MoveEnemy: calculate where enemy should be next

\text{enemyX} = \text{enemyX} + \text{eMoveInc} \times \text{eXDir}
\text{enemyY} = \text{enemyY} + \text{eMoveInc} \times \text{eYDir}

\text{IF} \text{enemyX} < 1 \text{ OR } \text{enemyX} > 400 \text{ THEN} \text{eXDir} = \text{eXDir} \times -1 : \text{enemyX} = \text{oldX}
\text{IF} \text{enemyY} < \text{scrnTop} \text{ OR } \text{enemyY} > 300 \text{ THEN} \text{eYDir} = \text{eYDir} \times -1 : \text{enemyY} = \text{oldY}
\text{eSize} = \text{eSize} + \text{eSizeInc}
\quad \text{'change enemy's size}

RETURN

EnemyShot: enemy fires a shot!

\text{enemyShots} = \text{enemyShots} + 1
\text{FOR } n = 1 \text{ TO } 3
\quad \text{SOUND} 10000,1 : \text{SOUND} 9000,1 : \text{SOUND} 8000,1
\text{NEXT}
\text{IF} \text{RND}(1) < \text{qEHitProb} \text{ THEN GOSUB PlayerKilled}
\quad \text{'probability of hit}

RETURN

PlayerKilled: player has been hit and killed

\text{FOR } n = 1 \text{ TO } 4
\quad \text{FOR } n = 2000 \text{ TO } 1000 \text{ STEP } -100 : \text{SOUND} n,.5 : \text{NEXT}
\text{CALL INVERTRECT(VARPTR(viewRect(1)))}
\text{FOR } t = 1 \text{ TO } 100 : \text{NEXT}
\text{CALL INVERTRECT(VARPTR(viewRect(1)))}
\text{FOR } t = 1 \text{ TO } 100 : \text{NEXT}
\text{NEXT}
\text{ships} = \text{ships} - 1 : \text{eSize} = 5
\text{CALL DrawExtra(ships - 1)}

RETURN

Warning: sound warning, enemy is getting close

\text{FOR } n = 1 \text{ TO } 4 : \text{SOUND} 200,1 : \text{SOUND} 100,1 : \text{NEXT}

RETURN
FireLasers:  
\[\text{mouse events come here: fire laser}\]
\[
\text{test} = \text{MOUSE}(0)\]
\[
\text{do this to set mouse x and y coords}\]
\[
\text{mouseX} = \text{MOUSE}(1) ; \text{mouseY} = \text{MOUSE}(2) ; \text{get mouse x and y coords}\]
\[
\text{IF mouseY} < \text{scrnTop} \text{ THEN mouseY} = \text{scrnTop}\]
\[
\text{SOUND 8000,2}\]
\[
\text{CALL PENMODE}(\text{penXor})\]
\[
\text{CALL MOVETO}(\text{gun1X},\text{gun1Y}) : \text{CALL LINETO}(\text{mouseX},\text{mouseY}) \quad \text{'draw two}\]
\[
\text{CALL MOVETO}(\text{gun2X},\text{gun2Y}) : \text{CALL LINETO}(\text{mouseX},\text{mouseY}) \quad \text{'erase two}\]
\[
\text{CALL MOVETO}(\text{gun1X},\text{gun1Y}) : \text{CALL LINETO}(\text{mouseX},\text{mouseY}) \quad \text{'lines}\]
\[
\text{CALL PENMODE}(\text{penCopy})\]
\[
\text{GOSUB CheckHit} \quad \text{'pointer on enemy?}\]
\[
\text{IF NOT hit THEN LMiss} \quad \text{'nope, better luck next time}\]
\[
\text{SOUND 50, 3}\]
\[
\text{eSize} = \text{eSize} - \text{eSizeDec} \quad \text{'yes, make}\]
\[
\text{IF eSize} < 0 \text{ THEN eSize} = 5 \quad \text{'enemy smaller}\]
\[
\text{GOSUB UpdateEnemy}\]
\[
\text{IF eSize} < \text{eHitSize} \text{ THEN GOSUB EnemyKilled} \quad \text{'hit if enemy small enough}\]
\[
\text{LMiss:} \quad \text{RETURN}\]

FirePhoton:  
\[\text{fire photon torpedoes!}\]
\[
\text{test} = \text{MOUSE}(0) \quad \text{get current pointer coordinates}\]
\[
\text{mouseX} = \text{MOUSE}(1) ; \text{mouseY} = \text{MOUSE}(2) \quad \text{'if no more photons then you can't fire them}\]
\[
\text{IF photonsLeft} = 0 \text{ THEN SOUND 5000,1 : SOUND 6000,1 : RETURN}\]
\[
\text{photonsLeft} = \text{photonsLeft} - 1 \quad \text{'calculate slope of}\]
\[
x1 = \text{gun1X} ; y1 = \text{gun1Y} ; x2 = \text{gun2X} ; y2 = \text{gun2Y}\]
\[
xlnc1 = \text{INT}((\text{mouseX} - \text{gun1X}) / \text{fSteps}) \quad \text{'laser path}\]
\[
ylnc1 = \text{INT}((\text{mouseY} - \text{gun1Y}) / \text{fSteps}) \quad \text{'laser path}\]
\[
xlnc2 = \text{INT}((\text{mouseX} - \text{gun2X}) / \text{fSteps}) \quad \text{'and follow path}\]
\[
ylnc2 = \text{INT}((\text{mouseY} - \text{gun2Y}) / \text{fSteps}) \quad \text{'and follow path}\]
\[
\text{FOR inc} = 1 \text{ TO } \text{fSteps}\]
\[
x1 = x1 + xlnc1 : x2 = x2 + xlnc2 \quad \text{'no matter what enemy size,}\]
\[
y1 = y1 + ylnc1 : y2 = y2 + ylnc2\]
\[
\text{SOUND } (\text{fSteps} - \text{inc}) ^ * 500 , .5 \quad \text{'if a hit then enemy is dead}\]
\[
\text{PUT } (x1,y1) - (x1 + 16, y1 + 16) \quad \text{GOSUB CheckHit} \quad \text{'return hit = true if it is}\]
\[
\text{PUT } (x2,y2) - (x2 + 16, y2 + 16) \quad \text{IF hit THEN GOSUB EnemyKilled}\]
\[
\text{GOSUB UpdatePhots}\]
\[
\text{RETURN}\]

CheckHit:  
\[\text{is current pointer position on top of enemy?}\]
\[
\text{hit} = \text{true} \quad \text{'return hit = true if it is}\]
\[
\text{IF mouseX<oldX OR mouseX>oldEndX OR mouseY<oldY OR mouseY>oldEndY THEN hit = false}\]
\[
\text{RETURN}\]
EnemyKilled:                      direct hit on enemy!
score = score + 1
GOSUB ExplodeEnemy
GOSUB UpdateScore
GOSUB NewEnemy
RETURN

ExplodeEnemy:                    show enemy exploding
t1 Rect(1) = oldY - 10 : t1 Rect(2) = oldX - 10
  'set rect to surround
  'enemy
CALL INVERTOVAL(VARPTR(t1 Rect(1)))
CALL PENMODE(penXor)
FOR n = 1 TO 4
  SOUND 1000,1
  t1 Rect(1) = t1 Rect(1) -10 : t1 Rect(2) = t1 Rect(2) - 10
  t1 Rect(3) = t1 Rect(3) +10 : t1 Rect(4) = t1 Rect(4) + 10
  CALL FRAMEOVAL(VARPTR(t1 Rect(1)))
  SOUND 1000,1
  CALL FRAMEOVAL(VARPTR(t1 Rect(1)))
NEXT
CALL PENNORMAL
  t1 Rect(1) = oldY - 10 : t1 Rect(2) = oldX - 10
  t1 Rect(3) = oldEndY + 10 : t1 Rect(4) = oldEndX + 10
  'to what it was originally
CALL INVERTOVAL(VARPTR(t1 Rect(1)))
PUT(oldX,oldY) - (oldEndX,oldEndY), ePict
RETURN

NewEnemy:                        screen just has stars; place enemy somewhere
enemyX = INT((350) * RND(1))     'generate random
enemyY = INT((250 - scrnTop) * RND(1) + scrnTop)'
x,y) coordinates,
eSize = 10                  'enemy size always starts out same
oldX = enemyX : oldY = enemyY
oldEndX = oldX + eSize : oldEndY = oldY + eSize * qEAspect
PUT (oldX,oldY) - (oldEndX,oldEndY), ePict
  'draw initial enemy
  eSizeInc = eSizeInc + 1
  eSizeDec = eSizeInc * 4
  eMoveInc = eMoveInc + 1
IF RND(1) < .5 THEN eXDir = -1 ELSE eXDir = 1  'set direction he is moving
IF RND(1) < .5 THEN eYDir = -1 ELSE eYDir = 1
RETURN

DrawView:                       draw player's viewing screen
CLS
PENNORMAL
FOR n = 1 TO ships - 1
  CALL DrawExtra(n - 1)
NEXT
CALL TEXTFONT(0)
LOCATE 1,30: PRINT "Enemies Shot:"
LOCATE 2,30: PRINT "Photon Torpedoes Left:"
GOSUB UpdateScore
GOSUB UpdatePhotons
LINE(0, scrnTop - 2) - (520, scrnTop - 1), bf
CALL FRAMERECT(VARPTR(viewRect(1)))
CALL INVERTRECT(VARPTR(viewRect(1)))
CALL PENMODE(penXor)
t1Rect(1) = gun1Y - 5 : t1Rect(2) = gun1X - 4
CALL PAINTOVAL(VARPTR(t1Rect(1)))
t1Rect(1) = gun2Y - 5 : t1Rect(2) = gun2X - 4
CALL PAINTOVAL(VARPTR(t1Rect(1)))
FOR n = 1 TO stars
starX = INT((500) * RND(1))
starY = INT((290 - scrnTop) * RND(1) + scrnTop + 1)
starR = INT(3 * RND(1) + 1)
t1Rect(1) = starY : t1Rect(3) = starY + starR
t1Rect(2) = starX : t1Rect(4) = starX + starR
PAINTOVAL(VARPTR(t1Rect(1)))
SOUND (starX + starY) * 15, .25
NEXT
CALL PENNORMAL
RETURN

UpdateScore: 'display updated score
LOCATE 1,42 : WRITE score
RETURN

UpdatePhotons: 'update number of photons left
LOCATE 2,50 : WRITE photonsLeft
RETURN

TitleScreen: 'draw title screen
CLS
oldX = 10 : oldY = 30 : oldEndX = oldX + 165 : oldEndY = oldY + 165 * qEAspect
PUT(oldX,oldY) - (oldEndX,oldEndY), ePict
PUT(250,200) - (415,275), pPict
TEXTFONT(0) : TEXTSIZE(36)
LOCATE 2,10 : PRINT "MacSpace!"
LOCATE 3,10 : TEXTSIZE(12)
PRINT "The final frontier...

CALL PENMODE(penXor)
CALL PENSIZE(3,3)
RESTORE GunPoints

'animate player
'firing at enemy
FOR n = 1 TO 4
READ ex, ey
SOUND 8000, 2
CALL MOVETO(253,235) : CALL LINETO(ex,ey)
CALL MOVETO(280,215) : CALL LINETO(ex,ey)
FOR t = 1 TO 1000 : NEXT
NEXT
CALL PENMODE(penCopy)
GOSUB ExplodeEnemy
RETURN

Instructions:

WINDOW 2, , (16,55) - (495,325), 2
PRINT
PRINT "Welcome to MacSpace"
PRINT "The object is to kill as many enemy Klaxxon ships as you can before they"
PRINT "destroy all of your ships. There are two ways to destroy the Klaxxons."
PRINT "The first is by shooting them with your lasers by putting the cross hairs"
PRINT "over the Klaxxon ship and clicking the mouse button. Every time you make"
PRINT "a hit, the Klaxxon ship will get smaller. Shrink the Klaxxon ship small"
PRINT "enough and it explodes. You have an unlimited number of laser shots."
PRINT "The second way to kill the Klaxxon is to aim with the cross hairs, then"
PRINT "fire a photon torpedo by pressing any key. No matter what the Klaxxon's"
PRINT "size, if you hit the ship with a torpedo it will explode. Be aware that every"
PRINT "time you destroy a Klaxxon ship, the next one is faster and trickier."
PRINT
CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,230) - (280,250)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1 : WINDOW CLOSE 2
RETURN

MakePhoton: build picture of photon torpedo
CLS
t1Rect(1) = 11 : t1Rect(2) = 11
t1Rect(3) = 18 : t1Rect(4) = 18
PAINTOVAL(VARPTR(t1Rect(1)))
LINE(14,8) - (14,20)
LINE(8,14) - (20,14)
GET(6,6) - (22,22), photonPict
RETURN

MakeShips: make two ship picture arrays
if MacSpace.graphics exists, then read it, else read from DATA statements
fileHere = true
ON ERROR GOTO FileError
OPEN"MacSpace.graphics" FOR INPUT AS #1
ON ERROR GOTO 0
IF NOT fileHere THEN ReadPictures
RETURN
INPUT #1, byteCount
image$ = INPUT$(byteCount,1)
PICTURE (85,90), image$
GET (85,90) - (140,125), ePict
CLS

noUse$ = INPUT$(1,1)
INPUT #1, byteCount
image$ = INPUT$(byteCount,1)
PICTURE (85,90), image$
GET (85,90) - (140,115), pPict
CLS
CLOSE 1
GOTO BuildPictEnd

ReadPictures:
'MacSpace.graphics isn't on disk: read from DATA
READ EPictData : CALL ReadArray(ePict())
READ PPictData : CALL ReadArray(pPict())

BuildPictEnd:
RETURN

getError:

IF ERR <> 53 THEN ON ERROR GOTO 0
'REturn fileHere = false if file not found
'REsult: unknown error: print and halt
RESUME NEXT

SUB DrawExtra(num) STATIC
'Sub: draw (or erase) extra ship
SHARED pPict()
x = (num MOD 5) * 40
y = INT(num / 5) * 15
PUT(x,y) - (x + 35, y + 20), pPict
END SUB

SUB ReadArray(array(1)) STATIC
'read given array from DATA statements
READ size
FOR n = LBOUND(array) TO LBOUND(array) + size -1
   READ num : array(n) = num
NEXT
END SUB

GunPoints: 'data for (x,y) endpoints for title screen's gun firing
DATA 200, 50, 60, 250, 60, 150, 70, 70

EPictData: 'data for the picture: EPict
DATA 150
DATA 56, 36
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 24
DATA 0, 0, 0, 24, 0, 0, 0, 24, 0, 0
DATA 0, 24, 0, 0, 0, 24, 0, 0, 0, 548
DATA 16384, 0, 0, 548, 16384, 0, 0, 1316, -24576, 0
DATA 0, 578, 16384, 0, 0, 32707, -512, 0, 1, -32768
DATA 384, 0, 6, 0, 96, 0, 8, 1023, -16368, 0
DATA 112, 1024, 8206, 0, 133, 18877, -27999, 0, 256, 4096
DATA 2048, -32768, 1023, -1, -1, -16384, 1024, 0, 0, 8192
DATA 2111, -1, -4, 4096, 1096, -30721, -7918, 8192, 2197, 20133
DATA 29353, 4096, 4360, -28672, 2320, -30720, 9215, -8192, 2047, -15360
DATA 4576, 0, 7, -30720, 2240, 0, 3, 4096, 1088, 0
DATA 2, 8192, 576, 0, 2, 16384, 448, 0, 3, -32768
DATA 32, 0, 4, 0, 16, 0, 8, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

PPictData: data for the picture: PPict
DATA 150
DATA 56, 26
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7
DATA -1, -4096, 0, 8, 0, 8192, 0, 7, -16381, -4096
DATA 0, 0, 16124, 8192, 8188, 0, 9344, 28672, 1030, 7169
DATA 18688, 0, 14337, 8710, -3080, 0, 4096, -25592, 9220, 0
DATA 4096, -30764, 18439, -16384, 8192, 32736, -28672, -32768, 8195, -32768
DATA 24583, -16384, 16328, 1092, 0, 8195, -32768, 2724, 0
DATA 8192, 32247, -15292, 0, 5440, -32248, 7, -16384, 4096, -32761
DATA -4096, -32768, 2049, 0, 3079, -16384, 15878, 0, 1016, 0
DATA 508, 0, 0, 4, 0, 0, 0, 56, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
III. DIVERSIONS

- MacBuild
- MacOrgan
- MacCompose
- MacFinance
- MacMeasure
MacBuild

Truth to tell, not all of us are potential Picassos. Creating drawings from scratch can be hard work, even if you have talent and easy-to-use tools like MacPaint. But have you ever wanted to just doodle with a few shapes and use them as "building blocks"—moving, arranging, and changing their size at will? If you have, and you have a 512K Fat Mac, MacBuild is for you.
MacBuild lets you play with prebuilt shapes and arrange them to form designs as simple or as complex as you wish. You can also create your own building blocks and add them to the set with the Customize option. You can use MacBuild as is for a few relaxing moments now and then, or you can expand its capabilities and create your own graphics "construction sets" after you've studied how the program works.

How to use MacBuild

When the program starts running, you'll see the screen shown in Figure 1. Note that the added MacBuild menu has two main options, Instructions and Customize. Choosing Instructions gives you the window shown in Figure 2. Customize, which we'll describe a little later, is used to create custom graphics shapes to use with the building blocks provided.

The MacBuild screen is divided into two areas, the "shapes" column and the "work" area. To select a shape, simply move the arrow pointer over the one you want to use, hold down the mouse button, and drag the shape into the work area of the screen. If you want to make the

FIGURE 1. The MacBuild screen and menu
shape larger, hit the period key or the > key (the shifted period). Each time you press the key, the period enlarges the shape by 1 pixel, and the > enlarges it in steps of 10 pixels. Similarly, to make the shape smaller, hit the comma or < key: The comma shrinks the object by 1 pixel each time, and the < shrinks it by 10 pixels.

When the shape has the size you want, and you’ve moved it to the location you want, you can “set” it there by releasing the mouse button. Once a shape is set, it cannot be picked up, moved, or manipulated further. You can, however, go back to the shape column and get another copy of the basic shape, since they’re never “used up.”

If you set an object and discover later that you don’t want it any more, you can get rid of it with the eraser (the little box with the E in it at the bottom of the shapes column). When you select the eraser and drag it into the building area, it becomes a small blinking circle. If you hold down the mouse button, everything you drag the eraser over disappears, just as with MacPaint’s eraser.

Figure 3 on the next page shows a sample design we created. We hope it gets your own “idea maker” going!
Creating a custom shape

The black box with the C in it, located just above the eraser, represents the “custom shape.” You can define this shape yourself by choosing Customize from the MacBuild menu. Figure 4 shows the editing window that appears on the screen when you choose this option. If you’ve ever used MacPaint, or have read about our MacCursor or MacAnimate programs, you’ll recognize the “FatBits” technique used here.

You start customizing your shape by using the Erase button to remove the C that first appears in the editing window. Then, to construct your new shape, simply move the mouse around in the FatBits area and click on the spots where you want black pixels to appear. As you work, you’ll see an actual-size image of the custom shape being built up in the shapes column where the boxed C had been. If you want to erase a black pixel, simply click on it with the pointer. To completely erase the custom shape, click the Erase button.
When you're satisfied with your custom shape, click the Continue button. Your main work area will then reappear, and any design you were working on will still be there. You can now use the custom shape in exactly the same way you used the predefined ones.

A few additional notes: The custom shape isn't saved at the end of a session, so you'll see the boxed C instead of the custom shape if you run the program again. If you want your custom shape again, you'll have to redefine it. Also, if you erase the custom shape and then use the eraser in the main work area, the C will also be redrawn.

**How MacBuild works**

Like most of our other utilities, this program has an endless main loop, but within this loop is another loop that handles the default activity: dragging shapes to the screen, enlarging or shrinking them, and positioning them. It makes sense to make the activity on which the user will be spending the
most time the default activity, so the program remains in this loop until it is interrupted by a menu event. This kind of structure is also used in our MacCursor utility.

The way the “custom shape” is defined, using the FatBits method in which the user clicks squares representing individual pixels to turn them on and off, is also very similar to the way a cursor is created in Mac­ Cursor. For this reason, we will cover the shape definition only briefly here and refer you to the MacCursor chapter for additional details.

An outline of MacBuild’s structure is shown in Figure 5.

---

**Initialization**
- allocate memory
- dimension arrays, define screen constants
- set up title window
- create and save shapes (DrawShapes)
- install MacBuild menu, enable menu event-trapping

**Main Program**
- WHILE true (endless loop, interrupted by menu events)
  - wait for button press
  - if button pressed in shapes area drag shape (DragShape)
  - if button pressed in eraser area, drag eraser (DragEraser)
- WEND

**Menu Events**
- on menu event, branch to appropriate subroutine
- Instructions: display instruction window
  - save screen area, open second window (OpenWindow)
  - create OK button
  - print instructions
  - wait for click on OK button to continue
- Customize: edit the custom image
  - save screen area, open second window (OpenWindow)
  - initialize edit window constants
  - create Erase and Continue buttons
  - show instruction text
  - draw work area, show current shape
  - WHILE not finished editing
    - if button pressed and in work area, reverse color of bit
  - WHILE button still down
    - if still in work area,
      - keep drawing in that color (AffectBit)
- WEND
  - if Continue button clicked, close window and return (CloseWindow)
  - if Erase button clicked, erase work area

**Figure 5. Program outline**
WEND
    close second window, restore screen (CloseWindow)

BasQuit: quit to BASIC

DeskQuit: quit to Mac Desktop

Subroutines

DragShape: hide the cursor
    WHILE button is down
        erase shape at old location
        get new location
        check for keypress to increase or decrease shape size
        draw shape at new location
    WEND
    if shape released in work area, OR it with screen contents
    show cursor

DragEraser:
    WHILE button is still down
        save area under eraser into temporary array
        draw expanding circle to erase small area
        if eraser isn’t in work area, restore area under eraser
    WEND

AffectBit:
    send output to Window 1
    change bit on main screen
    get changed image
    send output to editing window and put new image

OpenWindow:
    save area to be replaced by second window
    open window

CloseWindow:
    close window
    restore area under window

DrawShapes:
    create all shapes
    draw border at edge of screen

Figure 5. Program outline (continued)

Initialization

All variables are defined as integers: No other kind is needed, and using integers saves memory. Remember that saving memory, even when not crucial for fitting programs into 128K Slim Macs, helps the programs run faster. As the Microsoft BASIC manual points out, the amount of memory
not used by the program code, data, and stack, freed by an appropriate CLEAR statement, determines how much of BASIC can be kept in memory. More BASIC in memory translates to fewer disk accesses, and we all know that repeated Macintosh disk access is no fun!

Two arrays are defined for use with shapes. The *shapes* array holds the bit maps for the eight predefined shapes and the user-definable "custom" shape. The *eArray* array holds the smaller bit map for the eraser, which handles only a small area at a time. (The reason MacBuild only runs on Fat Macs is that these bit-map arrays consume too much of a 128K Slim Mac's memory.)

Most of the remaining statements in this section define screen areas. The first set of coordinates defines the size of the secondary window used as the dialog box for the instructions and the custom cursor definition box. The second set of coordinates (*gX1*, *gY1*, and so on) defines the area whose contents will have to be saved when the secondary window is used and restored when the window is closed. The variable *safety* is used to make the area saved overlap the secondary window a little to provide a safety margin.

In the final line of [Init 1], the size of the array needed to save the screen contents is determined by a formula given in the Microsoft manual (and described in detail in our chapter on MacGurkha).

Finally, the *InBits* function is defined so the program can tell if the mouse pointer has been clicked in the FatBits area of the editing window.

### Setting up the main screen

The MacBuild title window is set up in [Init 2]. This window contains the shapes column to the left, where the available shapes (including the custom shape and the eraser) are displayed, and the work area, where shapes are assembled to make pictures.

The predefined shapes are drawn via a call to *DrawShapes* [SR 6]. They are all simple shapes constructed mostly from line statements, line statements with the box option, and a circle statement. The custom shape is drawn as a filled box with the letter *C* in it, and the eraser is drawn as an open box containing the letter *E*. The last line is a double-thickness vertical barrier that separates the shapes column from the work area.

After *DrawShapes* returns, the shapes are saved into the *shapes* array, using a FOR...NEXT loop. First, the *y* coordinates of the borders between the shapes are read from the data statements at [Dat]. All the shapes are located in rectangles 20 pixels high except the horizontal line (shape 5),
whose rectangle is only 10 pixels high. Each rectangle starts 5 pixels below the preceding boundary. The GET statement gets each shape into the corresponding element of the shapes array. The custom shape also has its boundaries stored in variables at this time; these variables will be used later in working with it. This is done here, rather than back in the constant initialization section, so that the coordinates can be tied into the data statements and thus changed automatically if the latter change (perhaps due to addition of more shapes).

The MacBuild menu is created in [Init 3], which also enables menu event-trapping. Menu events will be handled by our usual Menu-Event subroutine [ME 0].

Main program loop

The main program loop [Main], as previously noted, is an endless loop. At this point, the shapes and work area have been set up, so the user may be taking one of two actions: choosing a menu option or manipulating shapes. If a menu item is chosen, control goes to MenuEvent and the appropriate subroutine is called from there. We'll cover the menu events later.

The user will start manipulating a shape by moving the mouse cursor over it and holding down the mouse button. Therefore, the first thing in the main program loop is a WHILE...WEND loop that simply waits for the mouse button to be held down. When MOUSE(0) returns a value of −1 or less, the button will be down, which means that the user wants to drag a shape into the work area.

Next, MOUSE(1) returns the mouse pointer's x coordinate, and its location is checked to see if it is to the right of the boundary line separating the shapes and the work area. If it is, the pointer's in the work area and there's no shape to select; control goes to the label NotInShapes, and we go back to the top of the main loop and enter the “wait” loop again.

If the mouse pointer is in the shapes column when the button is pressed, we need to find out which shape it is over so that the shape can be dragged along with the pointer. After a RESTORE resets the data pointer [Dat], the shape boundaries are read by a WHILE...WEND loop until the boundary just read has a y coordinate greater than or equal to the y coordinate of the mouse pointer's. When this happens, we exit the loop: The variable shape is between 0 and 11.

“But there are only ten shapes,” you might say. True—but if we stop with the first boundary (shape = 0), which is the top boundary of the first shape (the circle), the pointer is above this shape, since we stop when a
boundary has a higher y coordinate than the pointer. The second boundary \((shape = 1)\) would be just below the circle, so if we stop there, the pointer is over the circle, and so on through the rest of the shapes. \(shape = 10\) would be the boundary just below the eraser, so the mouse would be over the eraser. \(shape = 11\) would be the bottom of the shapes column below the eraser.

All this means is that if we leave the loop with a shape number between 1 and 9 inclusive, a shape has been selected and DragShape [SR 1] is called. If the shape number is 10, the eraser was selected and DragEraser [SR 2] is called; it is separate because it requires extra processing to handle the erasing function in addition to the dragging function. If the shape number is 0 or 11, the mouse pointer is not over a shape even though it is within the shapes column, so control goes by default to NotInShapes (in other words, this situation is treated just as though the mouse were completely outside the shapes area).

**Dragging, sizing, and placing a shape**

If a shape other than the eraser is selected, DragShape [SR 1] is called. At the beginning of this subroutine, a call to the QuickDraw HIDECURSOR routine makes the cursor invisible so that the shape is the only thing that moves in response to the mouse drag—in effect, the shape becomes the cursor. The starting mouse-pointer coordinates are stored in the variables \(oldX\) and \(oldY\). Because of the way GET and PUT work with bit-map arrays, the width and the height of the shape (plus 1) are found in the first and second elements of that part of the \(shapes\) array corresponding to the selected shape. (A note on this: The Microsoft manual says these would be in elements 0 and 1, but in this program we have set OPTION BASE to 1, so all arrays start with element number 1.) These dimensions are assigned to the variables \(w\) and \(h\) respectively. A PUT statement draws the shape where the last mouse-pointer location was determined to be.

A WHILE...WEND loop is then used to continue the dragging process as long as the mouse button is down. The loop first checks to see if a key was pressed by getting the current value of INKEY$. If a valid key was pressed, the user wants to either "grow" or "shrink" the currently selected shape. Next, the mouse-pointer location is checked. If it has not changed and no key was pressed, then the mouse hasn't been moved and there is no request for changing the shape's size. In this case, control goes on to the label NoMove and from there back again to the top of the WHILE...WEND loop. At that point things will be checked again to see if a change has occurred.
If the mouse has moved and/or a key was pressed, the shape at the old location is erased by being PUT again at that location. (Remember that PUT defaults to an XOR, so a second PUT at the same location erases the object.) The variables oldX and oldY are then updated by being assigned the new mouse coordinates. Next, any key that was read is checked. If the period key was pressed, the shape is made one pixel higher and wider; if the key was >, an increase of 10 pixels in both dimensions is made. The same method is used for the comma and < keys to decrease the shape's size. In this case, however, ABS (the absolute value) must be used so that the values of height and width don't go below 0, causing subsequent PUT statements to have a rectangle whose second coordinates are to the upper left of the first ones. The shape is then drawn in its new size and position.

This loop continues as long as the mouse button is down. When the button is released, the loop is exited, and, if the mouse pointer is still inside the work area, the shape is drawn in the final pointer position. The PUT statement uses OR so shapes can be combined with existing shapes in the work area without erasing them. You can use other modes (like XOR) if you want to create other effects. The keyboard buffer is also cleared out by a WHILE...WEND loop. This is important because many users like to hold a key down to “zoom” shapes in or out, which tends to leave extra keystrokes in the buffer. Finally the cursor is restored to normal, and the subroutine returns.

**Dragging the eraser**

The eraser is managed by DragEraser [SR 2]. This subroutine handles dragging in a way similar to that used for the shapes, but here we have to deal with the additional function of erasing the area under the eraser.

Again, the cursor is hidden at the beginning of the subroutine, and a WHILE...WEND loop handles dragging the eraser until the mouse button is released. A GET statement temporarily stores the contents of the screen under the eraser in eArray. This is done so that if the eraser goes out of the work area into the shapes area, the contents of the screen under it can be restored rather than erased.

Next, a FOR...NEXT loop is used to create a small circle that grows in radius, extending from one to three pixels out from the mouse-pointer coordinates. The alternating of the default black with white makes this circle appear to blink. The final circle, a white circle with a radius of three pixels, in effect erases the area underneath it. If the mouse-pointer location is in the shapes area rather than in the work area, the erased
contents are restored by PUTtting them from eArray with the OR option. Otherwise, they stay erased.

When the mouse button is up again, the normal cursor is restored and control returns to the main program loop.

Creating the custom shape

Now that we've covered the default activities of shape manipulation, we'll discuss the main menu option, Customize. The custom shape can be used as is (a black block containing the letter C), but that isn't very interesting. Normally, the user will want to create his or her own shape and will select Customize from the MacBuild menu to do so. When that happens, Menu-Event [ME 0] calls Customize [ME 2a].

Customize first calls OpenWindow [SR 4] to save the current contents of the main work area and then open a second window, which will be used to contain the editing area for the custom shape. In most of our other programs, there is no need to preserve the contents of a window underneath a new window so that the original window can be restored or "refreshed" later. Here, however, we want the user to be able to leave a set of shapes in the main work area, choose Customize, create a new shape, and then go back to building a picture. Therefore, the temp array is dimensioned to gSize (which was calculated using the formula mentioned back in [Init 1]), and the contents of the "save" area (whose coordinates were also specified in [Init 1]) are saved into this array. This is a large amount of memory; gSize for this size of dialog box is 6465 elements, or about 13K total. The editing window (or dialog box) is then opened.

After returning from OpenWindow, Customize sets up the coordinates for the area within the secondary window that will be used for the custom shape work area. The dimensions are eight times those of the shape's actual size. The function InBits (used in several other programs) will determine whether the mouse pointer is in this work area. Two buttons, Erase and Continue, are then created, and the simple customization instructions are displayed.

Next, the actual work area is drawn with a LINE statement containing the box option, and the current custom shape (which is either the default "C box" or the last custom image defined in the current session) is drawn in the work area. A WHILE...WEND loop is started; the flag
finished is used to exit this loop when the user has completed the custom shape definition and chooses to continue with normal work.

The mechanisms used in Customize's main loop for finding the "bit," or pixel, of the custom image corresponding to the mouse-pointer location, changing the color of the pixel under the mouse cursor, and updating the image are very similar to those used in MacAnimate. Note that in this program the "real size" version of the image is maintained and updated in the custom shape's place in the shapes area. Also note that the Erase button is handled simply by drawing a white box over the work area and then using AffectBit [SR 3] to erase the image.

AffectBit first changes the Output window to the main window in preparation for drawing the real-size custom image in the shapes area. Note that changing the Output window doesn't change the active window, which remains the secondary window where the customization work area is. The PSET statement changes the affected bit in the real-size image. (Remember that the "cus-" coordinates are those of the custom image in the shapes area, while the "b" coordinates are those of the customization work area.) The new custom image is then saved into the shapes array via a GET statement. The second window is again made the Output window, and the image in the customize work area is redrawn at the larger size.

When the user clicks the Continue button, the finished flag is set to true and the main loop in Customize is exited. CloseWindow [SR 5] is then used both to close the second window, and to restore the contents of the main work area that were covered. These contents are PUT from the temp array that was dimensioned in OpenWindow. The ERASE statement then completely eliminates the temp array from memory. This means that each time the sequence OpenWindow/do some processing/CloseWindow is performed, the temp array is redimensioned, used, and then removed. Thus the array occupies memory only while it is in use. This redimensioning technique, which is not available in most other BASICS, can help save substantial amounts of memory.

The other main menu event, Instructions, is handled by [ME 1]. It also uses OpenWindow to open a second window and save the contents of the main window (if any) and CloseWindow to close the second window and restore the main window's contents. The last two menu events, Quit to BASIC [ME 3] and Quit to Desktop [ME 4], are handled as usual.
Suggestions for MacBuild

One feature you might want to add is the ability to save custom shapes to disk and restore them as desired. You might also want to write a routine that will save a completed design to the Clipboard, so you can transfer it to MacPaint or another application. You could also add a whole second row of shapes or allow several custom shapes to be in use at the same time.

MacBuild could also be the basis for many "construction set" type programs, including educational programs for children.

A Load option could also be added to the MacBuild menu to take sets of images from MacInterface or MacAnimate and use them as the primary shapes. And because saving images as picture strings often takes much less memory than storing them as bit-map arrays, you could try re-writing the program to use picture strings for the shapes.
MacBuild

A construction graphics program

```
DEFINT A-Z
OPTION BASE 1
DIM shapes(50,9)
DIM eArray(20)
false = 0: true = NOT false
white = 30: black = 33
lSide = 36

dX1 = 100: dY1 = 60: dX2 = 430: dY2 = 280
xOff = 1: yOff = 40: safety = 10

'holds shape bit maps

'colors used for screen drawing
left side of drawing space

'instruction box coordinates
'and offsets

'coordinates used in saving
under inst. box

'calculate array size necessary to save screen under instruction box

'InBits returns true if given (x,y) is in FatBits area
DEF FN InBits(fX, fY) = (fX >= bX1 AND fX <= bX2 AND fY >= bY1 AND fY <= bY2)

WINDOW 1, "MacBuild", (2,40) - (508,337), 1
GOSUB DrawShapes
RESTORE
FOR shape = 1 TO 9
READ yCut
IF shape = 5 THEN height = 10 ELSE height = 20
GET(10, yCut + 5) - (30, yCut + 5 + height), shapes(1,shape)
IF shape <> 9 THEN NotCustom
cusTop = yCut + 5: cusLeft = 10
cusBot = cusTop + height : cusRight = 30
NotCustom:
NEXT shape

'Menu commands
MENU 6, 0, 1, "MacBuild"
MENU 6, 1, 1, "Instructions"
MENU 6, 2, 1, "Customize"
MENU 6, 3, 0, "."
MENU 6, 4, 1, "Quit to BASIC"
MENU 6, 5, 1, "Quit to DeskTop"
ON MENU GOSUB MenuEvent : MENU ON
```
**MAIN**

**MAIN PROGRAM**

```
WHILE true
    WHILE MOUSE(0) > -1 : WEND
    IF MOUSE(1) > 30 THEN NotInShapes
    RESTORE
    shape = -1 : yCut = -1
    WHILE MOUSE(2) > yCut
        READ yCut: shape = shape + 1
    WEND
    IF shape >= 1 AND shape <= 9 THEN GOSUB DragShape
    IF shape = 10 THEN GOSUB DragEraser
    NotInShapes:
    WEND
```

**ME 0**

**MENU EVENTS**

```
MenuEvent:
    MENU 6, 0, 0
    item = MENU(1)
    ON item GOSUB Instructions, Customize, , BasQuit, DeskQuit
    MENU 6, 0, 1
    RETURN
```

**ME 1**

**Instructions:**

```
GOSUB OpenWindow
BUTTON 1, 1, "OK", (250,20) - (290,40)
CALL TEXTFACE(1)
PRINT "INSTRUCTIONS"
CALL TEXTFACE(0)
PRINT "To use a shape:
  1. Place cursor on desired shape"
PRINT "  2. Hold down mouse button"
PRINT "  3. Drag shape to desired location"
PRINT "  4. Use '.' or '>' to increase the size"
PRINT "  5. Use ',' or '<' to decrease the size"
PRINT "  6. Release mouse button"
PRINT "Choose 'Customize' to change the custom shape."
PRINT "To erase, drag the eraser into the workspace."
WHILE DIALOG(O) <> 1 : WEND
GOSUB CloseWindow
RETURN
```

**ME 2a**

**Customize:**

```
GOSUB OpenWindow
bX1 = 10 : bY1 = 10
bX2 = bX1 + 8 * shapes(1,9) : bY2 = bY1 + 8 * shapes(2,9)
BUTTON 1, 1, "Erase", (200, bY2) - (310, bY2 + 16)
BUTTON 2, 1, "Continue", (200, bY2 + 20) - (310, bY2 + 36)
CALL MOVETO(200, bY1 + 10) : PRINT "Click the mouse on"
CALL MOVETO(200, bY1 + 26) : PRINT "a bit to change it."
```
LINE (bX1 - 1, bY1 - 1) - (bX2 + 1, bY2 + 1), b
PUT (bX1, bY1) - (bX2, bY2), shapes(1,9)

finished = false
WHILE NOT finished
    status = MOUSE(0) : x = MOUSE(1) : y = MOUSE(2)
    IF status <> -1 OR NOT FN InBits(x,y) THEN NotDown
        'reverse color of current bit
        IF POINT(x,y) = black THEN color = white ELSE color = black
        WHILE status < 0
            cX = INT((x - bX1) / 8)
            cY = INT((y - bY1)/8)
            'while button still held down
            IF FN InBits(x,y) THEN GOSUB AffectBit
            status = MOUSE(0) : x = MOUSE(1) : y = MOUSE(2)
        WEND
    WEND

NotDown:
    IF DIALOG(0) <> 1 THEN NoButton
    btn = DIALOG(1)
    IF btn = 2 THEN finished = true
    IF btn <> 1 THEN NoErase
    WINDOW OUTPUT 1
    LINE (cusLeft, cusTop) - (cusRight, cusBot), white, bf
    cX = 0 : cY = 0 : color = white
    GOSUB AffectBit
NoErase:
NoButton:
    WEND
    GOSUB CloseWindow
    RETURN

BasQuit:  '---------------------------------------------------------------------------------------------------- quit to BASIC
    MENU RESET
    END

DeskQuit: '--------------------------------------------------------------------------------------------------- Return to Mac Desktop
    MENU RESET
    SYSTEM

****************************************************************************************************** SUBROUTINES

DragShape: 'drag shape until user releases button
    CALL HIDE_CURSOR
    oldX = MOUSE(1) : oldY = MOUSE(2)
    w = shapes(1, shape) - 1
    h = shapes(2, shape) - 1
    PUT (oldX, oldY), shapes(1, shape)
    WHILE MOUSE(0) <> 0
        key$ = INKEY$
        IF MOUSE(1) = oldX AND MOUSE(2) = oldY AND key$ = "" THEN NoMove
        PUT (oldX, oldY) - (oldX + w, oldY + h), shapes(1, shape)
            'erase old shape
        oldX = MOUSE(1) : oldY = MOUSE(2)
        IF key$ = "" THEN NoKey
        IF key$ = "." THEN h = h + 1 : w = w + 1
IF key$ = ">" THEN h = h + 10 : w = w + 10
IF key$ = "," THEN h = ABS(h - 1) : w = ABS(w - 1)
IF key$ = "<" THEN h = ABS(h - 10) : w = ABS(w - 10)

NoKey:
   PUT (oldX, oldY) - (oldX + w, oldY + h), shapes(1, shape)  'draw new shape
NoMove:
   WEND

   PUT (oldX, oldY) - (oldX + w, oldY + h), shapes(1, shape)  'erase last XORed shape
   IF oldX > ISide THEN PUT (oldX, oldY) - (oldX + w, oldY + h), shapes(1, shape), OR
   WHILE INKEY$ <> "" : WEND  'clear keyboard buffer
   CALL SHOWCURSOR
   RETURN

DragEraser:
   ' _______________________________________________ drag eraser around to erase drawing
   CALL HIDECURSOR
   WHILE MOUSE(0) <> 0  'while button down
      mX = MOUSE(1) : mY = MOUSE(2)
      IF mX > 4 THEN GET(mX - 4, mY - 4) - (mX + 4, mY + 4), eArray
      FOR n = 1 TO 3
         CIRCLE (mX, mY), n
      NEXT n
      IF mX < ISide + 4 THEN PUT(mX - 4, mY - 4) - (mX + 4, mY + 4), eArray, OR
   WEND
   CALL SHOWCURSOR
   'GOSUB DrawShapes
   RETURN

AffectBit:
   ' ___________________________ affect bit for custom cursor at (cX,cY)
   WINDOW OUTPUT 1  'first, set new shape image
   PSET (cusLeft + eX, cusTop + cY), color
   GET (cusLeft, cusTop) - (cusRight, cusBot), shapes(1,9)  'save it
   WINDOW OUTPUT 2  'then redraw bit image
   PUT (bX1, bY1) - (bX2, bY2), shapes(1,9), PSET
   RETURN

OpenWindow:
   ' ___________________________________________ open second window
   DIM temp(gSize)  'allocate memory to save screen
   GET (gX1, gY1) - (gX2, gY2), temp  'use GET to save bit image
   WINDOW 2, , (dX1, dY1) - (dX2, dY2), 2  'open dialog box
   RETURN

CloseWindow:
   ' ___________________________________________ close second window
   WINDOW CLOSE 2  'close dialog box
   PUT (gX1, gY1) - (gX2, gY2), temp, PSET  'redraw screen
   ERASE temp  'dispose of memory
   RETURN
DrawShapes: draw shapes on screen
CIRCLE (20,20), 10 ‘draw circle
LINE (10,40) - (30,60), , b ‘draw box
LINE (20,70) - (20,90) ‘draw cross
LINE (10,80) - (30,80)
LINE (20,100) - (10,120) ‘draw triangle
LINE (20,100) - (30,120)
LINE (10,120) - (30,120)
LINE (10,130) - (30,130) ‘draw horiz. line
LINE (20,140) - (20,160) ‘draw vert line
LINE (10,190) - (30,170)
LINE (10,100) - (30,120) ‘draw slash
LINE (10,200) - (30,220) ‘draw backslash
LINE (10,230) - (30,250), , bf ‘draw custom default shape
CALL TEXTMODE(2) : CALL MOVETO(16,245) : PRINT "C"
CALL TEXTMODE(0) : CALL MOVETO(16,275) : PRINT "E"
LINE (10,260) - (30,280), , b ‘draw barrier
RETURN

y values for boundary limits between shapes
circ, sqr, cross, tri, hor line, vert line
DATA 5, 35, 65, 95, 120, 135
slash, backslash, custom, erase
DATA 165, 195, 225, 250, 280, 400

circle, square, cross, triangle, horizontal line, vertical line
slash, backslash, custom, erase
MacOrgan

In the last chapter we introduced our Mac music package, MacCompose. We hope that you have been inspired by MacCompose to play excerpts from sheet music, experiment a bit with music theory, or even compose some original melodies of your own.

Having provided a tool for composition, we move to the other half of the musical art: performance.
MacOrgan turns your Macintosh keyboard into a miniature organ with almost two chromatic octaves of single-note keys and three chord keys. We think you'll have fun using it to play simple melodies, including perhaps those you have written using MacCompose. After a few practice sessions with MacOrgan, you just might feel inspired to dress up in white tie and tails, set an elegant crystal vase with a single long-stemmed rose on your Mac, and hold a recital to display your keyboard virtuosity!

How to use MacOrgan

When you run MacOrgan, you'll see the screen shown in Figure 1. Everything you need is on this one screen. As soon as you see the screen, you're ready to play without having to make any menu or button selections. The instructions provide all the information for using the program (the MacOrgan menu contains simply the two standard quit options), but we'll elaborate on them a bit in the following section and give you a few playing tips.

Note that there are two kinds of keys on the MacOrgan keyboard. Most of the top two rows of the Macintosh keyboard are converted into musical keys.

![Image of MacOrgan screen]

This represents the top rows of keys on your Macintosh keyboard, converted into a musical keyboard. The letters on the keys represent the notes of the scale. For instance, the note 'A' is at the location of the 'Q' key on the keyboard, so if you press the 'Q' key the 'A' note will sound. Try it! Just tap the 'Q' key. Notice that the note plays longer when you hold the key down. In the same way, the '2' key sounds the note 'A#' and so on. The 'Fc' stands for the F chord and is played by the 'F' key on the Macintosh. Same for the 'Gc' and 'Cc'. It takes a while to get used to this musical keyboard but with a little practice you'll be playing with the confidence of Bach (or at least the Phantom of the Opera!)
that play a single note (only the lowercase keys sound notes, so make sure the Caps Lock key isn’t locked when you play). You can think of this as a piano keyboard with the keys in the QWERTY row being the white keys and the keys in the number row being the black keys. If you know anything about the piano and organ, you know that the white keys play the whole notes and the black keys the corresponding half notes.

If you type across only the QWERTY row from C to C, you’ll hear a diatonic musical scale. If you let your finger linger a little longer on some of the keys, you’ll notice that their notes last longer the longer your finger presses the key. Finally, if you type across the QWERTY row from C to C and include the half notes in the row above, the result will be a complete, or chromatic, scale.

Now look at the keys marked Fe, Gc, and Cc in the figure, which are the F, G, and C keys on the Mac keyboard. The little cs indicate that these keys play chords (the F chord, G chord, and C chord, as you’ve probably deduced). These chords are groups of notes (three each in this case) that make a harmonious tone when sounded together. When you briefly press one of these keys, you hear a much richer tone because these keys play three notes at once and use the Macintosh’s capability to play several voices simultaneously. You can think of a voice as being a separate sound channel that can have its own characteristic tonal color (musicians call this timbre). Like the single-note keys, the chord keys continue to sound their tones as long as you hold them down.

**Some playing tips**

If the music coming out of your Mac is too soft or too loud, you can choose Control Panel from the Apple menu and adjust the sound level by sliding the volume control up and down with the mouse.

Do you want to play from sheet music? We suggest using a simple melody (the melody line in simple “melody-plus-chords” guitar music works best; piano music tends to have too many notes and is confusing for the beginner). Try to pick pieces that fit within two octaves (that is, the lowest and highest notes are no more than 16 full notes apart).

Playing tunes with the correct rhythm take a bit of getting used to, since you have to judge how long to hold down the key to get a note of a given duration. The chord keys in particular tend to run away and play for a long time if held too long.
How MacOrgan works

Once the required initialization has been completed, the program handles only one ongoing activity: playing notes and chords when the appropriate keys are struck. There are only two menu events, which are the standard quit loop options used in our other programs. The main program is an endless loop that simply processes keys as they are struck. The structure of the program is outlined in Figure 2.

<table>
<thead>
<tr>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimension arrays for pitch and organ waveform</td>
</tr>
<tr>
<td>initialize pitch array and note key string</td>
</tr>
<tr>
<td>initialize organ waveform</td>
</tr>
<tr>
<td>set voices</td>
</tr>
<tr>
<td>draw keyboard on the screen (DrawKeyBoard)</td>
</tr>
<tr>
<td>display instructions (Instructions)</td>
</tr>
<tr>
<td>install MacOrgan menu, enable menu event trapping</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE true (endless loop interrupted by menu events)</td>
</tr>
<tr>
<td>get character from keyboard</td>
</tr>
<tr>
<td>if a &quot;c&quot;, play C chord (Cchord)</td>
</tr>
<tr>
<td>if an &quot;f&quot;, play F chord (Fchord)</td>
</tr>
<tr>
<td>if a &quot;g&quot;, play G chord (Gchord)</td>
</tr>
<tr>
<td>if a valid key, play corresponding note in organ voice</td>
</tr>
<tr>
<td>WEND</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Menu Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuEvent: branch to appropriate subroutine</td>
</tr>
<tr>
<td>BasQuit: quit to BASIC</td>
</tr>
<tr>
<td>DeskQuit: quit to Mac Desktop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cchord: play C chord</td>
</tr>
<tr>
<td>Fchord: play F chord</td>
</tr>
<tr>
<td>Gchord: play G chord</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DrawKeyBoard:</th>
</tr>
</thead>
<tbody>
<tr>
<td>display &quot;MacOrgan&quot; title</td>
</tr>
<tr>
<td>draw large backspace and tab keys (DrawBox)</td>
</tr>
<tr>
<td>for each key,</td>
</tr>
<tr>
<td>read its position and label</td>
</tr>
<tr>
<td>put label at specified position on screen</td>
</tr>
<tr>
<td>draw box around the key (DrawBox)</td>
</tr>
<tr>
<td>Instructions: display instructions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subprograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrawBox: assign coordinates to array and draw round-cornered box</td>
</tr>
</tbody>
</table>

FIGURE 2. Program outline
Setting up arrays, menu, and screen

Two arrays are used by MacOrgan, and they are dimensioned in [Init 1]. The first is pitch, which holds the actual pitch (in cycles per second) of the 21 single notes that are available on the keyboard. The second array is waveOrgan%, which holds the specifications for a waveform with an "organ" sound. Later we’ll see how this and the other waveforms are used in the four “voices” to create chords and notes.

The actual acoustic frequencies (in cycles per second) of the 21 notes are set next. The first note, A, is set to the standard 440 cycles. This and the rest of the notes represent an ascending sequence. You may know that a note in the next higher octave has a frequency of exactly twice the frequency of the lower note. For example, the second A on our keyboard has an acoustic frequency of 880. Another way of putting this is that each octave represents an added power of 2 (since increasing the power of 2 by 1 is the same as doubling). Because there are 12 notes a half step apart in an octave, the frequency of each note is 2 to the $\frac{1}{12}$ power times the preceding one. So in the formula used in the FOR...NEXT loop here each element of pitch is assigned the value of the preceding one times 2 to the $\frac{1}{12}$ power.

The last two lines in [Init 1] obtain the ASCII characters corresponding to the single-note keys (the chord keys are handled separately). The characters are contained in a DATA statement [Dat 2] that is simply READ and assigned to the string-variable key$. Besides the benefit of being more compact in both memory and listing space than using an array and a FOR...NEXT loop to read in individual characters, the storing of the characters in a single string allows the BASIC INSTR function to determine both if a character represents a valid note key, and to return the note’s position in the sequence without a loop being required.

If you compare the characters in [Dat 2] that are assigned to key$ with the keyboard, you’ll see that the characters are in ascending order of musical note (that is, chromatic-scale order), rather than being one keyboard row followed by the other. This is so the character sequence will match the note sequence. You may also notice that the first character, a question mark, isn’t actually one of the notes; it is a “dummy character” whose function will be explained when we look at how a key is converted to a note to be played.

In [Init 2] the characteristic tone color for single notes is defined. In connection with this, we’ll briefly review the parameters of the SOUND statement and the use of waveforms and voices. A more complete treatment of sound on the Macintosh can be found in Chapter 18 of Microsoft Macinations (Microsoft Press, 1985).
The SOUND statement has two required parameters: frequency (or pitch) and duration. There are two optional parameters: volume and voice. In MacOrgan, the pitch obtained from the pitch array element corresponds to the key pressed, and the duration is fixed (6 for chords, 3 for single notes). The actual duration seems to vary with the length of time the key is held down because the Macintosh keys auto-repeat while held down, and what is heard as a long, continuing note is actually a series of notes played in rapid succession.

The optional volume parameter is not used, so the volume is controlled by the default setting set by the user on the Macintosh Control Panel. The voice parameter is used, however. The Macintosh can play up to four independent “voices,” each of which can have its own tonal characteristics. MacOrgan uses the first voice (called voice 0 by BASIC) for the single notes, and the other three for the three notes used to form chords.

The single-note voice is defined by the values in the array waveOrgan%. Each element in this integer array has a value that indicates the relative “height” or amplitude of the wave at that point. The statements in the FOR...NEXT loop calculate the values of two sine waves: a dominant frequency and a frequency that is both one-third the amplitude and four times the dominant frequency. This results in a complex waveform somewhat similar to that produced by an organ. The shape of this waveform, the formula that produced it, and those of other common waveforms are shown in Figure 3.

After the organ waveform is defined, the WAVE statements are used to associate waveforms with the corresponding voices. The first voice (0) is assigned the waveform defined in waveOrgan%. The other three voices are assigned the predefined wave type SIN (or “sine wave”). This waveform has a curved shape like that shown in Figure 3, and has a full, “mellow” sound.

The MacOrgan window is opened in [Init 3]. As mentioned earlier, there is only one screen that will appear in this window. The elements of the screen are drawn next. First, DrawKeyboard [SR 4] is called to draw a representation of the part of the Macintosh keyboard used to play music. The title “MacOrgan” is displayed at the top in a large text size.

There are two keys that are wider than the rest: the Tab and Backspace keys. Neither of these keys is used to play a note, but all keys in the top two rows are represented, even those that don’t have notes associated
DIM waveSine%(255)  ; sine wave...flute-like
FOR n = 0 TO 255
    waveSine%(n) = 127 * SIN((n/256) * 6.2836)
NEXT n

DIM waveHiss%(255)  ; random noise...hissing, buzzing
FOR n = 0 TO 255
    waveHiss%(n) = RND * 256 - 128
NEXT n

DIM waveSqr%(255)   ; square wave...computer-like
FOR n = 0 TO 127
    waveSqr%(n) = -128
    waveSqr%(255 - n) = 127
NEXT n

DIM waveTri%(255)   ; triangle wave...oboe-like
FOR n = 0 TO 127
    waveTri%(n) = (n * 2) - 126
    waveTri%(255 - n) = (n * 2) - 126
NEXT n

DIM waveSaw%(255)   ; sawtooth wave...violin-like
FOR n = 0 TO 255
    waveSaw%(n) = n - 126
NEXT n

DIM waveOrgan%(255) ; addition of sine waves...organ-like
FOR n = 0 TO 255
    majorVoice = 96 * SIN((n/256) * 6.2836)
    minorVoice = 32 * SIN((n/64) * 6.2836)  ; 1/3 vol, 4x freq
    waveOrgan%(n) = majorVoice + minorVoice  ; 2 voices into 1
NEXT n

FIGURE 3. Some common waveforms and their formulas
with them; this allows the "musical" keys to be placed in the right context on the keyboard. These two wide keys are drawn by separate sequences using calls to the DrawBox subprogram [SP 1]. DrawBox takes four integers which are passed to it when it is called, representing the upper left and lower right coordinates of a rectangle with round corners. Since variables and numbers that are passed to a subprogram must be of an explicit type, we define \( x \) and \( y \) to be integers in [Init 1]. Operations done with these variables, such as defining screen locations, are therefore forced to return integer values. DrawBox takes these integer values and reassigns them to the \( \text{rect} \) array before using the FRAMEROUNDRECT QuickDraw routine to draw the keys as round-cornered rectangles.

Since the rest of the keys are the same size, a FOR...NEXT loop is used to draw them. For each key the \((x,y)\) screen location and name string are read from the data statements at [Dat 1]. A \text{MOVETO} places the QuickDraw pen at the specified coordinates, the name string in \text{key$} is printed, and the key is outlined with a call to the DrawBox subprogram using coordinates relative to the \((x,y)\) location.

After DrawKeyboard is finished and returns, Instructions [SR 5] is called to display the instructions in the bottom half of the screen. The only difference between the instructions in this program and those found in most others in this book is that the instructions here are not in a separate window selected as a menu option but appear and remain on the screen throughout the user's session with the program.

Next, the MacOrgan menu is installed and menu event-trapping enabled. The menu is even more limited than that of the average fast-food joint, as it has only two options: the standard Quit to BASIC and Quit to Desktop options.

The main program loop

The main program is an endless loop. It starts by getting the next keystroke to be processed from the keyboard buffer (INKEY$) and assigning it to \text{play$}. Three IF...THEN statements then check to see if the key is a chord key (C, F, or G). If it is, the appropriate chord subroutine (Cchord [SR 1], Fchord [SR 2], or Gchord [SR 3]) is called. These subroutines are identical except for the frequencies of the three notes that make up the chord to be SOUNDeed.

Next, we check for non-chord keys. The INSTR function determines if the character representing the key struck (\text{play$}) is in the string containing all the possible valid single-note keys (\text{keys$}). If the key pressed isn't one in the string, INSTR returns 1; otherwise, it returns the position
where the character representing the key pressed was found in keys$. You can now see the reason for the "dummy" first character we put in keys$: Since 1 is returned if a character is not found, we want 2 instead of 1 to be the position of the first valid character. Putting the dummy character at the start of the string bumps all the positions up by one and enables us to test for \( n > 1 \) to determine if a valid character was found, and then to use \( n - 1 \) to get the actual position of the note represented by the character in the musical scale.

If a valid character is found, the note is played using a SOUND statement with the pitch found in the \((n - 1)\) element of PITCH, since the true relative position of the note is 1 less, due to the dummy character. Since the characters in keys$ represent the 21 notes in ascending pitch order, they correspond to the pitches in the pitch array.

The playing of notes and chords continues until a Quit menu option is chosen. These standard menu options are self-explanatory, so we won't elaborate on them.

**Suggestions for MacOrgan**

The range of the organ could be extended about two octaves simply by including the bottom two rows of keys (except for keys designated as chord keys). No modification other than extending the appropriate loops and data statements would be needed.

You could add additional chords (perhaps as an alternative to the above). If you don't know what notes are used to construct particular chords, consult a beginning music text (many "how-to-play-the-guitar" variety of books also have this information). If you modify the loop in [Init 2] to print the pitches of the notes, you could construct a table of notes and frequencies to use in setting up the SOUND statements to play chords. Note that some fancier kinds of chords have more than three notes.

If you're feeling ambitious, you could combine the above two projects and use the uppercase of a key to refer to a chord and the lowercase to refer to a single note.

You can experiment with voices by changing the values put in the waveOrgan% array and listening to the resulting notes. You could have a two-dimensional array with different voices simulating different instruments (it would probably take some tinkering and/or reading a book on electronic sound theory) so you could set different voices as you play a selection, perhaps by using Option-key sequences.
MacOrgan program listing

MacOrgan
Play your Macintosh like an organ!

'** ** INITIALIZE

DIM pitch(21), waveOrgan%(256) 'for pitch and wave
DEFINT x-y
false = 0 : true = NOT false

pitch(0) = 440 'establish pitch for A
FOR n = 1 TO 21 'assign next 21 pitches to ascending notes
  pitch(n) = pitch(n - 1) * 2 ^ (1 / 12)
NEXT n

RESTORE KeyASCII 'get ASCII equivalents for notes
READ keys$ FOR pitch and wave

FOR t = 0 TO 255 'define multiple sine waveform for 'organ' sound
  majorVoice = 100 * SIN((t / 256) * 6.2838)
  minorVoice = 34 * SIN((t / 64) * 6.2838) '1/3 volume, 4x frequency
  waveOrgan%(t) = majorVoice + minorVoice
NEXT

WAVE 0, waveOrgan% 'establish waveforms for all 4 voices
WAVE 1, SIN
WAVE 2, SIN
WAVE 3, SIN

WINDOW 1, , (10,32) - (502,330), 2 'draw screen
GOSUB DrawKeyboard
GOSUB Instructions 'establish waveforms for all 4 voices

MENU 6, 0, 1, "MacOrgan" 'install MacOrgan menu
MENU 6, 1, 1, "Quit to BASIC"
MENU 6, 2, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON

'*** MAIN PROGRAM

WHILE true 'get code of key pressed
  play$ = INKEY$ 'if it's a chord, play it
  IF play$ = "c" THEN GOSUB Cchord
  IF play$ = "f" THEN GOSUB Fchord
  IF play$ = "g" THEN GOSUB Gchord
  n = INSTR(keys$, play$)
  IF n > 1 THEN SOUND pitch(n - 1), 3, , 0 'no vol. param with waveforms
WEND

INIT 1

INIT 2

INIT 3

MAIN
MAC ORGAN

**MENU EVENTS**

MenuEvent: 'on menu event, branch to appropriate subroutine

```
MENU 6, 0, 0
ON MENU(1) GOSUB BasQuit, DeskQuit
MENU 6, 0, 1
RETURN
```

BasQuit: 'quit to BASIC

```
MENU RESET
END
```

DeskQuit: 'quit to Mac desktop

```
MENU RESET
SYSTEM
```

**SUBROUTINES**

Cchord: 'pressing c on keyboard gives this chord

```
SOUND 329.6, 6,, 1
SOUND 392, 6,, 2
SOUND 523.25, 6,, 3
RETURN
```

Fchord: 'pressing f on keyboard gives this chord

```
SOUND 349.23, 6,, 1
SOUND 440, 6,, 2
SOUND 523.25, 6,, 3
RETURN
```

Gchord: 'pressing g on keyboard gives this chord

```
SOUND 293.66, 6,, 1
SOUND 392, 6,, 2
SOUND 493.88, 6,, 3
RETURN
```

DrawKeyboard: 'display image of keyboard

```
RESTORE KeyDefs
CALL TEXTFONT(0) : CALL TEXTSIZE(24)
LOCATE 1,12 : PRINT "MacOrgan!"  
\text{\textit{print large 'MacOrgan' on screen}}
CALL TEXTSIZE(12)
CALL MOVETO(55,75) : PRINT "Tab"
CALL DrawBox(50, 61, 89, 79)
FOR keys = 1 TO 29
READ x, y, key$
CALL MOVETO(x,y) : PRINT key$
\text{\textit{get key location and name}}
\text{\textit{go there and print key's note name}}
CALL DrawBox(x - 4, y - 14, x + 19, y + 4)
\text{\textit{draw box around key}}
NEXT
CALL DrawBox(376, 41, 414, 59)
\text{\textit{backspace}}
RETURN
```
Instructions:

CALL TEXTFONT(2) : CALL MOVETO(20,150)
PRINT "This represents the top rows of keys on your Macintosh keyboard,"
PRINT "converted into a musical keyboard. The letters on the keys represent the"
PRINT "notes of the scale. For instance, the note 'A' is at the location of the 'Q'"
PRINT "key on the keyboard, so if you press the 'Q' key the 'A' note will sound"
PRINT "Try it! Just tap the 'Q' key. Notice that the note plays longer when you"
PRINT "hold the key down. In the same way, the '2' key sounds the note 'A#' and "
PRINT "so on. The 'Fe' stands for the F chord and is played by the 'F' key on the "
PRINT "Macintosh. Same for the 'Gc' and 'Cc'. It takes a while to get used to this"
PRINT "musical keyboard but with a little practice you'll be playing with the"
PRINT "confidence of Bach (or at least the Phantom of the Opera)";
RETURN

SUB DrawBox(left%, top%, right%, bottom%)
rect%(0) = top% : rect%(1) = left% : rect%(2) = bottom% : rect%(3) = right%
CALL FRAMEROUNDRECT(VARPTR(rect%(0)), 12, 12)
END SUB

KeyDefs: define key location and label
DATA 55, 55, "", 80, 55, "", 105, 55, "A#", 130, 55, ""
DATA 155, 55, "C#", 180, 55, "D#", 205, 55, "", 230, 55, "F#"
DATA 255, 55, "G#", 280, 55, "A#", 305, 55, "", 330, 55, "C#"
DATA 355, 55, "D#", 95, 75, "A", 120, 75, "B", 145, 75, "C"
DATA 170, 75, "D", 195, 75, "E", 220, 75, "F", 245, 75, "G"
DATA 370, 75, "E", 395, 75, "F", 180, 95, "Fc", 205, 95, "Gc"
DATA 170,115, "Cc"

KeyASCII: string values for notes: given as A, A#, B, C, etc.
"?” is a dummy character, included so INSTR returns 1 if given null$
DATA "?q2we4r5ty7u8i9op-=[=]"
MacCompose

Nearly all of us are musical to some degree—even if we don’t think so. True, it’s probably been years since we paraded up and down the piano keyboard in lockstep with the persistent commands of the sainted Miss Crumwiddie. The $49 Sears Roebuck electric guitar outfit we bought in a burst of adolescent enthusiasm on the imagined road to stardom may rust silently in a
cobwebbed closet. But chances are that we still remember something of the fundamentals of music and retain the desire to make music ourselves now and then. This book brings you two programs that can help to revive or expand your musical expression: MacCompose and MacOrgan.

MacCompose is an aid to writing down music, whether your own compositions or tunes you've copied from sheet music. Once written, MacCompose will play the notes at the click of a button. On the other hand, MacOrgan, which we will cover in the next chapter, is for the performer in you: It turns your Mac into an electronic organ and lets you perform music—including music written with MacCompose, of course—at its keyboard. These programs promise neither Carnegie Hall glory nor rock 'n' roll riches, but they are useful little tools for exploring the elements of music and writing and playing simple melodies.

For example, if you just want to hear a favorite tune, you can enter the notes from sheet music and play them. Or, if you don't know much about the elements and structure of music, you can play with melodies, intervals, and rhythms by assembling them on the screen in standard musical notation and then hearing what they sound like. The ability to relate what you see to what you hear makes MacCompose a useful supplement to a music textbook. Or, if you're one of those amateur musicians who plays by ear or can sight-read only with difficulty, you can enter notes (such as a melody line) from sheet music, let MacCompose play the tune a few times for you, and then pick it up by ear yourself.

How to use MacCompose

Using MacCompose is extremely easy. When you first run the program, the title screen shown in Figure 1 briefly appears. Next, the working screen shown in Figure 2 appears, complete with the custom MacCompose menu containing Instructions, Quit to BASIC and Quit to Desktop. Choosing Instructions displays a window of brief instructions, which we'll elaborate on a bit here.

The center of the screen contains a standard staff with a treble clef. For those of you who don't know anything about musical notation, we'll mention a few fundamentals. The lines on the staff from bottom to top correspond to the notes E, G, B, D, F (Miss Crumwiddie may have told you to remember them as “Every Good Boy Does Fine”). The spaces between the lines, again starting at the bottom, correspond to the notes F, A, C, and E. MacCompose can handle all these plus four extra notes: a C and D below the staff and a G and A above it. That bottom C, by the way,
FIGURE 1. The MacCompose title screen

FIGURE 2. The MacCompose working screen
roughly corresponds to the middle C on the piano, although a given piano and the Mac may not be in tune with each other.

Along the bottom of the screen are the musical symbols you can put on the staff. The row begins with four notes: whole, half, quarter, and eighth. The remaining symbols, in order, are two rests (silent pauses)—quarter and eighth—and the sharp and flat signs (called accidentals). To put a symbol on the staff, place the arrow pointer over it, drag it to the staff, and release it at the vertical position you want it to have. You don’t have to worry about horizontal spacing: MacCompose automatically puts the new symbol in the next available space on the staff. If the symbol you place is a note, MacCompose will play it; if it’s a rest, you’ll hear a faint clicking sound. Rests and accidentals are moved and placed just like notes. A sharp raises the note that precedes it one half-step (or semitone) in frequency, and a flat lowers it one half-step.

You’ll note that there is no line on the staff low enough to accommodate the low C (middle C). To place this note, simply judge where the appropriate line would be if there were one and then release the note there: The note and line segment will appear as in Figure 2. Use the same kind of judgment to put the D just below the staff or the high G and A above it when necessary.

If you make a mistake, simply move the arrow pointer over the symbol you want to remove and click the mouse button: The symbol will disappear. If you want to clear everything off the staff and start over, click the New button in the lower left corner of the screen.

To play the tune you create, click the Play button; if you want to hear the tune several times, rapidly click the Play button several times. (If you haven’t already guessed, our tune in Figure 2 is the beginning of the familiar drinking song, “To Anacreon in Heaven,” but some of you might know it under its more recent title, “The Star-Spangled Banner.”)

**Copying sheet music**

Entering and playing sheet music with MacCompose can present some complications. We suggest that, at least at first, you pick sheet music with a simple melody line and a key signature that has no more than one sharp or flat. If the music you want to copy contains notes smaller than those MacCompose provides (such as sixteenth or thirty-second notes), you can multiply all the note values by 2 or 4 so they will fit in the sizes available and still preserve the correct rhythmic relationships. That is, you would change sixteenth and thirty-second notes to eighth notes, eighth notes to quarter notes, and so on.
If your sheet music has notes too high or low to fit in the range handled by the program, you might try transposing the melody to fit. This can be a bit tricky, but you may be able to correct it by ear. Finally, if you encounter dotted notes, try to choose individual notes with the correct values and substitute them. For example, a dotted half note can be replaced with a half note and a quarter note, since the dot adds half again to the value of the note.

We'll admit that MacCompose has some limitations. The staff can hold a maximum of 22 notes, and there are no bar lines, so it is hard to see phrasing. You can't enter chords, either. If you're working with sheet music for piano, we suggest that you pick out the melody line in the treble clef (after you've entered a few notes, you can play them and let your ear help you determine which ones belong in the melody). MacCompose works best with short and simple melodies.

**How MacCompose works**

This program centers on a main loop that controls the addition and deletion of notes to a melody. Button events control playing the current melody and initializing a new one. Since there are very few menu and button options, the main work of the program is in correctly placing and updating notes, as well as managing complications such as notes on ledger lines and accidentals. Our program outline in Figure 3 shows how these problems are handled. (Incidentally, we save space in the outline by using the term “note” to refer to regular notes, rests, and accidentals; the context should indicate what is being referred to in a given case.)

```
Initialization
  dimension arrays for note symbols, frequencies, durations, staff image,
  note type, note position
  initialize duration array
  set sine waveform
  draw and save note symbol images (FormPics)
  show title screen (TitleScreen)
  create Play and New buttons, enable button event-trapping
  install MacCompose menu, enable menu event-trapping

Main Program
  WHILE true (endless loop interrupted by menu and button events)
  set new tune flag to false
  set number of stored notes to zero
  redraw screen and prepare for new tune (InitTune)
```

*FIGURE 3. Program outline*
WHILE new tune not selected
    if mouse button anything but drag, ignore loop
    get current mouse pointer coordinates
    if pointer within staff area, check for note deletion (CheckDelete)  SR 4
    if pointer within note symbol area, drag note to staff (HandleNote)  SR 3
WEND
WEND

Menu Events

MenuEvent: branch to appropriate subroutine  ME 0

Instructions:
    open second window
    display instructions
    create button, wait for button click
    close second window

BasQuit: quit to BASIC  ME 2
DeskQuit: quit to Mac Desktop  ME 3

Button Events

ButtonID: branch to appropriate subroutine  BE 0

Play: play each note in sequence (PlayNote)  BE 1 (SR 6)
Start: set flag for restarting main loop  BE 2

Subroutines

InitTune:
    initialize note types and position information
    redraw screen (DrawScreen)  SR 1
    initialize frequency array  SR 2

DrawScreen:
    draw clef and staff (DrawClef, DrawStaff)  SR 2
    draw note symbols below staff  SR 11, 12
    place any melody present on staff

HandleNote:
    hide pointer
    move chosen symbol while still dragging
    if pointer in staff area, place symbol at first empty position
    if note placed just above or below staff, draw ledger line (Ledger)  SR 7
    add note to note sequence (AddNote)  SR 5
    show pointer

CheckDelete:
    get note position on staff from x coordinate
    if note at that position, erase shape, reset arrays
    if ledger line at that location, erase it (EraLedger)  SR 8
(continued)

Figure 3. Program outline (continued)
AddNote:  
store note region number for this note  
store note type at this position  
increment number of stored notes if note placed at end of tune  
if note symbol not sharp or flat, play note (PlayNote)  

PlayNote:  
if note symbol is note, play note, adjust if sharp or flat  
if note symbol is rest, pause for appropriate duration  

Ledger: draw ledger line  

EraLedger: erase ledger line  

TitleScreen:  
draw clef symbol (DrawClef)  
draw staff (DrawStaff)  
place each character of title string on staff at random locations  

FormPics: draw all note symbol shapes, save in bit-map array  

DrawClef: read data points, plot and connect points, save image  

DrawStaff: save staff segment equal to one note symbol width  

Figure 3. Program outline (continued)  

Initialization and constants

After making sure that there will be enough memory for its arrays, the program establishes the types of all variables except those beginning with f or z as integer. MacCompose uses the following six arrays for information and graphics:

- notePic holds the graphic images for the eight types of symbols;
- freq holds the frequencies of the thirteen notes that will fit in the staff area (including the four notes outside the staff proper);
- duration holds the duration (length of time for sounding) associated with the four sizes of notes handled (whole, half, quarter, and eighth) and the two sizes of rests (quarter and eighth);
- staffPiece holds the graphic image for a segment of the staff one note position wide. This will be used to redraw the background when a note is erased or changed;
- noteN holds a number corresponding to the type of note or symbol at each possible position on the staff;
- position holds a number corresponding to the scale value (and also the vertical region on the staff) of the note at each possible staff position.
The constant $z$ is based on the fact that musical-scale frequencies have a progression based on powers of 2. The constant will be used to raise or lower to the appropriate frequency when a sharp or flat is encountered.

The FOR...NEXT loop stores the values of the durations of the various notes and rests in the *duration* array. Since the durations shorten by powers of 2, the expression used with INT can set them in turn from 16 for a whole note to 2 for an eighth note. The durations given to the rests are simply assigned from those for the corresponding notes. An alternative, of course, would be to READ them all in from a set of DATA statements. The latter code would be easier to understand, but our code shows the mathematical relationship better.

**Drawing the title screen**

At [Init 2] FormPics [SR 10] is called so that the symbols for the various notes, rests, and accidentals can be drawn and stored in the *notePic* array. Of particular interest is the drawing of the filled bodies of the quarter and eighth notes by using a series of CIRCLE statements with the radius increased by 1 each time. (An alternative technique would be to use calls to the QuickDraw FRAMEOVAL and PAINTOVAL routines.) If you look closely at the filled notes in Figure 2 that aren’t on staff lines, you’ll see the tiny “eyes” on either side of their centers that are a consequence of this technique: The circles aren’t perfectly concentric. Also note that since the symbols for the accidentals are shorter than those for the notes and rests, you offsets are used to place them lower than the rest of the symbols.

After the symbols are drawn and stored, TitleScreen [SR 9] is called. TitleScreen in turn first calls DrawClef [SR 11] to draw the clef (the symbol that appears at the beginning of the staff). Because the clef is a complicated curve, the subroutine draws it essentially by a “connect the dots” technique. The points along the clef that indicate the gradations of curvature are specified in the DATA statements at ClefPoints [Dat 2], and the points are plotted with PSET statements and connected with LINE statements. This is a rather painstaking procedure, but sometimes there’s no practical alternative (although you could use MacPaint and MacInterface to create the image).

TitleScreen then calls DrawStaff [SR 12], which draws the staff, a refreshingly simple procedure. Next, TitleScreen sets up the somewhat
over-optimistic “Be a Mozart” title legend and specifies a large, bold text-font and style. A FOR...NEXT loop draws the title characters with a randomized vertical y location. The expression used with RND will evaluate randomly to −1, 0, or 1 (to see this, plug in some values of RND(1) between 0 and 1). Thus, the progression of character “notes” will vary but will not jump more than two spaces between characters, an arrangement that gives the general impression of a musical phrase. The IF statements and the following statement keep the value of \( j \) from positioning the character out of the staff area.

After a delay loop is finished, TitleScreen returns to [Init 3]. Next, the Play and New buttons are created and button event-handling is enabled. Then the MacCompose menu is installed and menu event-handling is enabled. (We will cover button and menu events and their handling after we consider the main processing loop.)

**The main program**

As usual, the main program is an “endless” loop that is exited only when one of the two quit menu options is executed. The loop starts by setting the NewTune flag to \emph{false}. It then calls InitTune [SR 1] to initialize the note information and staff contents.

InitTune starts by setting the number of the maximum note stored in memory to 1 and using a FOR...NEXT loop to set the note values and positions to \(-1\), indicating that there are no notes in memory or on the staff. It next calls DrawScreen [SR 2] to draw the work screen.

DrawScreen starts by drawing the clef and staff with DrawClef and DrawStaff, just as was done for the title screen. DrawScreen then draws the available notes and other symbols below the staff area by using a FOR...NEXT loop to PUT the images from the \emph{notePic} array.

The rest of DrawScreen is a FOR...NEXT loop that draws any stored melody notes on the staff (though there won’t be any stored melody the first time through this subroutine). Before we explain how this is done, let’s pause to introduce two terms that will be useful in describing the manipulation of notes and the other musical symbols in various parts of this program. A \emph{staff position} is a horizontal segment of the staff that is one note-image wide. It represents a “slot” into which a note can be put. A \emph{note region} is a vertical segment of the staff representing the spot where a note of a given pitch would be put. For example, the space between the bottom line of the staff proper and the next line up is the region for the note F above middle C.
The FOR...NEXT loop starts by getting the x coordinate for the left border of the next staff position. Its next step is to get the y coordinate for the top of the note region where the note (if any) at that position will be found. You may remember that the position array holds the number of the note region for the note in each staff position. After the note is PUT on the screen, Ledger [SR 7] is called if the note's y coordinate indicates that it is on a ledger line (the “invisible” line just below or just above the staff proper). The subroutine draws the ledger line through the note.

The whole loop has as its limit \( maxN - 1 \). The variable \( maxN \), defined as the number of notes saved in memory plus 1, was set to 1 by InitTune. When InitTune calls DrawScreen, there are no notes to draw since we are initializing a new tune, so the FOR...NEXT loop is not executed. But after the user enters some notes, \( maxN \) will be greater than 1 and this loop will be executed when DrawScreen is called; for example, after instructions have been displayed and the work screen needs to be restored.

After DrawScreen is done, InitTune reads in the note frequencies for the 13 tones from the DATA statements in [Dat 1] into the freq array. Finally, InitTune returns to the main program loop in [Main].

**Placing notes: The inner ”work” loop**

The default activity in MacCompose is placing or deleting notes in the staff area. The other options, including playing the current melody, are handled by button or menu events that are processed as they occur.

The inner WHILE...WEND loop runs as long as newTune is not true. If that flag is set to true (by the user clicking the New button), control falls out of the “work” loop and goes back into the outer loop. InitTune will then be executed again to clear the way for building a new melody.

Let's begin our study of the work loop by looking at the way a melody is built up. The work loop first checks MOUSE(0): If it is 0, the user is not dragging a note with the pointer, or clicking a note to delete it, so the rest of the loop is skipped (by jumping to the label Ignore). MOUSE(0) is then read again to see if anything has happened yet, and so on.

Assuming MOUSE(0) isn’t 0, a FOR...NEXT delay loop is executed so that if a drag is taking place there will be time for the mouse to be moved a reasonable distance. Then MOUSE(0) is checked again, and its value is stored in the variable click.

If click is not -1 (a drag in progress) or 2 (a double-click), the rest of the loop is skipped, since the selection is something not involved with the notes. Otherwise, the current pointer y and x coordinates are obtained and assigned to the variables crntY and crntX.
If the y coordinate is between 35 and 170, the pointer is probably in the staff region (though it may be too far to the left), and the user may be clicking on a note to delete it from the staff. CheckDelete [SR 4] is therefore called to process the deletion.

CheckDelete first gets the note's staff position by dividing the pointer's x coordinate by 20, after subtracting the offset of 40 (representing the distance of the left end of the staff from the left border of the screen). If this is less than 1, the mouse is too far to the left and there is no note to delete, so control jumps to the label WrongNote and the subroutine exits. This also happens if the value in the position array for the staff position is -1, since this means that the pointer is on the staff but there is no note at that position. Otherwise, there is a note at the staff position and note region containing the pointer. The x and y coordinates for the note image are then obtained, and the note is erased by PUTting it with the XOR option. The noteN and position arrays are also updated to show that there is now no note at that position. Since the section of the staff under the former note was also erased, the staffpiece image is PUT to restore the staff lines at that position. If the erased note was on a ledger line, the line is erased by a call to EraLedger [SR 8], which produces an erasure effect by redrawing the line with color 30 (white).

Back in the main program, we skip to the bottom of the loop if the pointer was in the staff area (it was either a deletion, which we've just processed, or the pointer was out of bounds to the left of the staff). We then check to see if the pointer is too high above the staff to be over a note region or too low to be in the symbol area (the area containing the symbols that can be dragged onto the staff). Again, we skip to the bottom of the loop if that is the case.

Finally, if none of those IF statements is true, we are left with the possibility of a note having been selected and dragged toward the staff. The note number is therefore obtained from the mouse pointer's x coordinate. If it is less than 1 or greater than 8, it's not a valid note or symbol (the mouse is to the left or right of the symbol area), so no processing takes place. Otherwise, a valid symbol is being dragged, and HandleNote [SR 3] is called to process it. The subroutine first hides the cursor, which is the usual practice when having an object drag along with the pointer. A WHILE...WEND loop then moves the symbol (by repeatedly drawing and erasing it in XOR mode) as long as the mouse button is down.
At the end of the drag, the mouse pointer's y coordinate is obtained from MOUSE(2) and assigned to \( crntY \). If it is too high or too low to fit on the staff or ledger lines, the subroutine is exited via a jump to the label OutOfRange; otherwise, the note region is obtained from the \( y \) coordinate. Note regions are 10 pixels high. There are also two offsets to be figured in: 40 is added to account for the fact that the staff area starts 40 pixels from the top of the screen, and 35 is subtracted because each note image starts that far above the region occupied by the note body. The reason for this is that notes have stems, while accidentals don't—but this doesn't affect their positioning, since their images are the same size as note images.

Now that the vertical note region has been determined, we must find out which horizontal staff position the note will be placed in. This is done by finding the first staff position that has no note in it, starting from the left. This is because we can't simply use the position following the current value of \( maxN \), because a note may have been deleted somewhere in the middle of the tune. If one has, the new note must be put in the leftmost empty position—that is, the position of the deleted note. (If you don't want the new note put in the position of the deleted note, the simplest thing to do is start your melody over.) The FOR...NEXT loop determines the first empty position, which is a position with a value of \( -1 \) in the \( noteN \) array. The variable \( seqNum \) is set to this number. The number is then used to calculate the \( x \) coordinate for the note image, after which the note can be PUT on the staff. If the note needs a ledger line, Ledger is called to draw it.

AddNote [SR 5] is then called to add the note to the stored melody so that it can eventually be played. AddNote updates the element of the \( position \) array for the note's staff position (8 is added before dividing so that the coordinate can be "rounded" to the nearest region). The note-type number is then put in the position in the \( noteN \) array corresponding to the note's staff position. Thus \( position \) now contains the note's note region (from which its pitch will be obtained), and \( noteN \) contains its type (which will determine its duration if it's a tone or rest, or special processing if it's an accidental). \( maxN \) is also increased by 1 to account for one more note in the current melody if it's not a replacement for a deleted note.

Then, if the currently placed symbol is a sharp or a flat, it does not represent a note itself but affects the previous note. Therefore the note position is only equal to the current sequence position if the current note is less than 7. Finally, AddNote calls PlayNote [SR 6] to play the note.
In PlayNote, if the current note position does not contain a note, or if the symbol at that position is a sharp or flat, no note is sounded and the subroutine returns. Otherwise, depending on the kind of note symbol at the current position, that is, the value of noteN(notePos), control jumps to a routine which either plays a tone, or just waits for the length of the appropriate rest.

If the note type is between 1 and 4 inclusive, it is a regular note or tone. If the symbol at the next position is a sharp or a flat, then freqAdj is set to a value which, when multiplied with the frequency of the base note, will raise or lower it by one half-note, or semitone. Then the routine SOUNDS the note by specifying its frequency (from the freq array) and its duration (determined by its note type referencing the duration array).

If the note type is 5 or 6, the note symbol is a rest. The routine at the label Rest works much like that for playing a tone, with the exception that the frequency played is zero cycles per second, producing a silent pause rather than an audible tone.

This completes the processing of a newly added note or symbol.

PlayNote and then AddNote return, and control goes back to the top of the work loop through HandleNote. The work loop continues until a button or menu event is encountered.

### Button and menu events

Clicking on one of the two buttons causes control to go to ButtonID [BE 0]—the button handler. If a button was indeed pushed, the button number is obtained from DIALOG(1) and the appropriate subroutine is called.

The Play button is handled by Play [BE 1]. This subroutine uses a FOR...NEXT loop to call PlayNote to play each note. Accidentals will be processed as encountered, and the frequency of the corresponding note will be adjusted if needed.

The Start button is handled by Start [BE 2]. This simply sets the newTune flag to true so that control will fall out of the inner work loop and return to the outer main loop [Main]. A new tune will be initialized before the next notes are added.

Menu events are handled by MenuEvent [ME 0]. Quit to BASIC [ME 2] and Quit to Desktop [ME 3] are standard and need not be discussed in detail. Note, however, that Instructions [ME 1] calls DrawScreen to redraw the work screen with any existing staff contents when the user is
done with the instructions. It is important to remember that if you enable a menu option like Instructions that you want available while the user works with the program, you have to make sure you can restore whatever was going on when the program was interrupted.

**Suggestions for MacCompose**

One obvious addition to the program would be an option to save the current melody to disk or restore it from there. You could also expand MacCompose (especially if you have a Fat Mac) to allow several staves’ worth of notes to be handled. There is nothing inherently difficult in doing this, but you *would* have to check for staff boundaries and redraw the staves as needed. You could also add sixteenth and thirty-second notes and longer rests as well.

Another improvement would be to allow the user to set the key signature (the group of sharps or flats often found just following the clef) so you could then just enter notes without remembering to put the appropriate accidental after each note of a given pitch encountered.
MacCompose program listing

```
MacCompose
Musical Composition on the Macintosh

INITIALIZE
IF FRE(0) < 20000 THEN CLEAR, 20000  'force larger data segment
DEFINT a-e : DEFINT g-y
OPTION BASE 1
DIM notePic(80,8), freq(13), duration(6), staffPiece(280)
DIM noteN(30), position(30)
false = 0: true = NOT false
z = 2 ^ (1 / 12)  'semitone factor used to create sharps and flats

FOR n = 1 TO 4  'set up note duration array
    duration(n) = INT(16 / (2 ^ (n - 1)))  'whole = 16, half = 8
NEXT
duration(5) = duration(3) : duration(6) = duration(4)
WAVE 0, SIN  'set sine waveform
GOSUB FormPics
GOSUB TitleScreen  'build note pictures

BUTTON 1, 1, "Play", (10,240) - (60,260)  'install buttons
BUTTON 2, 1, "New", (80,240) - (130,260)
ON DIALOG GOSUB ButtonID : DIALOG ON

MENU 6, 0, 1, "MacCompose"  'install MacCompose menu
MENU 6, 1, 1, "Instructions"
MENU 6, 2, 0, "-"
MENU 6, 3, 1, "Quit to BASIC"
MENU 6, 4, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON

MAIN PROGRAM
WHILE true  'endless loop interrupted by menu events
    newTune = false
    GOSUB InitTune
    WHILE NOT newTune  'loop until user clicks New
        IF MOUSE(0) = 0 THEN Ignore  'do nothing if button not pressed
        FOR t = 1 TO 300 : NEXT  'wait to check for drag
        click = MOUSE(0)
        IF click <> -1 AND click <> 2 THEN Ignore
        crntY = MOUSE(2) : crntX = MOUSE(1)
        IF crntY < 170 AND crntY > 35 THEN GOSUB CheckDelete
        IF crntY < 170 AND crntY > 35 THEN Ignore
        IF crntY > 215 OR crntY < 35 THEN Ignore
```

```
note = INT((crntX + 15) / 50)  
IF note >= 1 AND note <= 8 THEN GOSUB HandleNote

Ignore:
  WEND
  WEND

************************************** MENU EVENTS
MenuEvent:
  'on menu event, branch to appropriate subroutine
  MENU 6, 0, 0
  item = MENU(1)
  ON item GOSUB Instructions, , BasQuit, DeskQuit
  MENU 6, 0, 1
  RETURN

Instructions:
  'display instructions
  WINDOW 2, , (16,55) - (495,325), 2
  PRINT
  PRINT "Instructions for MacCompose"
  PRINT
  PRINT "With MacCompose you can enter sheet music into your Macintosh and play"
  PRINT "it. Do this by 'picking up' the note you want to use with the mouse pointer"
  PRINT "and dragging it to the proper vertical location on the staff. When you"
  PRINT "release the button the note will be placed in the next empty place on the"
  PRINT "staff. Delete notes by moving the pointer over the note you wish to delete"
  PRINT "and clicking: The next note you add will be put into the empty space."
  PRINT
  PRINT "The notes as seen across the screen are: Whole, Half, Quarter, Eighth,"
  PRINT "Quarter rest, Eighth rest, Sharp, and Flat. The Sharp and Flat symbols"
  PRINT "raise or lower the frequency of the notes they are placed after by a"
  PRINT "semitone."
  PRINT
  CALL INITCURSOR
  BUTTON 1, 1, "Continue", (190,240) - (280,260)
  WHILE DIALOG(0) <> 1 : WEND
  BUTTON CLOSE 1 : WINDOW CLOSE 2
  GOSUB DrawScreen
  RETURN

BasQuit:  'quit to BASIC
  MENU RESET
  BUTTON CLOSE 1 : BUTTON CLOSE 2
  WINDOW CLOSE 1
  END

DeskQuit:  'quit to Mac Desktop
  MENU RESET
  SYSTEM
BUTTON EVENTS

ButtonID:

IF DIALOG(0) <> 1 THEN NotButton
btn = DIALOG(1)
ON btn GOSUB Play, Start
NotButton:
RETURN

Play:
FOR notePos = 1 TO maxN - 1
GOSUB PlayNote
NEXT
RETURN

Start:
newTune = true
RETURN

SUBROUTINES

InitTune:
maxN = 1
FOR notePos = 1 TO 30
noteN(notePos) = -1
position(notePos) = -1
NEXT
GOSUB DrawScreen
RESTORE Freq
cancies
FOR n = 1 TO 13 : READ freq(n) : NEXT
RETURN

DrawScreen:
GOSUB DrawClef
GOSUB DrawStaff
FOR n = 1 TO 8
PUT(50 * n, 175), notePic(1, n)
NEXT
FOR n = 1 TO maxN - 1
currX = n * 20 + 40
currY = position(n) * 10
IF noteN(n) <> -1 THEN PUT (currX, currY), notePic(1, noteN(n)), OR
IF currY = 130 OR currY = 10 THEN GOSUB Ledger
NEXT n
RETURN

HandleNote:
CALL HIDE CURSOR
WHILE MOUSE(0) = -1
PUT(MOUSE(1), MOUSE(2)), notePic(1, note)
PUT(MOUSE(1), MOUSE(2)), notePic(1, note)
WEND
crntY = MOUSE(2)
IF crntY > 131 OR crntY < 5 THEN OutOfRange
    crntY = INT((crntY - 35) / 10) * 10 + 40
FOR X = 1 TO maxN
    IF noteN(X) = -1 THEN seqPos = X : X = maxN
NEXT X

IF note < 7 THEN GOTO PutNote
IF seqPos = 1 AND note > 6 THEN OutOfRange
IF note > 6 AND noteN(seqPos - 1) > 4 THEN OutOfRange

PutNote:
    crntX = seqPos * 20 + 40
    PUT(crntX, crntY), notePic(1, note), OR
    "place note on screen"
    "if this is one line above or below staff, draw line"
    IF crntY = 130 OR crntY = 10 THEN GOSUB Ledger
GOSUB AddNote

OutOfRange:
    CALL SHOWCURSOR
    RETURN

CheckDelete:
    IF note < 7 THEN GOTO PutNote
    IF note = 6 AND noteN(seqPos - 1) > 4 THEN OutOfRange
    PUT(crntX, crntY), notePic(1, noteN(note)), XOR
    "recalculate note location"
    "and position on staff"
    IF crntY = 130 OR crntY = 10 THEN GOSUB Ledger
WrongNote:
    RETURN

AddNote:
    position(seqPos) = INT((crntY + 8) / 10)
    noteN(seqPos) = note
    IF seqPos = maxN THEN maxN = maxN + 1
    IF note > 6 THEN notePos = seqPos - 1 ELSE notePos = seqPos
GOSUB PlayNote
RETURN

PlayNote:
    IF noteN(notePos) = -1 OR noteN(notePos) > 6 THEN RETURN
    ON noteN(notePos) GOTO Tone, Tone, Tone, Tone, Rest, Rest
Tone:
    freqAdj = 1
    IF noteN(notePos + 1) = 7 THEN freqAdj = z
    IF noteN(notePos + 1) = 8 THEN freqAdj = 1 / z
    SOUND freq(position(notePos)) * freqAdj, duration(noteN(notePos)), 255
RETURN
Rest:

SOUND 0, duration(noteN(notePos))

RETURN

duration but no volume

Ledger:

'. ___________________________________ draw a LINE above OR below staff

LINE (crntX - 3, crntY + 30) - (crntX + 19, crntY + 30)

RETURN

EraLedger:

'. ___________________________________ erase line above or below

LINE (crntX - 3, crntY + 30) - (crntX + 19, crntY + 30), 30

RETURN

TitleScreen:

GOSUB DrawClef
GOSUB DrawStaff
title$ = "MacCompose - Be a Mozart"
CALL TEXTSIZE(24): CALL TEXTFONT(0): CALL TEXTMODE(1)
RANDOMIZE TIMER
CALL MOVETO(60, 110)
p = 5
FOR i = 1 TO LEN(title$)
  j = INT(RND(1) * 3) - 1
  IF p < 1 THEN j = 1
  IF p > 8 THEN j = -1
  p = p + j
  CALL MOVE(0, j * 10): PRINT MID$(title$, i, 1);
NEXT i
FOR t = 1 TO 8000: NEXT
RETURN

FormPics:

 '. ___________________________________ form notes and put into array

WINDOW 1,, (0, 10) - (530, 340), 2
CIRCLE (20, 200), 8, , , .6
GET (12, 170) - (28, 205), notePic(1, 1)

CIRCLE (50, 200), 8, , , .6
LINE (58, 170) - (58, 200)
GET (42, 170) - (58, 205), notePic(1, 2)

FOR r = 0 TO 8
  CIRCLE (80, 200), r, , , .6
NEXT r
LINE (88, 170) - (88, 200)
GET (72, 170) - (92, 205), notePic(1, 3)

FOR r = 0 TO 8
  CIRCLE (110, 200), r, , , .6
NEXT r
LINE (118, 170) - (118, 200)
LINE (118, 170) - (121, 180)
LINE (121, 180) - (122, 188)
GET (102, 170) - (122, 205), notePic(1, 4)
'quarter rest
LINE (135, 170) - (142, 175)
LINE (142, 175) - (135, 180)
LINE (135, 180) - (145, 190)
LINE (145, 190) - (135, 200)
LINE (135, 200) - (134, 203)
GET (130, 170) - (150, 206), notePic(1, 5)
FOR r = 0 TO 2
  CIRCLE (162, 178), r
NEXT
LINE (160, 180) - (166, 183)
LINE (166, 183) - (172, 180)
LINE (172, 180) - (166, 200)
GET (155, 170) - (175, 206), notePic(1, 6)
ofst = 8
LINE (190, 180 + ofst) - (190, 202 + ofst)
LINE (195, 178 + ofst) - (195, 200 + ofst)
LINE (185, 190 + ofst) - (200, 186 + ofst)
LINE (185, 196 + ofst) - (200, 192 + ofst)
GET (180, 170) - (200, 208), notePic(1, 7)
ofst = 5
LINE (215, 178 + ofst) - (215, 202 + ofst)
LINE (215, 192 + ofst) - (218, 189 + ofst)
LINE (218, 189 + ofst) - (223, 189 + ofst)
LINE (223, 189 + ofst) - (225, 194 + ofst)
LINE (225, 194 + ofst) - (223, 198 + ofst)
LINE (223, 198 + ofst) - (215, 202 + ofst)
GET (210, 170) - (230, 208), notePic(1, 8)
RETURN

DrawClef:
' clear screen, draw treble clef
CLS
RESTORE ClefPoints
READ clefX, clefY
PSET (clefX, clefY)
FOR clef = 1 TO 20
  READ clefX, clefY
  LINE - (clefX, clefY)
NEXT
RETURN

DrawStaff:
' draw staff
FOR staff = 60 TO 140 STEP 20
  LINE (10, staff) - (500, staff)
NEXT
GET (80, 30) - (100, 165), staffPiece
RETURN
Frequencies:

| DATA 880, 784, 698.5, 659.3, 587.3 |
| DATA 523.3, 493.9, 440, 392, 349.2 |
| DATA 329.6, 293.7, 261.6 |

ClefPoints:

| DATA 25, 160, 25, 50, 35, 60, 35, 70, 30, 80, 25, 90, 15, 100, 10, 110 |
| DATA 6, 120, 5, 130, 10, 140, 20, 148, 25, 150, 30, 148, 40, 140, 45, 130 |
| DATA 43, 120, 39, 110, 25, 104, 18, 110, 14, 123 |
Although the admonition of Shakespeare's Polonius—"Neither a borrower nor a lender be"—is still prudent advice, it would be hard to take it literally today. We not only borrow for the occasional big purchase like a house or car, but most of us have charge accounts and credit cards as well. "Automatic" borrowing is built into the very mechanism of modern commerce.
“Very well,” you might say. “I’ll admit I borrow money in various guises, but I’m not really a lender. I’ve been avoiding my freeload ing cousin Harry for years…” Well, think again. If you have a savings account, certificates of deposit (CDs), savings bonds, or similar kinds of financial paper, you’re a lender: You’re lending to the bank, Uncle Sam, or some other institution. A savings account is an open-ended loan with no fixed rate or time period, while CDs and bonds usually have fixed rates and specified periods, but they’re all loans just the same.

If you are a borrower or a lender—and, as you can see, you’re probably both—you need MacFinance. This easy-to-use program will lend you a hand when you want to answer questions like:

- If I take out a $2,500 loan with a two-year term, monthly payments, and an annual rate of 12.5%, how much will my payments be?
- How much am I actually going to pay the bank in interest on a $2,500 loan?
- If I take out a $5,000 CD with a one-year term and a rate of 10.5%, compounded daily, what will its future value (principal plus total interest over term) be?

After you’ve tried MacFinance and studied how it works, you’ll be able to expand its capabilities by adding routines to answer other money questions you might ask frequently. The program is also an excellent example of how to integrate menus and dialog boxes in BASIC programs, and also demonstrate advanced window-management techniques.

How to use MacFinance

When MacFinance is run, the title screen shown in Figure 1 appears, including the MacFinance menu. The menu contains these five options:

![MacFinance Menu](image)

The Borrowing option allows you to calculate how much your payments will be for a loan with specified terms, and Investing shows you how much an investment will be worth at the end of its term. There are also our standard options: Instructions, Quit to BASIC, and Quit to Desktop.
The first time you choose either the Borrowing or Investing option, a window is set up for that calculation, ready for you to enter the necessary information. Once the window is activated, you can move it around the screen and resize it just like the Macintosh Finder's and Microsoft BASIC's windows. You can click the window's close box to make it go away, and when you again choose the same option on the menu the window will reappear with its contents preserved from the last time you used it. You can have both the Borrowing and Investment windows open at once, but only one can be active at a time.

To get a better idea of what MacFinance does after you choose a menu option, picture this little scene. You've just learned that your local computer dealer is having a sale on Macintosh goodies like hard disks and lots of new software. You look over the ad, tote up the prices on your Mac's built-in calculator, and figure in the tax. You note that your dealer, in connection with a local bank, is offering two-year loans at 18.5% to finance purchases over $2,000, and it happens that your purchases (with tax) add up to $2,433.35.

So you go to the dealer and ask him how much the monthly payments will be. He rolls his eyes and groans in anguish as he scrabbles
in a desk drawer for a calculator. He finds it and begins entering figures, muttering as he tries to recall the formula for calculating compound interest. "Uh... just a minute, I think I can lend you a hand," you tell him.

At this point you pull your MacFinance disk out of your pocket, slip it into the dealer's Mac before his startled eyes, run MacFinance and choose Borrowing from the MacFinance menu. A window titled Borrowing appears at the left side of the screen, as shown in Figure 2, ready for you to enter the relevant figures. The little rectangles are standard edit fields like those used for dialog windows and for editing icon names in the Finder. Just click the mouse pointer in a field to put the insertion point there, then start typing. Hitting the Return or Tab key moves you to the next item, and you can also use the standard Cut, Paste, and Copy options. After you fill in the boxes, you simply click the Calculate button and the program computes and displays both the monthly payment amount and the total cost of the loan.

It might happen that you have the cash, but you're considering investing it rather than blowing it on computers. MacFinance can help you here, too. The Investing window in Figure 2 shows you what a CD costing about as much as the loan in the left window would yield. At the respective rates given here, it will cost about $500 to borrow the $2,433.35 for
two years, whereas you could make almost $600 on the investment. Thus it might make more sense to invest the money and finance the computer purchase than to pay cash. To be sure, it's marginal, and other factors might push the argument either way. Since we're here to advise you on BASIC programming, not on managing your money, we'll let it go at that.

Although MacFinance is somewhat limited as a financial planning tool, it is expandable, and studying it closely will reward you with a real mastery of interactive programming on the Macintosh, not to mention window-refreshing techniques.

How MacFinance works

This program configures and enables two windows, one for borrowing and one for lending. Choosing the appropriate menu item opens the window. Once open, windows can be sized, moved, or closed. Only one window is active at a time, but the contents of each window are preserved until the window's close box is clicked.

These windows should be distinguished from the dialog boxes used for many functions in our programs. MacFinance's windows are dynamic, which means that the program must keep track of their status, manage their edit fields, and refresh their contents when they move on and off each other or a dialog box covers them up. These operations are handled by checking for the appropriate dialog event and examining the window's current status, as shown in the program outline in Figure 3.

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Init 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimension arrays for edit fields and prompts</td>
<td></td>
</tr>
<tr>
<td>define functions</td>
<td></td>
</tr>
<tr>
<td>initialize arrays for edit fields (InitFields)</td>
<td></td>
</tr>
<tr>
<td>install MacFinance menu</td>
<td></td>
</tr>
<tr>
<td>enable menu and dialog event-trapping</td>
<td>Init 2 (SR 6)</td>
</tr>
<tr>
<td>open title window</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Program</th>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>create endless loop interrupted by menu and dialog events</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Menu Events</th>
<th>ME 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuEvent:</td>
<td></td>
</tr>
<tr>
<td>suspend dialog event trapping</td>
<td></td>
</tr>
<tr>
<td>disable menu</td>
<td></td>
</tr>
<tr>
<td>branch to appropriate subroutine</td>
<td></td>
</tr>
<tr>
<td>enable menu and resume dialog event-trapping</td>
<td></td>
</tr>
</tbody>
</table>

(continued)

FIGURE 3. Program outline
OpenWind: open window for borrowing or investing
create Calculate button

Instructions: display instructions
open window 3
print instructions
create button, wait for button click
close window

BasQuit: quit to BASIC
DeskQuit: quit to Mac Desktop

Dialog Events

DialogEvent:
suspend menu event trapping
branch to appropriate subroutine
resume menu event trapping

Btn: button clicked in current window
if button clicked, perform calculation (Calculate)

NField: move to a new edit field
save contents of all edit fields (GetEditFields)

NewWindow: make a new window active

GoAway: click in close box of current window
if title window, just allow closing
close Calculate button
reactivate pertinent menu item
set new window flag

Refresh: refresh part of a window
get window number
save last window number
set output to new window
if title window, redraw title screen (TitleScreen)
if not, redraw pertinent window (DrawWindow)
reset output to last window
switch to active edit field for that window

Key: Return to Tab key pressed
save contents of edit fields (GetEditFields)
increment to next field
move to that field

Subroutines

DrawWindow:
display message
activate edit fields in current window (FieldActivate)
set current edit field
if calc. flag set for current window (Calculate)

Figure 3. Program outline (continued)
Initialization

Several arrays need to be established for maintaining the edit fields. *editField$* will contain the numbers the user types into the eight edit fields (four per window). *editName$* contains the descriptive labels, or prompts, that will be printed in front of the edit fields, and also the description of the totals. *wCalc* is an array of two flags that will indicate whether a calculation has been done in a given window. We need to keep track of this so that the program will know whether a calculation has to be redone when a window is refreshed after being covered. Finally, the array *fieldID* holds the edit field ID of the current edit field in each window.

Since anything that is typed into an edit field is treated as a string, numbers in edit fields must be converted to their numeric values before any calculations can be performed with them. The function DollarVal is therefore defined to convert an edit-field string to its dollar value, removing the dollar sign if it’s present. If you sort out the parentheses, you’ll see that the value of the expression $(LEFT$(d$,1) = "\$")$ is being added to the length of $d$, and the RIGHTS$ function is being used to extract that many characters starting from the right. The trick is that if the first character
in \( d\$ \) is a dollar sign, the expression is true and thus has a numeric value of \(-1\). This means that 1 is subtracted from the length of the string, so \( \text{RIGHT}\$ \) will extract everything except the dollar sign. If there is no dollar sign, the expression is false (0), so the full length of the string is extracted. Finally, the VAL (numeric value) of the string is taken.

Next, the TwoDec function is defined, which formats the display of dollar amounts calculated by the program, and the WINDOW CLOSE statement ensures that any remaining window is closed. This is always a good idea, because a program might run out of memory if the environment it happens to run in has a lot of memory tied up in windows and associated edit fields and buttons.

\text{InitFields} [SR 6] is then called to set up the conditions for the starting environment. This involves establishing the contents of the arrays just described, which will provide the window contents when the windows are created. The first pair of nested FOR...NEXT loops in this subroutine puts a dollar sign as the default string in the first edit field for each of the two windows. This will remind the user that the first item in each window is a dollar amount. The inner loop in the pair sets the remaining three edit fields in each window to null or nothing ("""). The first edit field in each window is made the current edit field. The "calculation has been done" flag for each window is initialized to \text{false}, and the "newly entered window" flag is set to \text{true}. (The latter is logical because whichever window is first used is by definition "new.") Finally, the second FOR...NEXT pair reads in the edit-field names from the DATA statements in FieldNames [Dat]. Putting these strings in DATA statements makes it very easy to change them if you want to.

Control now returns to [Init 2]. The MacFinance menu is set up, and menu event-trapping is enabled. Menu events are handled by Menu Event [ME 0]. Dialog events (in this program, these are activities affecting windows and edit fields) are also enabled: They will be handled by DialogEvent [DE 0].

The MacFinance title window is now created, in what proves to be a rather arcane fashion. TitleScreen [SR 7] displays the window straightforwardly enough. The mystery is in how this subroutine gets called. Somehow it has been executed before we get to the main program loop [Main], since the title window appears when the program starts even though no obvious call to TitleScreen has appeared.

Doing a little detective work, we find that the only mention of TitleScreen (other than the subroutine label itself) is in Refresh [DE 5].
It happens that when a window needs to be refreshed, DIALOG(0) returns the value 5. This event also triggers the dialog trap, which sends control to DialogEvent [DE 0]. The ON DIALOG(0) GOSUB here passes the baton to Refresh. Looking at the code in Refresh, we see that TitleScreen is called if the title window (Window 4) needs refreshing. When the window is first created, the window immediately needs to be refreshed, and its contents are displayed. Got all that? Wow!

**Processing menu events**

As previously mentioned, menu events are processed by MenuEvent [ME 0]. The first thing this subroutine does is suspend dialog event-trapping with the DIALOG STOP statement. Note carefully the distinction between suspending dialog event-trapping with DIALOG STOP and disabling it with DIALOG OFF. Suspending dialog event-trapping allows BASIC to continue to record all dialog events but defers dialog event-trapping until DIALOG is turned back ON. At that point, DIALOG(0) will return the latest dialog event in the queue, another DIALOG(0) will return the next most recent event, and so on until all unprocessed events have been obtained. It's like handing each customer a numbered ticket rather than closing the shop each time we wait on someone.

MenuEvent next checks the MENU(0) function to get the number of the menu from which the choice was made. As the program is currently set up, the MacFinance menu (number 6, since menus are numbered starting from 0 at the left) is the only menu that would be in use. However, using MENU(0) rather than just assuming Menu 6 makes it easier to install additional menus if you wish.

The menu selected is then disabled, the item number of the selection (item) is obtained from the MENU(1) function, and the variable windID (window ID) is set to the item number. Since the menu is arranged so that Borrowing is item 1 and Investing item 2, the item number corresponds to the ID of the window that will be used for the selected option. This allows us to use general-purpose routines to manage windows based on their IDs. (The Instructions and the two quit menu operations receive window IDs 3 through 6, but these are not further used, since Instructions uses a fixed window and the quit options don't involve windows at all.)

Let's look now at the way the two major options, Borrowing and Investing, are processed. In both cases, control goes to OpenWind [ME 1]. This subroutine opens the appropriate window as specified by the value of windID. Note the coordinates that put Window 1, titled “Borrowing,” in the left half of the screen and Window 2, titled “Investing,” in the right
half. Also note that the last number in the WINDOW statements indicates
the window is type 1, a “document window,” which is movable and in-
cludes a size box and a title bar. Compare this with the window created in
Instructions [ME 2], which is type 2—a fixed dialog box.

After the selected window is opened, the corresponding menu
item is disabled (dimmed) and the Calculate button is created. You might
wonder how one set of coordinates can specify a button that can go in
either of two windows. The answer is that the coordinates for the rect-
angle specified in the BUTTON statement are always relative to the cur-
rent Output window, so the button automatically goes in the appropriate
position in whichever window is being created.

That’s all the processing that the program does here. BASIC re-
cords all the user activities of entering data and requesting calculation, as
well as manipulating windows once they exist, as dialog events, which will
be processed by DialogEvent [DE 0]. We will note in passing that the In-
structions [ME 2] and the two quit menu events [ME 3] and [ME 4] are set
up as usual for programs in this book, except that the quit options call an
additional subroutine, CleanUp (SR 5) for their janitorial service. Now
let’s go on to explore the many dialog events involved in MacFinance.

**Dialog events**

DialogEvent [DE 0] first suspends menu event-trapping, for the same rea-
sons MenuEvent suspends dialog event-trapping. The current window ID
is obtained from WINDOW(0); this function returns the ID of the window
that is currently active. A window is active when it is first opened, and
again later if the user clicks in it to make it active. The BUTTON and EDIT
FIELD statements and functions are always related to the active window.

The function DIALOG(0) returns the number associated with the
dialog event encountered. The ON...GOSUB statement then transfers con-
trol to the appropriate subroutine.

Now let’s look at the events that must be handled. Rather than
take them in the order they appear in the code (which is determined by the
numbers BASIC assigns to the corresponding dialog events), we’ll break
them down into these two categories: window manipulation, and data en-
try and calculation.

**Window manipulation**

If the user activates a window by clicking in it, NewWindow [DE
3] is called. This subroutine gets the window ID from DIALOG(3), which
has been defined so it will return the window ID of the newly activated window. A WINDOW statement then makes this window the current Output window so that any text that will be displayed during subsequent processing (such as the result of a calculation) will be displayed in this window. Both calculation windows can be on the screen (along with things like the Macintosh calculator and other desk accessories), but only one window will be active and be the current Output window. Field editing and dialog functions will all relate to this window.

If the user clicks the close box to remove a window, GoAway [DE 4] is called. DIALOG(4) contains the ID of the window clicked on. Since the title window has no buttons and is not connected with any menu item, we simply return if that was the window closed. Otherwise, the Calculate button in the current window is removed, the corresponding menu item is reactivated so the user can create this window again later, and the corresponding flag in the newWind array is set to true, indicating that this is a "new" window—which it will be if it's requested and set up again later and we need to restore its contents. (We'll see later why we need to keep track of this “new” window.)

The Macintosh takes care of moving and changing the size of windows, but the programmer is responsible for restoring windows' contents that may have been overwritten, such as by other windows being moved over them. When MacFinance needs to refresh a window, it calls Refresh [DE 5]. In this subroutine, DIALOG(5) returns the ID of the window in question. The variable crntWind is used to save the ID of the current window (which may not be the same as the one that needs refreshing), obtaining it from the WINDOW(1) function. The WINDOW OUTPUT statement then directs output to the window to be refreshed. If the window ID indicates that it is the title window, TitleScreen [SR 7] is called to redraw this window (you may remember that this is how the title window is drawn in the first place).

If the window to be refreshed is not the title screen, DrawWindow [SR 1] will be used to redraw the window. This subroutine first prints the "generic" instructions in the window (again, the coordinates used are relative to the current Output window, so they work in either window). Then a FOR...NEXT loop calls the FieldActivate subprogram [SP 1] four times to activate the four edit fields.

Note that FieldActivate is a subprogram, not a subroutine. This allows us to give it (via the SHARED statement) just the global variables it needs, so we don't have conflict between variables in the subprogram and the main program. The STATIC attribute preserves the values of the
specified variables between calls to the subprogram. The shared data accessed by the subprogram consists of the `editName$` and `editField$` string arrays, which, you may recall, contain the prompts and the default contents respectively of the four edit fields in each of the two windows. The other variables accessed are the `newWind` flag array and `fieldBase` (set in InitFields), which holds the y coordinate of the first edit field.

The y coordinate for the base of the edit field is offset by multiplying the field ID (which will be incremented in turn by the FOR...NEXT loop in the calling routine, DrawWindow) by 30 to get the desired spacing between fields. The variable `left` is set to the coordinate for the left boundary of the edit field; the IF statement is needed because the first field is much wider than the others and thus needs to start farther to the left than the others. The variable `just` is also set; it is used to justify the contents of the field boxes: The first edit field will be left justified, and the rest will be center justified.

The appropriate prompt or label is first printed near the left boundary of the window. The string to be displayed in the window is then obtained. Finally, if the `newWind` flag for the current window ID indicates that it's a "new" window (one just activated or one that was closed and then requested again from the menu), the EDIT FIELD statement is used to establish the edit field with the specified ID, default contents, rectangle coordinates, type, and justification, which uses the value stored in `just`. (If you're not familiar with the options that can be specified in the EDIT FIELD statement and how they determine the appearance and behavior of the edit field, please see the detailed discussion of edit fields in the MacGraph chapter.)

After the edit fields have been established or reactivated and we've returned from the subprogram, DrawWindow uses one more EDIT FIELD statement to make the edit field whose ID is stored in `fieldID` for the currently active window the current edit field. This means that the user can pick up where he or she left off when reactivating a window. DrawWindow then sets the appropriate `newWind` flag to `false`. If a calculation result had been obtained since the window was activated (as indicated by the flag in the `wCalc` array), DrawWindow calls Calculate [SR 2] to perform the calculation again and put the result in the "total" area of the window. (We'll talk more about Calculate a little later, when we discuss what happens when the user clicks the Calculate button.)

Upon return to Refresh, the output is reset to the current window (whose ID we saved in `crntWind`) — which, as we have noted, may not be the same as the window that was refreshed. Also, if the ID indicates that
the window is one of the calculation windows, an EDIT FIELD statement is used to reestablish the current edit field. Again, we must do this because the refreshed window in which we set the current edit field may not be the current Output window, which has the edit field we really want to be current, since this is the window where the user is acting.

Now that we’ve covered how the user’s movement between windows and other window manipulation dialog events are handled, we’re ready to get to the actual work done by the program: getting data and performing calculations. It is often said that modern software uses more code to support the user’s interaction (to be “friendly”) than to actually perform the tasks the program was written for. This recognition of the importance of the human element in computer use is integral to the Macintosh, so it’s not surprising that many Macintosh programs that do relatively little actual data processing (like this one, for example) have a large proportion of “support” code.

### Data entry and calculations

Only one edit field in the current window is active at any one time. The user may type a number in, erase with the Backspace key, select the field, and cut, paste, and copy it with the standard Macintosh editing facilities. These activities are handled automatically by BASIC. However, when the user wants to move to a new edit field by either clicking the pointer in it, or pressing the Tab or Return key, the program must take several actions to keep track of things.

NField [DE 2] is called when the program detects a dialog event indicating that the user clicked in a new edit field. This subroutine calls GetEditFields [SR 4], which uses a FOR...NEXT loop to assign the current contents of the edit fields in the active window to the editField$ array. You will recall that this array is initialized to default strings (the dollar sign for the first field, nothing for the others), but when the user has moved to a new field one or more fields probably have data in them. The data is found via BASIC’s EDIT$ function, which then returns the contents of the specified edit field.

After returning from this subroutine, NField makes the new edit field (whose ID is returned by DIALOG(2)) the new current edit field. It also sets the corresponding current field indicator in fieldID for future use.

The other two ways the user can get into a new edit field are by pressing the Return or the Tab key. These keys also generate dialog events, both of which take us to Key [DE 6]. This processing, like that in NField, calls GetEditFields. Next, the new field ID must be obtained. It can’t be
gotten from a dialog function, as it was with the mouse click, but we know it should be the next ID (or 1, if the user moved from field 4). Thus the current field ID is simply MODded with 4 to get the new ID, our current field ID is updated, and the new field is edited with an EDIT FIELD statement.

After entering data, the user will eventually click on the Calculate button to get a total. The “button click” dialog event sends control to Btn [DE 1], which checks DIALOG(1) to find which button was pressed. If it is the Calculate button, Calculate [SR 2] is called.

This subroutine first calls GetEditFields to ensure that it has the latest data from the edit fields. Then the DollarVal function discussed earlier is used to get the numeric values from the edit fields into the appropriate variables. Because both the loan and the investment calculations use the same variables for principal, number of years, interest rate, and periods per year, the same variables can reasonably be used in the program code for either calculation.

Before the subroutine performs the appropriate calculation, an ON ERROR GOTO statement traps the possible error of division by zero. This statement will call ErrorHandler [SR 3] if an error is detected. If the error is other than division by zero, the ERROR statement in ErrorHandler prints the error number. If the error is division by zero (by far the most likely error here), the total is set to zero and the routine returns to Calculate. (Of course, you could add an error message to make the problem more explicit to the user if you wish.)

Either the loan or the investment calculation is now performed, depending on which window is current. The formulas are too complex to explain here, but we will say that they’re based on the standard compound interest formula used by banks and loan companies.

Finally, the appropriate flag in wCalc is set to indicate that a calculation has been performed in the current window (which may have to be refreshed later). The appropriate label and value are then printed. Note that the value is rounded to two decimal places using the TwoDec function defined in [Init 1]. If you don’t like occasionally getting an answer like $345.2 (instead of $345.20), you could add the appropriate code to include the trailing zero(es).

Suggestions for MacFinance

Unlike traditional programs, which have paths through them rigidly laid out, Macintosh programs that take advantage of the user interface tend to have the kind of “woven” structure we observed in MacFinance, where
the flow of control depends on both menu and dialog events. This structure (or seeming lack of it) may take a little getting used to. However, when you understand how the menu and dialog events interact, you will be rewarded with the ability to use the full potential of that user interface.

It would be fairly easy to add more windows to MacFinance for additional financial calculations. If a new calculation is a variant of the existing ones and also uses four variables, you would only have to add the necessary elements to the arrays and handle the new menu selection. If your calculation is substantially different from ours, you would probably want to generalize the subroutines involving field editing and window refreshing so that you could handle a varying number of edit fields.
MacFinance program listing

MacFinance
Financial Package

INIT 1

CLEAR ,12000
false = 0 ; true = NOT false
OPTION BASE 1
DIM editField$(2,4), editName$(2,6), wCalc(2), fieldID(2)
DEF FN DollarVal(d$) = VAL(RIGHT$(d$, LEN(d$) + (LEFT$(d$,1) = "$")))
DEF FN TwoDec(z) = INT(z * 100 + .5) / 100
INITIALIZE

INIT 2

WINDOW CLOSE 1
GOSUB InitFields
MENU 6, 0, 1, "MacFinance"  
MENU 6, 1, 1, "Borrowing"
MENU 6, 2, 1, "Investing"
MENU 6, 3, 1, "Instructions"
MENU 6, 4, 0, ""
MENU 6, 5, 1, "Quit to BASIC"
MENU 6, 6, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON
ON DIALOG GOSUB DialogEvent : DIALOG ON
WINDOW 4, "MacFinance", (2,40) - (508,337), 1

MAIN

WHILE true : WEND

MENU EVENTS

MenuEvent:
DIALOG STOP
menuBar = MENU(0)
MENU menuBar, 0, 0
item = MENU(1) : windID = item
ON item GOSUB OpenWind, OpenWind, Instructions, , BasQuit, DeskQuit
MENU menuBar,0,1
DIALOG ON
RETURN

OpenWind:
IF windID = 1 THEN WINDOW 1, "Borrowing", (3,40) - (245,250), 1
IF windID = 2 THEN WINDOW 2, "Investing", (250,40) - (509,250), 1
CALL TEXTMODE(1)
MENU 6, windID, 0
BUTTON 1, 1, "Calculate", (10,10) - (90,30), 1
RETURN
Instructions: show user some instructions
WINDOW 3, (16,55) - (495,325), 2
PRINT
PRINT "Instructions for MacFinance"
PRINT
PRINT "This program allows you to calculate both sides of the lending"
PRINT "game. You can see both what your payments would be if you borrowed"
PRINT "money, and how much you would make if you lent it."
PRINT
PRINT "Choose either Borrowing or Investing from the MacFinance menu."
PRINT "For either one, enter the values you wish to try. Then click the Calculate"
PRINT "button in the respective window to see what the result will be."
PRINT
PRINT "While entering numbers, pressing either the Tab or Return key can be"
PRINT "used to move to the next edit box."
PRINT
CALL INITCURSOR
BUTTON 1, 1, "Continue", (190,240) - (280,260)
WHILE DIALOG(0) <> 1 : WEND
BUTTON CLOSE 1 : WINDOW CLOSE 3
RETURN

BasQuit: quit to BASIC
GOSUB CleanUp
END

DeskQuit: quit to Mac Desktop
GOSUB CleanUp
SYSTEM

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
DIALOG EVENTS

DialogEvent:
MENU STOP
windID = WINDOW(0) ' Set current window, and act on dialog event.
ON DIALOG(0) GOSUB Btn, NField, NewWindow, GoAway, Refresh, Key, Key
MENU ON
RETURN

Btn: buttonID = DIALOG(1) 'which button?
IF buttonID = 1 THEN GOSUB Calculate
RETURN

NField: move to a new edit field
GOSUB GetEditFields
EDIT FIELD DIALOG(2)
fieldID(windID) = DIALOG(2)
RETURN
NewWindow: click in a new window
WINDOW DIALOG(3)
RETURN

GoAway: close box clicked in current window
IF DIALOG(4) = 4 THEN RETURN 'don't worry about title window
BUTTON CLOSE 1
MENU 6, DIALOG(4), 1
newWind(DIALOG(4)) = true
RETURN

Refresh: refresh indicated window
windID = DIALOG(5) 'get window number
crtWind = WINDOW(1) 'save current window number
WINDOW OUTPUT(windID) 'set output to window to refresh
IF windID = 4 THEN GOSUB TitleScreen ELSE GOSUB DrawWindow
WINDOW OUTPUT(crtWind) 'reset output to first window
IF crtWind = 1 OR crtWind = 2 THEN EDIT FIELD fieldID(crtWind)
RETURN

Key: Return or Tab key pressed; move to next field
GOSUB GetEditFields
fieldID(windID) = (fieldID(windID) MOD 4) + 1
EDIT FIELD fieldID(windID)
RETURN

SUBROUTINES

DrawWindow: draw window indicated by WindID
CALL MOVETO(96,15) : PRINT "Enter values,"
CALL MOVETO(96,30) : PRINT "then click Calculate."
FOR n = 1 TO 4
    CALL FieldActivate(windID,n)
NEXT
EDIT FIELD fieldID(windID)
newWind(windID) = false
IF wCalc(windID) THEN Calculate
RETURN

Calculate: calculate value for window WindID
GOSUB GetEditFields
prin = FN DollarVal(editField$(windID,1)) 'assign edit field values
years = FN DollarVal(editField$(windID,2))
rate = FN DollarVal(editField$(windID,3))
period = FN DollarVal(editField$(windID,4))

ON ERROR GOTO ErrorHandler 'set error handler in case of div by 0
IF windID <> 1 THEN Investing

growthRate = rate / (100 * period)
amt = prin * growthRate / (1 - 1 / (1 + growthRate) ^ (period * years))
GOTO PrintTotal

Investing:
'investment value
growthRate = 1 + rate / (100 * period)
amt = prin * growthRate ^ (period * years)
'convert decimal to percent
'calculate investment amount

PrintTotal:
ON ERROR GOTO 0
wCalc(windID) = true
'flag calculation for this window

CALL TEXTMODE(0)
LOCATE 11,1 : PRINT editName$(windID,5);
CALL TEXTFACE(1) : PRINT FN TwoDec(amt) : CALL TEXTFACE(0)
PRINT editName$(windID,6);
CALL TEXTFACE(1)
IF windID = 1 THEN PRINT FN TwoDec(amt * period * years - prin)
IF windID = 2 THEN PRINT FN TwoDec(amt - prin)
CALL TEXTMODE(1) : CALL TEXTFACE(0)
RETURN

ErrorHandler: handle Division by Zero error
IF ERR <> 11 THEN ERROR ERR
t = 0
RESUME NEXT

GetEditFields: get active edit fields and save them
FOR n = 1 TO 4
editField$(windID,n) = EDIT$(n)
NEXT
RETURN

CleanUp: clean up before quitting
FOR n = 1 TO 4
WINDOW CLOSE n
EDIT FIELD CLOSE n
NEXT
MENU RESET
RETURN

InitFields: set up startup configuration
fieldBase = 55
FOR windID = 1 TO 2
editField$(windID,1) = "$"
FOR fieldID = 2 TO 4
editField$(windID,fieldID) = ""
NEXT
fieldID(windID) = 1
wCalc(windID) = false
newWind(windID) = true

NEXT windID

RESTORE FieldNames
FOR n = 1 TO 2
  FOR m = 1 TO 6
    READ editName$(n,m)
  NEXT m
NEXT n
RETURN

TitleScreen: display title screen
CALL TEXTFONT(0) : CALL TEXTSIZE(36) : CALL TEXTMODE(1)
CALL MOVETO(112,72) : PRINT "MacFinance"
CALL TEXTSIZE(12)
CALL MOVETO(110,144) : PRINT "Calculate Loan or Lending payments."
CALL TEXTFONT(1)
CALL MOVETO(100,280)
PRINT "Choose an option from the MacFinance Menu."
RETURN

SUB FieldActivate(win,fieldID) STATIC activate an edit field
SHARED editName$(), editField$(), newWind(), fieldBase
b = fieldBase + (fieldID - 1) * 30

IF fieldID = 1 THEN
  left = 150 : just = 1
ELSE left = 180 : just = 2
ENDIF

CALL MOVETO(left, b)
PRINT editName$(win,fieldID);
f$ = editField$(win,fieldID)

IF newWind(win) THEN EDIT FIELD fieldID, f$, (left, b - 15) - (220,b), 1, just
END SUB

FieldNames: names for each edit field for both windows
DATA "Original Loan Balance", "Term of Loan (in years)"
DATA "Annual Percentage Rate", "Payments per year"
DATA "Each payment will be: $", "Total interest paid: $"
DATA "Original Investment", "Number of years"
DATA "Nominal Interest Rate", "Compounding periods / yr."
DATA "Future value will be: $", "Total profit: $"
Most of us have trouble figuring out how far it really is from point A to point B on a map. It's true that maps have scales that show how many actual units (usually miles or kilometers) are represented by a given distance on the map. We can then span the scale with our fingers and try to see approximately how many finger spans cover the map distance between the two points.
of interest. If we want to be more precise in our calculations, we can measure both the scale and the map distance with a ruler, then use the formula (expressed here in miles):
\[
\text{real dist. (miles)} = \text{map dist. (inches)} \times \frac{\text{scale dist. (miles)}}{\text{scale length (inches)}}
\]

Thus, if the scale shows that 100 miles is equal to 3 inches and it's approximately 5 inches on the map from Mud Flats to Rambunctious Junction, we could figure that the actual distance between these points is \(5 \times 100/3\), or about 167 miles.

Well, it happens that the Macintosh comes equipped with a built-in measuring device—the mouse. In order for the mouse to be the precise pointer it is, the Macintosh must convert the distance covered by the mouse to a proportional distance on the screen, measured in screen coordinates. MacMeasure takes advantage of this fact to create an automatic distance-measuring tool that enables you to quickly and easily determine the distances between points on a map.

**How to use MacMeasure**

When you run MacMeasure, you'll see the title screen shown in Figure 1. The MacMeasure menu contains two major options, Calibrate and Measure Trip, as well as the two standard quit options.

**Calibrating the scale**

Before you can measure distances on a map, you must tell MacMeasure the length of the map's scale and the distance the scale length represents. With your chosen map handy, pull down the MacMeasure menu and choose Calibrate. You'll see the screen of instructions shown in Figure 2.

The calibration process is simple. Move the mouse so that the pointer is on the left side of the screen. Put your map on a hard, flat surface and smooth out the wrinkles if there are any, then locate the map scale (usually near the bottom of the map or inset in a little box). Place the mouse so its left edge is lined up with the beginning of the scale. Now hold down the mouse button and slide the mouse along the scale. As you slide the mouse, a line is drawn on the screen (as shown in Figure 2) corresponding to the path of the mouse. This is a "rubber band" line—it stretches out behind the pointer and bends from side to side if the path you're tracing isn't quite straight. If the line isn't straight, simply adjust the mouse position until the line on the screen is straight. This will improve the accuracy.
First calibrate the mouse, then trace trip distances on a map.

FIGURE 1. The MacMeasure title screen and menu

CALIBRATION INSTRUCTIONS
1. Place the mouse on the starting point of the scale on the map.
2. Press the mouse button and hold it down.
3. Slide the mouse along the scale to the number of units you wish to use.
   (The line you form will 'rubberband,' showing the length so far.)
4. Release the mouse button to record the length of the scale.

How many units does this represent?

1

○ Miles ○ Kilometers

FIGURE 2. Calibrating the scale
of your calibration. If you happen to release the mouse button without moving the mouse, an error box appears and informs you that you have attempted to calibrate with a scale of "no length." If this happens, simply click the OK button in the box and start over.

Here's a drawing that illustrates how the four steps of the instructions would be carried out in practice:

![Diagram of calibration steps]

When the left edge of the mouse reaches the end of the scale (or the end of the part you wish to use), release the mouse button. The dialog box shown in Figure 2 appears, asking what real distance (in miles or kilometers) is represented by the scale distance you traced. This is the number marked on the map scale at the point where you released the mouse button when calibrating. A default of 1 mile is shown, and you can edit the number using all the standard editing functions.

Two small "radio" buttons, marked Miles and Kilometers, appear near the bottom of the box. The default is miles; if you want to use kilometers, click on the Kilometers button. Select the unit corresponding to where you stopped calibrating. For example, if your map scale has both miles and kilometers and you released the button at the point indicating 500 miles, you would enter 500 for the number of units and leave the default button for Miles selected.

Clicking the OK button completes the calibration process and leaves you ready to measure distances on the map. Note that if you quit the program and restart, or decide to use a different map, you will have to repeat the calibration procedure.

**Measuring trip distances**

To measure a distance on your map, you need a reference point or mark on the mouse that you can use to make sure that you're tracing precisely between the two points in question. When you calibrate along the map scale, it isn't too hard to trace a straight line using the edge of the mouse, and the
First calibrate the mouse, then trace trip distances on a map.

FIGURE 1. The MacMeasure title screen and menu

CALIBRATION INSTRUCTIONS
1. Place the mouse on the starting point of the scale on the map.
2. Press the mouse button and hold it down.
3. Slide the mouse along the scale to the number of units you wish to use. (The line you form will 'rubberband,' showing the length so far.)
4. Release the mouse button to record the length of the scale.

How many units does this represent?

1

Miles  Kilometers

OK

FIGURE 2. Calibrating the scale
of your calibration. If you happen to release the mouse button *without* moving the mouse, an error box appears and informs you that you have attempted to calibrate with a scale of "no length." If this happens, simply click the OK button in the box and start over.

Here's a drawing that illustrates how the four steps of the instructions would be carried out in practice:

![Diagram of calibration steps](image)

When the left edge of the mouse reaches the end of the scale (or the end of the part you wish to use), release the mouse button. The dialog box shown in Figure 2 appears, asking what real distance (in miles or kilometers) is represented by the scale distance you traced. This is the number marked on the map scale at the point where you released the mouse button when calibrating. A default of 1 mile is shown, and you can edit the number using all the standard editing functions.

Two small "radio" buttons, marked Miles and Kilometers, appear near the bottom of the box. The default is miles; if you want to use kilometers, click on the Kilometers button. Select the unit corresponding to where you stopped calibrating. For example, if your map scale has both miles and kilometers and you released the button at the point indicating 500 miles, you would enter 500 for the number of units and leave the default button for Miles selected.

Clicking the OK button completes the calibration process and leaves you ready to measure distances on the map. Note that if you quit the program and restart, or decide to use a different map, you will have to repeat the calibration procedure.

**Measuring trip distances**

To measure a distance on your map, you need a reference point or mark on the mouse that you can use to make sure that you're tracing precisely between the two points in question. When you calibrate along the map scale, it isn't too hard to trace a straight line using the edge of the mouse, and the
“rubber band” indicates when the line is deviating. On a map, however, points may be at all sorts of angles to each other, since you’ll probably want to trace along highway routes or other lines that have bends in them.

The best solution is to attach a pointer of some sort to the mouse. One quick and easy way of doing this is to find a round toothpick and break it in half. Take one half and wedge it in the outermost groove in the mouse cord’s strain relief so that the point of the toothpick is suspended about a sixteenth of an inch above the surface of the map (see Figure 3). You can put the toothpick, or similar doodad, on either side of the strain relief depending on whether you’re more comfortable with your left or right hand on the mouse. Then you’re ready to start tracing complicated paths on the map. One warning though: Since the tracing pointer is a short distance away from the roller ball on the bottom of the mouse, you must make sure to keep the mouse square with respect to the sides and bottom of the map. In other words, try not to rotate the mouse when tracing.

Once you’ve established a reference point on the mouse, choose Measure Trip from the MacMeasure menu to open the window shown in Figure 4. A representation of the scale you have calibrated is displayed at the top of the window, along with some instructions. As the instructions indicate, you simply position the mouse reference point over the starting point on the map and click once to indicate that this is where measurement is to begin. You will hear a beep, and the trip-counter total will appear at the top of the screen. Initially it will show a trip of 0 miles (or kilometers, as appropriate). As you trace a path on the map, a dotted line appears behind the mouse pointer on the screen, and the trip counter is updated with the distance in your chosen units represented by the map distance you’ve

FIGURE 3. The mouse with toothpick tracing guide
TRIP DISTANCE INSTRUCTIONS
Place the mouse on your map at the starting point of the trip. Press the button once and release it to indicate the starting point. A beep will sound indicating that measurement is now taking place. If you hear a tone rising in pitch as you trace the path of your trip, it means that the pointer is getting near one of the borders of the screen. Press the mouse button to end measurement for that segment of the trip, mark that point on the map, and reposition the pointer on the screen. Make a note of the distance travelled so far, and keep measuring segments of the trip until you are done. Add up the distances for each segment to find the total length of the trip.

FIGURE 4. The Measure Trip instruction window

covered so far. When you've reached your destination on the map, click the mouse button once more to stop measurement.

At some point before you actually begin tracing your route on the map you must make sure that mouse tracking is set to 0. To do this, choose Control Panel from the Apple menu to open this window:
The two buttons labeled 0 and 1 in the mouse-tracking box affect the relationship between the movement of the mouse on a tabletop and the corresponding distance the mouse pointer travels on the screen. If the 1 button is selected, a "velocity component" of mouse movement is added to the actual distance traveled by the mouse. The effect is that faster movements of the mouse result in larger movements of the pointer on the screen, which will throw off the trip counter. Clicking the 0 button, however, establishes a one-to-one correspondence between mouse and pointer movements, resulting in greater accuracy when tracing routes on the map.

You may occasionally find that the mouse pointer reaches the edge of the screen before you've traced the full distance on the map. (It may help to know that the working area of the mouse on the tabletop is about the size of a postcard). Since you're likely to be looking at the map rather than at the screen, MacMeasure provides an audible warning for this situation: As the mouse pointer gets near a screen edge, you'll hear a "chut-chut" tone that will rise in pitch as you get closer to the edge. When you hear this tone, there are two things you can do.

If the map is small enough to turn easily, you can lift the mouse with one hand and rotate the map under it 180 degrees so that the destination is facing back the way you came. You can then continue tracing in that direction, and measurement will continue. Since the trip counter records only distance, not direction, the final total will be the same as if you had traced the whole distance in the same direction.

If the map is rather bulky, an alternative and altogether more workable approach is to mark the point on the map that you've reached, release the mouse button to stop measurement, and note the total on the trip counter. You can then move the mouse pointer to the left or the top of the screen (depending on the direction you're tracing) and measure the rest of the distance. When you're done, simply add the two distances together. In fact, you can use the built-in Macintosh calculator to record and add a series of distances in this fashion.

Unless you have a fairly large map, many repeated measurements or map rotations usually won't be necessary if you start your tracing with the arrow at the left side of the screen (if you're tracing distance in a west-east direction) or at the top of the screen (if you're tracing north-south).

After you finish a measurement, simply choose Measure Trip again if you want to make a new measurement. The trip counter will be reset to 0 automatically. Don't forget to recalibrate if you change maps or want to use a different scale or unit of measurement.
A little more about maps

Maps differ in both scale and projection. Scale is no problem, since MacMeasure can handle a small-scale map (for example, the whole state of California at 80 miles to the inch) just as easily as, say, a large-scale street map of your town, which might have a scale of only a half-mile to the inch. When the scale gets small enough, however (as on a map of the entire United States or the whole world), the problem of projection arises.

We won’t go into the details of the way map projections are made. Let’s just say that, in general, projection is an attempt to represent a round, three-dimensional object (the earth) on a flat piece of paper. In a small area the curvature of the earth doesn’t make much difference, but over longer distances distortion occurs. For example, the most common world maps use the Mercator projection, which tends to make bodies of land near the poles much larger than they are in reality: Greenland is a good example. Other methods of projection represent areas accurately but introduce other types of distortion.

The main reason we mention this is to warn you that MacMeasure will not be as accurate on world maps as it is on maps of the United States or smaller areas. World maps often qualify their scale with a phrase like “along equator” or “along meridian.” In the former case, MacMeasure will measure areas near the equator more accurately than those nearer the poles; north-south distances will be measured more accurately than east-west ones on maps scaled along the meridian.

If you’re really interested in accuracy on small-scale maps, you can find formulas in books of navigation or cartography that provide for adjusting the scale of a map with a given type of projection according to the latitude or longitude at which you are measuring. You could modify MacMeasure’s trip totals using the appropriate formula, or modify the program so it will ask you the latitude and/or longitude and apply the formula to calculate the result automatically.

How MacMeasure works

This program has a very simple structure. There are two main menu events—Calibrate and Measure Trip—and the two Quit options. An endless main loop waits for menu selections. The most interesting feature in the program is the way it uses the mouse as a measuring device. A program outline for MacMeasure is shown in Figure 5.
Initialization
initialize variables
create window
install MacMeasure menu, enable menu event-trapping
display title screen

Main Program
run in idle loop; wait for menu event

Menu Events
MenuEvent: branch to appropriate subroutine

Calibrate: calibrate mouse movement with map distance
show instructions
WHILE a valid distance not entered
wait for a button press
save start and end points of mouse drag (GetMouse)
WHILE waiting for end of drag
update line on screen (GetMouse, DrawLine)
WEND
if distance covered at end of drag is zero,
display error message (ErrorMessage)
WEND
create dialog box
ask user for equivalent map distance
calculate scale from screen distance and map distance
close dialog box

Trip: measure a trip distance on the map
display scale and unit of measurement
show instructions
wait for first button press and release
WHILE mouse button not down
draw a dot at current pointer location
get next pointer position
if pointer in boundary area, sound alarm (Warning)
find relative movement of pointer in pixels
add distance to cumulative total distance in pixels
convert distance to equivalent map distance
set starting point for next movement of pointer
WEND

BasQuit: quit to BASIC

DeskQuit: quit to Mac Desktop

Subroutines
ErrorMessage:
create dialog box (SetupWindow)
display error message
wait for button click, close box

SetupWindow: create dialog box and OK button

GetMouse: get coordinates of start and end of drag

DrawLine: draw given line

FIGURE 5. Program outline
Initialization

In [Init 1] the dimensions of the calibration dialog box are set, and the coordinates of the alarm positions (the left, right, top, and bottom boundaries at which we want to warn that the mouse is too close to the edge of the screen) are also established. The default unit type is set to "miles" (we're old-fashioned). The MacMeasure window is then set up.

The MacMeasure menu is set up in the [Init 2] subroutine and menu event-handling is enabled. [Init 3] draws the title screen (which is so simple that it doesn't need a separate subroutine).

The main program loop [Main] is an endless loop that waits for menu events. Menu events are handled by MenuEvent [ME 0]. The two Quit options [ME 3, ME 4] are standard and need no explanation. Now, with these preliminaries out of the way, let's look at MacMeasure's two main operations: the calibration of the mouse to a map scale and the measurement of distances on a map.

Calibration

The calibration procedure, logically enough, is handled by Calibrate [ME 1a-c]. It starts by clearing the screen and displaying the necessary instructions. In [ME 1b] the distance in pixels the mouse pointer will be moved is stored in the variable `distance`, which is initialized of course as 0. The calibration procedure is framed with an outer WHILE...WEND loop that will repeat until a valid distance is obtained—that is, until the mouse is clicked, dragged some distance, and released.

The first of two inner WHILE...WEND loops waits for the mouse button to be pressed. (Values of MOUSE(0) greater than −1 mean that the mouse is either not active at all or not currently down.) Once the mouse button is detected as being down, GetMouse [SR 3] is called. This subroutine gets two sets of coordinates: The point at which the mouse drag began (x₁,y₁) is obtained with MOUSE(3) and the point the mouse had reached at the time MOUSE(0) was last called (x₂,y₂) is obtained with MOUSE(6). If the drag has ended, the second point will be the location where the drag ended. The first time this subroutine is called, the mouse will have barely moved, but the distance will increase as the mouse is being dragged, so longer lines will be drawn.

Upon return from GetMouse, the mouse's starting coordinates are stored in the variables `startX` and `startY`. The PENMODE is then set to 10, or XOR. This lets us draw a line and then erase it by drawing the same line again, since XOR reverses any pixel that is already black when you
draw over it. DrawLine [SR 4] is now called to draw the first line. DrawLine uses the QuickDraw MOVETO routine to position the pen at the starting coordinates and then uses the QuickDraw LINETO routine to draw a line to the current mouse-pointer location.

A WHILE...WEND loop is then used to repeat this process as long as the mouse continues to be dragged. The first call to DrawLine in each pass has the effect of erasing the current line, due to the effect of XOR just described. GetMouse then gets new coordinates for the mouse, and another call to DrawLine draws a line to the new mouse position. Since this all happens rather quickly, the drawing and redrawing of the line and its following the mouse pointer from side to side if the mouse moves a bit up or down gives a stretching, or “rubber-band” effect.

When the mouse is no longer being dragged, the loop is exited. PENMODE is then set back to the default Copy mode, which draws a black line regardless of the contents of the existing screen, and DrawLine is called one more time to draw the final line representing the length of the scale on the map.

The scale distance is now converted to a unit distance. You may recall from your study of analytic geometry in your school days that the distance between two points, \((x_1, y_1)\) and \((x_2, y_2)\), is equal to the square root of the sum of the squares of the changes in the \(x\) and \(y\) coordinates of the points. In our case we have \((\text{start}X, \text{start}Y)\) as the starting point and \((x_2, y_2)\) as the ending point. Thus the change in the \(x\) coordinate is \(x_2 - \text{start}X\) and the change in the \(y\) coordinate is \(y_2 - \text{start}Y\). Each quantity is then squared, the two quantities are added together, and the square root of the whole shooting match is taken.

The program then checks to see whether the distance is zero: That is, the mouse hasn’t been moved at all during the mouse drag. If this is the case, ErrorMessage [SR 1] is called. This in turn calls SetupWindow [SR 2], which creates a dialog box with an OK button. (Having this box set up by a separate subroutine is handy because we will need it more than once during the program.) After control returns to ErrorMessage, a message is displayed to the effect that the user has created a scale of zero length (which isn’t very useful). ErrorMessage then waits for the OK button to be clicked, after which it closes the dialog box and returns control to Calibrate [ME 1a].

If the distance was 0, control goes back to the top of the loop in Calibrate [ME 1b] so that another attempt at calibration can be made (the user probably accidentally released the mouse button after the initial button press). If a valid distance was obtained, SetupWindow is called to set
up our multipurpose dialog box again. The dialog box then asks the user what the distance the mouse was dragged is supposed to represent. The user will have an opportunity to specify the type of unit when the Miles and Kilometers buttons are set up.

After the *How many units?* prompt is displayed, the EDIT FIELD statement is used to allow the user to enter the requested data. We discuss the use of edit fields extensively in the MacGraph and MacAnimate chapters, so you might refer to them if you’re not familiar with edit fields—they are discussed in even more detail in *Microsoft Macinations* (Microsoft Press, 1985). Here we have only one field to edit, so we don’t have to worry about managing or moving between multiple edit fields.

The default value of one unit is displayed (it is the 1 in quotes that appears as a parameter in the EDIT FIELD statement). The user can use the usual edit functions (cut, paste, backspace, and so on) to change the unit to the desired value. Next, two buttons marked Miles and Kilometers are created. (Note that they are button numbers 2 and 3; button number 1 is the OK button provided by SetupWindow.) The last parameter in the BUTTON statements (3) makes them small “radio” buttons. The second 2 in the statement for button 2 displays the Miles button as preselected (filled in with a circle), indicating that miles is the default unit of measurement. `unitType$` is set to this default, and the flag `pressed` is set to 0.

A WHILE…WEND loop then waits for button presses. If either button 2 or 3 is pressed, the unit type is then set to the appropriate unit. When button 1 (OK) is pressed, the loop is exited.

Finally, the results of the calibration are used to determine the scale for future use at the end of [ME 1c]. The value the user entered for the number of units in the scale is returned by the EDIT$ function (1 is the edit-field number; there’s only one). Then the BASIC VAL function converts this to a numeric value. The length of the scale was the distance the mouse traveled in pixels, obtained earlier. Dividing the number of units by the length of the scale gives the amount of mouse distance corresponding to one “real-world” unit. This value is stored in the variable `scale` and will be used in converting the distance the mouse travels during trip measurement to the actual measured trip distance.

**Measuring a trip**

When the Measure Trip menu option is chosen, Trip [ME 2a] is used to calculate the distance between two points on the map. For this to work, the user must have calibrated the mouse for the current map as described in the previous section.
Trip first clears the screen and then displays the unit of measurement in the current calibrated scale. A series of calls to the QuickDraw LINE and MOVE routines is used to show a line the length of the current scale. The instructions for measuring trip distance are then displayed.

The variable distance in [ME 2b], which will be used to keep track of the accumulating trip length, is set to 0. The "dummy" statement that follows, which simply gets any pending mouse event from MOUSE(0), is a useful precaution. No action is performed by this event; the purpose of the statement is to ensure that there isn't a leftover mouse click that would cause measurement to begin before the user is really ready.

A WHILE...WEND loop is now used to wait until the mouse button is pressed and then released. This loop uses the condition WHILE MOUSE(0) < 1 to serve two purposes: (1) wait for the mouse button to go down, and (2) wait for the button to be released again. As long as the mouse button hasn't been touched at all, MOUSE(0) will be 0, which is less than 1. Once the button is down, MOUSE(0) will be less than 0 as long as it remains down. When the button is released, however, MOUSE(0) will return a value of 1 or greater (the actual values depend on whether single-, double-, or triple-clicking occurred—normally it would be a single click). When the release is detected, the loop is exited and a beep is sounded to let the user know that measurement of mouse movement will now begin.

In [ME 2c], the initial mouse x and y coordinates are obtained from MOUSE(1) and MOUSE(2), respectively. Control then drops into a WHILE...WEND loop that will handle the tracing of the trip and its measurement: The loop runs as long as the mouse button is not down, which, as you recall from the user instructions, is the signal to stop measuring.

A PSET statement is used to draw a point at the starting position (posX,posY). The mouse position is then checked again (it probably will have moved), and the values returned (the new mouse position) are assigned to (newX,newY).

Next, the program determines whether the user must be warned that the mouse is too close to an edge of the screen. Four IF...THEN statements check the boundaries in the four directions, and in each case the Warning subprogram [SP 1] is called if the boundary is exceeded (recall that the alarm boundaries were set back in [Init 1]). The boundaries are far enough back from the edges of the screen to provide enough warning for the user to stop tracing before hitting the actual screen edge.

The Warning subprogram is passed a value equal to the distance the mouse has gone past the warning boundary toward the edge of the
If you examine the SOUND statement used in Warning, you can see that this has the effect of increasing both the pitch and the volume of the sound as the mouse gets farther from the boundary and moves closer to the edge of the screen.

After any necessary warning is given, the loop continues and the distance between the last reading of the mouse’s position and its new position is calculated. The first time through the loop, this will be the distance from the position read just before entering the loop; on subsequent passes, the distance will be the difference between points reached in the preceding and the current pass. The calculation used here is similar to that used in Calibrate [ME 1a]: The square root of the sum of the squares of the distances is added to the previous total distance to get the new total distance.

Next, the program prints the current distance on the “trip counter” display in the upper left corner of the screen. First, the distance in screen (pixel) units is converted to the “real-world” distance by multiplying it by the scale established as a result of the earlier scale calibration. The result is expressed in the proper units up to two decimal places.

After the current trip total is displayed, the current mouse (x,y) coordinates are stored as the “old” coordinates (posX,posY), and the loop executes again if the mouse button has not been clicked. When the mouse button is clicked, the loop is exited. A beep signals that measurement is finished, for the mouse is presumably at its destination.

Suggestions for MacMeasure

MacMeasure is a useful tool for looking at historical maps. You can measure the progress of military campaigns, for example. While it may not look very impressive for General Lee to lead a tattered band of Confederate troops four-and-a-half map inches in three days, you might get a better feeling for the effort involved if MacMeasure informs you the distance marched was 102 miles.

You could write programs using routines like MacMeasure’s for other measuring tasks. For example, you could write a program that would let you trace a dress pattern, show it on the screen, and print the various dimensions that would be obtained if you reduced or expanded it by so many sizes. You could also make a version of MacMeasure that measures areas and use it to calculate the areas of rectangular plots such as city lots or farm fields. You can be master of all your mouse surveys!
draw over it. DrawLine [SR 4] is now called to draw the first line. DrawLine uses the QuickDraw MOVETO routine to position the pen at the starting coordinates and then uses the QuickDraw LINETO routine to draw a line to the current mouse-pointer location.

A WHILE...WEND loop is then used to repeat this process as long as the mouse continues to be dragged. The first call to DrawLine in each pass has the effect of erasing the current line, due to the effect of XOR just described. GetMouse then gets new coordinates for the mouse, and another call to DrawLine draws a line to the new mouse position. Since this all happens rather quickly, the drawing and redrawing of the line and its following the mouse pointer from side to side if the mouse moves a bit up or down gives a stretching, or "rubber-band" effect.

When the mouse is no longer being dragged, the loop is exited. PENMODE is then set back to the default Copy mode, which draws a black line regardless of the contents of the existing screen, and DrawLine is called one more time to draw the final line representing the length of the scale on the map.

The scale distance is now converted to a unit distance. You may recall from your study of analytic geometry in your school days that the distance between two points, \((x_1,y_1)\) and \((x_2,y_2)\), is equal to the square root of the sum of the squares of the changes in the \(x\) and \(y\) coordinates of the points. In our case we have \((startX,startY)\) as the starting point and \((x_2,y_2)\) as the ending point. Thus the change in the \(x\) coordinate is \(x_2 - startX\) and the change in the \(y\) coordinate is \(y_2 - startY\). Each quantity is then squared, the two quantities are added together, and the square root of the whole shooting match is taken.

The program then checks to see whether the distance is zero: That is, the mouse hasn’t been moved at all during the mouse drag. If this is the case, ErrorMessage [SR 1] is called. This in turn calls SetupWindow [SR 2], which creates a dialog box with an OK button. (Having this box set up by a separate subroutine is handy because we will need it more than once during the program.) After control returns to ErrorMessage, a message is displayed to the effect that the user has created a scale of zero length (which isn't very useful). ErrorMessage then waits for the OK button to be clicked, after which it closes the dialog box and returns control to Calibrate [ME 1a].

If the distance was 0, control goes back to the top of the loop in Calibrate [ME 1b] so that another attempt at calibration can be made (the user probably accidentally released the mouse button after the initial button press). If a valid distance was obtained, SetupWindow is called to set
up our multipurpose dialog box again. The dialog box then asks the user what the distance the mouse was dragged is supposed to represent. The user will have an opportunity to specify the type of unit when the Miles and Kilometers buttons are set up.

After the *How many units?* prompt is displayed, the EDIT FIELD statement is used to allow the user to enter the requested data. We discuss the use of edit fields extensively in the MacGraph and MacAnimate chapters, so you might refer to them if you're not familiar with edit fields—they are discussed in even more detail in *Microsoft Macinations* (Microsoft Press, 1985). Here we have only one field to edit, so we don't have to worry about managing or moving between multiple edit fields.

The default value of one unit is displayed (it is the 1 in quotes that appears as a parameter in the EDIT FIELD statement). The user can use the usual edit functions (cut, paste, backspace, and so on) to change the unit to the desired value. Next, two buttons marked Miles and Kilometers are created. (Note that they are button numbers 2 and 3; button number 1 is the OK button provided by SetupWindow.) The last parameter in the BUTTON statements (3) makes them small “radio” buttons. The second 2 in the statement for button 2 displays the Miles button as preselected (filled in with a circle), indicating that miles is the default unit of measurement. `unitType$` is set to this default, and the flag `pressed` is set to 0.

A WHILE...WEND loop then waits for button presses. If either button 2 or 3 is pressed, the unit type is then set to the appropriate unit. When button 1 (OK) is pressed, the loop is exited.

Finally, the results of the calibration are used to determine the scale for future use at the end of [ME 1c]. The value the user entered for the number of units in the scale is returned by the EDIT$ function (1 is the edit-field number; there's only one). Then the BASIC VAL function converts this to a numeric value. The length of the scale was the distance the mouse traveled in pixels, obtained earlier. Dividing the number of units by the length of the scale gives the amount of mouse distance corresponding to one “real-world” unit. This value is stored in the variable `scale` and will be used in converting the distance the mouse travels during trip measurement to the actual measured trip distance.

### Measuring a trip

When the Measure Trip menu option is chosen, Trip [ME 2a] is used to calculate the distance between two points on the map. For this to work, the user must have calibrated the mouse for the current map as described in the previous section.
Trip first clears the screen and then displays the unit of measurement in the current calibrated scale. A series of calls to the QuickDraw LINE and MOVE routines is used to show a line the length of the current scale. The instructions for measuring trip distance are then displayed.

The variable distance in [ME 2b], which will be used to keep track of the accumulating trip length, is set to 0. The "dummy" statement that follows, which simply gets any pending mouse event from MOUSE(0), is a useful precaution. No action is performed by this event; the purpose of the statement is to ensure that there isn't a leftover mouse click that would cause measurement to begin before the user is really ready.

A WHILE...WEND loop is now used to wait until the mouse button is pressed and then released. This loop uses the condition WHILE MOUSE(0) < 1 to serve two purposes: (1) wait for the mouse button to go down, and (2) wait for the button to be released again. As long as the mouse button hasn't been touched at all, MOUSE(0) will be 0, which is less than 1. Once the button is down, MOUSE(0) will be less than 0 as long as it remains down. When the button is released, however, MOUSE(0) will return a value of 1 or greater (the actual values depend on whether single-, double-, or triple-clicking occurred—normally it would be a single click). When the release is detected, the loop is exited and a beep is sounded to let the user know that measurement of mouse movement will now begin.

In [ME 2c], the initial mouse x and y coordinates are obtained from MOUSE(1) and MOUSE(2), respectively. Control then drops into a WHILE...WEND loop that will handle the tracing of the trip and its measurement: The loop runs as long as the mouse button is not down, which, as you recall from the user instructions, is the signal to stop measuring.

A PSET statement is used to draw a point at the starting position (posX,posY). The mouse position is then checked again (it probably will have moved), and the values returned (the new mouse position) are assigned to (newX,newY).

Next, the program determines whether the user must be warned that the mouse is too close to an edge of the screen. Four IF...THEN statements check the boundaries in the four directions, and in each case the Warning subprogram [SP 1] is called if the boundary is exceeded (recall that the alarm boundaries were set back in [Init 1]). The boundaries are far enough back from the edges of the screen to provide enough warning for the user to stop tracing before hitting the actual screen edge.

The Warning subprogram is passed a value equal to the distance the mouse has gone past the warning boundary toward the edge of the
If you examine the SOUND statement used in Warning, you can see that this has the effect of increasing both the pitch and the volume of the sound as the mouse gets farther from the boundary and moves closer to the edge of the screen.

After any necessary warning is given, the loop continues and the distance between the last reading of the mouse's position and its new position is calculated. The first time through the loop, this will be the distance from the position read just before entering the loop; on subsequent passes, the distance will be the difference between points reached in the preceding and the current pass. The calculation used here is similar to that used in Calibrate [ME 1a]: The square root of the sum of the squares of the distances is added to the previous total distance to get the new total distance.

Next, the program prints the current distance on the “trip counter” display in the upper left corner of the screen. First, the distance in screen (pixel) units is converted to the “real-world” distance by multiplying it by the scale established as a result of the earlier scale calibration. The result is expressed in the proper units up to two decimal places.

After the current trip total is displayed, the current mouse (x,y) coordinates are stored as the “old” coordinates (posX,posY), and the loop executes again if the mouse button has not been clicked. When the mouse button is clicked, the loop is exited. A beep signals that measurement is finished, for the mouse is presumably at its destination.

Suggestions for MacMeasure

MacMeasure is a useful tool for looking at historical maps. You can measure the progress of military campaigns, for example. While it may not look very impressive for General Lee to lead a tattered band of Confederate troops four-and-a-half map inches in three days, you might get a better feeling for the effort involved if MacMeasure informs you the distance marched was 102 miles.

You could write programs using routines like MacMeasure’s for other measuring tasks. For example, you could write a program that would let you trace a dress pattern, show it on the screen, and print the various dimensions that would be obtained if you reduced or expanded it by so many sizes. You could also make a version of MacMeasure that measures areas and use it to calculate the areas of rectangular plots such as city lots or farm fields. You can be master of all your mouse surveys!
MacMeasure

Use the mouse to measure map distances.

```
false = 0 : true = NOT false
xCen = 256 : size = 150 'size of dialog box
alarmL = 80 : alarmR = 420 'left and right alarm trigger positions
alarmU = 100 : alarmD = 240 'up and down alarm trigger positions
unitType$ = "Miles"
```

WINDOW 1, "MacMeasure", (3,22) - (509,340), 3

MENU 6, 0, 1, "MacMeasure" 'install MacMeasure menu
MENU 6, 1, 1, "Calibrate"
MENU 6, 2, 1, "Measure Trip"
MENU 6, 3, 0, "-
MENU 6, 4, 1, "Quit to BASIC"
MENU 6, 5, 1, "Quit to Desktop"
ON MENU GOSUB MenuEvent : MENU ON

CLS : CALL TEXTFACE(1) : CALL TEXTSIZE(36) 'draw Title Screen
LOCATE 3,5 : PRINT "MacMeasure!"
CALL TEXTSIZE(12) : LOCATE 12,13
PRINT "First calibrate the mouse,"
LOCATE 14,13
PRINT "then trace trip distances on a map."
CALL TEXTFACE(0)

WHILE true : WEND 'do nothing but wait for menu event

```
 MenuEvents:
 MENU 6, 0, 0 item = MENU(1)
 ON item GOSUB Calibrate, Trip, , BasQuit, DeskQuit
 MENU 6, 0, 1 're-enable MacMeasure menu
 RETURN
```

Calibrate:

```
CLS
LOCATE 3,1
PRINT "CALIBRATION INSTRUCTIONS"
PRINT "1. Place the mouse on the starting point of the scale on the map."
```
PRINT "2. Press the mouse button and hold it down."
PRINT "3. Slide the mouse along the scale to the number of units you wish to use."
PRINT "4. Release the mouse button to record the length of the scale."

distance = 0 'set dummy distance to start loop
WHILE distance = 0 'loop until valid distance is entered
    WHILE MOUSE(0) > -1 : WEND 'wait for button press and drag
    GOSUB GetMouse 'get coordinates
    startX = x1: startY = y1
    CALL PENMODE(10) 'set XOR penmode
    GOSUB DrawLine 'draw first line
    WHILE MOUSE(0) < 0 'while dragging
        GOSUB DrawLine 'get new coordinates
        GOSUB GetMouse 'draw new line
        GOSUB DrawUne
        GOSUB GetMouse
        GOSUB DrawLine
    WEND

    CALL PENMODE(8) 'set COPY penmode
    GOSUB DrawLine 'draw new line
    distance = SQR((x2 - startX)^2 + (y2 - startY)^2) 'while distance = 0
    IF distance = 0 THEN GOSUB ErrorMessage 'loop until valid distance is entered

GOSUB SetupWindow 'create dialog box for setting scale
CALL TEXTFONT(0)
LOCATE 2, 4: PRINT "How many units does this represent?"
EDIT FIELD 1, "1", (size - 15, 40) - (size + 15, 55), 2, 2
BUTTON 2, 2, "Miles", (20,70) - (100,90), 3
BUTTON 3, 1, "Kilometers", (110,70) - (210,90), 3
unitType$ = "Miles"

pressed = 0
WHILE pressed <> 1 'repeat until OK button pressed
    WHILE DIALOG(0) <> 1 : WEND 'wait for button press
    pressed = DIALOG(1)
    IF pressed = 2 THEN BUTTON 2,2 : BUTTON 3,1 : unitType$ = "Miles"
    IF pressed = 3 THEN BUTTON 2,1 : BUTTON 3,2 : unitType$ = "Kilometers"
WEND

units = VAL(EDIT$(1))
unitLength = distance
scale = units / distance
WINDOW CLOSE 2 'close dialog box for setting scale
RETURN

Trip: "" measure trip
CLS
LOCATE 2,1 : PRINT units " " unitType$ 'print calibrated units
MAC MEASURE

LOCATE 3,2
CALL LINE(0,-10) : CALL MOVE(0,5)
CALL LINE(unitLength,0)
CALL MOVE(0,-5) : CALL LINE(0,10)
LOCATE 5,1
PRINT "TRIP DISTANCE INSTRUCTIONS"
PRINT "Place the mouse on your map at the starting point of the trip."
PRINT "Press the button once and release it to indicate the starting point."
PRINT "A beep will sound indicating that measurement is now taking place."
PRINT "If you hear a tone rising in pitch as you trace the path of your trip, "
PRINT "it means that the pointer is getting near one of the borders of the screen."
PRINT "Press the mouse button to end measurement for that segment of the trip,"
PRINT "mark that point on the map, and reposition the pointer on the screen.
PRINT "Make a note of the distance travelled so far, and keep measuring segments"
PRINT "of the trip until you are done.
PRINT "Add up the distances for each segment to find the total length of the trip."

\[ \text{distance} = 0 \]  \quad \text{initialize distance}
\text{dummy} = \text{MOUSE}(0)  \quad \text{clear out any pending mouse events}
\text{WHILE} \text{MOUSE}(0) < 1 : \text{WEND}  \quad \text{wait until button is pressed, then released}
SOUND 1000,1 : CLS  \quad \text{make sound and clear screen to see path traced}

\text{posX} = \text{MOUSE}(1) : \text{posY} = \text{MOUSE}(2)  \quad \text{set x and y start position}
\text{WHILE} \text{MOUSE}(0) = 0  \quad \text{while button not down}
PSET (\text{MOUSE}(1), \text{MOUSE}(2))  \quad \text{draw mouse position}
\text{newX} = \text{MOUSE}(1) : \text{newY} = \text{MOUSE}(2)  \quad \text{new x and y}
IF \text{newX} < \text{alarmL} \quad \text{THEN} \text{CALL Warning(\text{alarmL} - \text{newX})} : \text{GOTO PastWarn}
IF \text{newX} > \text{alarmR} \quad \text{THEN} \text{CALL Warning(\text{newX} - \text{alarmR})} : \text{GOTO PastWarn}
IF \text{newY} + 40 < \text{alarmU} \quad \text{THEN} \text{CALL Warning(\text{alarmU} - (\text{newY} + 40))} : \text{GOTO PastWarn}
IF \text{newY} > \text{alarmD} \quad \text{THEN} \text{CALL Warning(\text{newY} - \text{alarmD})}

\text{PastWarn:}
\text{dx} = \text{ABS}(\text{newX} - \text{posX})
\text{dy} = \text{ABS}(\text{newY} - \text{posY})
distance = \text{SQR}(\text{dx} ^ 2 + \text{dy} ^ 2) + \text{distance}  \quad \text{set cumulative distance}
\text{Trip} = \text{INT}(\text{scale} \times \text{distance} \times 100) / 100  \quad \text{show distance}
\text{LOCATE} 1,1 : \text{PRINT} "The trip is " Trip \" " \text{unitType$}
\text{posX} = \text{newX} : \text{posY} = \text{newY}
\text{WEND}
\text{SOUND} 1000,1
\text{RETURN}

\text{BasQuit:}  \quad \text{quit to BASIC}
\text{MENU} \text{RESET}
\text{END}
DeskQuit: quit to Mac DesktTop

MENU RESET
SYSTEM

** SUBROUTINES **

ErrorMessage:
GOSUB SetupWindow
CALL TEXTFONT(0): PRINT
PRINT "You have made an attempt to calibrate the"
PRINT "planimeter with a scale of no length."
LOCATE 5,8: PRINT "Please try again."
WHILE DIALOG(0) <> 1: WEND
CALL TEXTFONT(1)
WINDOW CLOSE 2
RETURN

SetupWindow:
WINDOW 2, , (xCen - size, 200) - (xCen + size, 300), -2
BUTTON 1, 1,"OK", (2 * size - 50, 70) - (2 * size - 10, 90)
RETURN

GetMouse:
\[ x1 = \text{MOUSE}(3) : y1 = \text{MOUSE}(4) \]  
\[ x2 = \text{MOUSE}(5) : y2 = \text{MOUSE}(6) \]  
RETURN

DrawLine:
CALL MOVETO(x1,y1)
CALL LINETO(x2,y2)
RETURN

SUB Warning(alarm) STATIC  
\[ \text{SOUND} 50 + \text{alarm} * 4, 1, 50 + \text{alarm} * 2 \]  
END SUB
ABOUT THE AUTHORS

Mitchell Waite is president of The Waite Group, Inc., a computer-book publishing firm renowned for producing well-written books. Best-sellers from The Waite Group include *UNIX Primer Plus*, *C Primer Plus*, and *Assembly Language Primer for the IBM PC*. Fluent in a variety of computer languages, Mitchell has been involved in the personal computer industry since 1976, when he bought an Apple I computer from Steve Jobs.

Chuck Blanchard is a consultant and engineer with several years of programming experience. Chuck has worked extensively with BASIC, Pascal, FORTH, LISP, and many assembly languages, and is currently working on a project aimed at developing a visual programming language for the Macintosh.

Don Urquhart is an instructor in Electronics Technology at the College of Marin in Kentfield, California, where he conducts courses in analog and digital systems. Before becoming a teacher, Don spent 15 years as an aerospace engineer.

Dan Putterman took his first college computer course at the age of thirteen. An accomplished programmer, he now teaches and consults on the use of microcomputers. He is the author of the FrontEnd, a popular DOS utilities program for the IBM PC, and a co-founder of Innovative Techniques, a microcomputer-software company based in Marin County, California.
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 Greg Hickman. Design executed on the DICOMED IMAGINATOR™ computer-graphics system at Pacific Color, Seattle, Washington.

Interior text design by Karen-Lynne de Robinson and Christopher Banks. The high-resolution screen displays were created on the Apple Macintosh and printed on the Hewlett-Packard LaserJet.

Text composition by Microsoft Press in Rotation with display in Erbar Light Condensed, using the CCI composition system and the Mergenthaler Linotron 202 digital phototypesetter.
OTHER TITLES FROM MICROSOFT PRESS

THE APPLE ENVIRONMENT

*The Apple Macintosh Book, Second Edition*
by Cary Lu $19.95

*Excel in Business*  Number-Crunching Power on the Apple Macintosh
by Douglas Cobb and the Cobb Group $22.95

*The Printed Word*  Professional Word Processing with Microsoft Word
 on the Apple Macintosh
by David A. Kater and Richard L. Kater $17.95

*MacWork MacPlay*  Creative Ideas for Fun and Profit on Your Apple Macintosh
by Lon Poole $18.95

*Presentation Graphics on the Apple Macintosh*  How to use Microsoft Chart to
 Create Dazzling Graphics for Professional and Corporate Applications
by Steve Lambert $18.95

*Microsoft Macinations*  An Introduction to Microsoft BASIC for the
 Apple Macintosh
by The Waite Group, Mitchell Waite, Robert Lafore, and Ira Lansing $19.95

*Creative Programming in Microsoft BASIC for Optimal Macintosh Performance*
by Steve Lambert $18.95

*Microsoft Multiplan: Of Mice and Menus*  Models for Managing Your Business
 with the Apple Macintosh
by The Waite Group, Bill Bono, and Ken Kalkis $16.95

*Command Performance: Multiplan on the Apple Macintosh*  The Microsoft
 Desktop Dictionary and Cross-Reference Guide
by Eddie Adamis $19.95

*Inside MacPaint*  Sailing Through the Sea of FatBits on a Single-Pixel Raft
by Jeffrey S. Young $18.95

*AppleWorks*  Boosting Your Business with Integrated Software
by Charles Rubin $16.95

*The Endless Apple*  How to Maintain State-of-the-Art Performance on
 Your Apple II and IIe
by Charles Rubin $15.95

GENERAL

*Online*  A Guide to America's Leading Information Services
by Steve Lambert $19.95

*Silicon Valley Guide to Financial Success in Software*
by Daniel Remer, Paul Remer, and Robert Dunaway $19.95

by "The Cracker" (Bill Landreth)  
softcover $9.95
hardcover $19.95

*A Much, Much Better World*
by Eldon Dedini $6.95
Now... available only by ordering directly from Microsoft Press... here is the authorized Companion Disk for Macintosh Midnight Madness. Save yourself hours of work tediously typing in line after line of program code... and many more hours finding and fixing those frustrating and inevitable typos. With the Companion Disk, you'll get clean, ready-to-run programs for all 17 projects in Macintosh Midnight Madness, plus these additional files not included in the book:

- 12 clever animation sequences for MacAnimate
- 17 inventive cursor patterns for the MacCursor program

Companion Disk for Macintosh Midnight Madness only $19.95
ORDER TODAY!

ORDER CARD

YES... please send me _______ copies of the Companion Disk for Macintosh Midnight Madness at $19.95 each. (US funds only.)

Postage and handling charges: $1.00 per disk (domestic orders)
$2.00 per disk (foreign orders)

California residents add 6.5% sales tax; Washington state residents add 8.1% sales tax.

Name ______________________________________________________ (Please print)
Address ____________________________________________________
City __________________________________ State _______ ZIP ______

Payment: □ Check □ Money Order □ VISA □ American Express □ MasterCard

Credit Card No. __________________ ________ ________ Exp. Date ____________
Signature ____________________________________________________

Please allow 4 weeks for delivery.
Here are just a few of the programs included on your Companion Disk for Macintosh Midnight Madness:

- MacGurkha
  Land in the target zone.

This Companion Disk is only available directly from Microsoft Press. To order, use the card below, or, if the card has already been used, refer to the introduction for ordering instructions.

Name_________________________
Address_________________________

Microsoft Press
Box 97200
10700 Northup Way
Bellevue, WA 98009
MACINTOSH™ MIDNIGHT MADNESS joins the growing Microsoft Press library of high-quality, critically-acclaimed books on the Macintosh.

Explore the far-reaching horizons of Microsoft BASIC for the Apple Macintosh! Here is a wealth of clever, useful, and amazing BASIC programs—17 in all—that take full advantage of this feature-rich language. Complete with programming insights, techniques, and listings from The Waite Group, Macintosh Midnight Madness offers these valuable and inventive programs: a fast-moving animation program, a streamlined business graphics program, a graphics "construction set," a music composition program, and much more.

U.S.A.  $18.95
U.K.    £15.95
AUST.   $28.95
       (recommended)
CAN.    $28.95

ISBN  0-914845-30-6