MACINTOSH™ PROGRAMMING USING MS-BASIC™ 2.0

RICHARD C. VILE, JR.

No. 2621
$24.95

MACINTOSH™ PROGRAMMING USING MS-BASIC™ 2.0

RICHARD C. VILE, JR.

TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

To my parents with love and appreciation.

Macintosh is a trademark licensed to Apple Computer, Inc.
MacWrite is a trademark of Apple Computer, Inc.
MacPaint is a trademark of Apple Computer, Inc.
Apple is a registered trademark of Apple Computer, Inc.
The Finder is a trademark of Apple Computer, Inc.
Microsoft is a registered trademark of Microsoft Corporation.
MS is a trademark of Microsoft Corporation.

TAB BOOKS Inc. offers software for sale. For information and a catalog, please contact TAB Software Department, Blue Ridge Summit, PA 17294-0850.

FIRST EDITION
FIRST PRINTING

Copyright © 1985 by TAB BOOKS Inc.
Printed in the United States of America

Reproduction or publication of the content in any manner, without express permission of the publisher, is prohibited. No liability is assumed with respect to the use of the information herein.

Library of Congress Cataloging in Publication Data

Vile, Richard C., 1943-
Macintosh programming using MS—BASIC 2.0.

1. Macintosh (Computer)—Programming. 2. BASIC (Computer program language) I. Title.
QA76.8.M3V55 1985 005.2'65 85-22218
ISBN 0-8306-0321-2
ISBN 0-8306-0221-6 (pbk.)

Cover photograph courtesy of Apple Computer, Inc.

Contents

	Acknowledgments	v
	Program Titles	vi
	Introduction	viii
	Section 1 Getting Started	1
1	A Tour of MS-BASIC on the Mac	1
2	Programs, Programs, Programs	7
	Section 2 Learning the Ropes	11
3	Menu Trainer	11
4	Menu Maze	18
5	Prototype Menu Handler	23
6	Pretty Flashy	29
7	Window Trainer	35
8	Elevator	46
9	Pattern Maker	52
10	Address Book	58

11	Mouse Utilities	62
12	Quickdraw Trainer	70
13	Scribble and Video	91
14	Rose Patterns	98
15	Ups 'N Downs	112
16	Fancy MOUSE (0)	118
17	Help File Generator	124
18	The Help File Support	128
	Section 3 A Programmer's Notebook	132
19	Please Take Note . . .	132
20	Macintosh Programming Notes	134
21	Bugs	156
	Section 4 Applications	162
22	Cursor Editor	162
23	Mouse Kaleidoscope	180
24	Vocabulary Quiz	204
25	Dialog Designer	219
	Section 5 Keeping at It	247
26	Now It's Your Turn	247
27	Challenges Revisited	251
	Appendix A Command/Program Cross Reference	257
	Appendix B Program Highlights	262
	Appendix C Help File Listings	266
	Appendix D Feedback	274
	Index	275

Acknowledgments

I wish to thank Kevin Burton of TAB BOOKS Inc. for his help and guidance during this project. Thanks go to TAB also for giving me the extra time to take full advantage of MS-BASIC 2.0. Tom Steppe of FAME Software gave me the idea for the Pretty Flashy program. Stan Wyszomirski, also of FAME Software, kindly tried out some of my programs and offered valuable suggestions. Finally, to my wife and children, I owe more than I can say. Without their love, patience, and understanding, I never would have started, let alone finished this project.

Program Titles

Program 3-1	Menu Trainer
Program 4-1	Menu Maze
Program 5-1	Prototype Menu Handler
Program 6-1	Pretty Flashy
Program 7-1	Window Trainer
Program 8-1	Elevator
Program 9-1	Pattern Maker
Program 10-1	Address Book Entry
Program 11-1	Click to Continue Subroutine
Program 11-2	Crosshairs
Program 11-3	Flying Crosshairs
Program 11-4	Mouse Inside Rectangle Subroutine
Program 11-5	Drag Open a Rectangle
Program 12-1	Quickdraw Trainer
Program 13-1	Scribble or Random Neckties
Program 13-2	Video
Program 14-1	Rose Patterns
Program 15-1	Ups 'N Downs
Program 16-1	Fancy MOUSE (0)
Program 17-1	Help File Generator
Program 18-1	Help File Support
Program 20-1	Windows With One-Pixel Borders
Program 22-1	Cursor Editor

Program 23-1 Mouse Kaleidoscope
Program 24-1 Vocabulary Quiz
Program 25-1 Dialog Design (Startup Program)
Program 25-2 Dialog Boxes
Program 25-3 Generated Basic for Print Setup
Program 25-4 Generated Basic for Print . . .
Program 27-1 Changing Cursors
Program 27-2 Double Click Analyzer

Introduction

This book tells you how to write Interactive programs on the Macintosh using Microsoft BASIC 2.0. It assumes that you already know or are rapidly learning BASIC. The emphasis is on using the interactive features of the Macintosh: menus, graphics, buttons, edit fields, the mouse, and so on. It contains over 20 programs designed to illustrate these topics in as much depth as the thickness of the book will allow.

THE ORGANIZATION OF THE BOOK

The book is divided into five sections:

Section One:	Getting Started
Section Two:	Learning the Ropes
Section Three	A Programmer's Notebook
Section Four	Applications
Section Five	Keeping at It

Each section begins with a brief introduction explaining the contents and use of the section.

Section One discusses MS-BASIC on the Macintosh and how to use the book.

Section Two consists of 16 chapters containing sample programs that demonstrate features and techniques. The emphasis is on the interactive programming features. Each program concentrates on one specific area such as menus, edit fields, or buttons. Each chapter discusses significant points about the program it contains.

Section Three is a collection of short discussions of a wide variety of programming

topics. First there is a short introductory chapter. The next is a chapter of notes dealing with details of programming interactively. The third and last chapter of the section is a collection of bugs. Each section of the chapter discusses a specific bug. Most of these bugs infested the programs in the book during their development. By studying the bugs chapter, I hope you can avoid some grief. You may also discover how to debug some of your existing programs if you have already started writing programs with MS-BASIC 2.0.

Section Four presents four sample application programs partially developed. They are all usable, but none could be considered fully polished. They are complete enough to show you most of what you need to write your own applications. They are incomplete enough to allow you to change them to suit your own style of implementation.

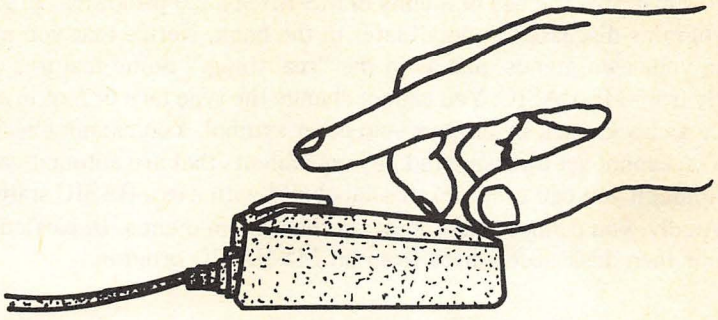
Section Five wraps up the book with suggestions about how to continue your application development career using MS-BASIC 2.0. I have included summaries and cross references of the points illustrated by the programs. This includes a command by command chart. If you want to know which programs use the DIALOG (1) function, simply consult the chart.

PROGRAM LISTINGS

Each program is listed in full in the chapter that discusses it. I have tried to debug these programs as much as possible. In a few cases, bugs have been left in as challenges. In all such cases, I have indicated how to debug the code.

The program listings have many lines that contain a bold, outlined plus sign, ✚ . This is used whenever a long BASIC statement had to be placed on two printed lines. If you enter the programs by hand, you should make sure all such lines are put back together again. Otherwise, the lines will probably cause a syntax error. I will repeat this warning in several places throughout the book, so please bear with me. I would rather bore you with the explanation than have your versions of the programs fail to operate.

Chapter 1



A Tour of MS-BASIC on the Mac

Welcome to the world of Macintosh programming! This book introduces you to MS-BASIC 2.0 on the Macintosh computer. This is a much different language than the microcomputer BASICs of the past. But then the Macintosh is much different from the microcomputers of the past. The key to these differences lies in the user interface.

I originally entitled this book *Interactive Programming* because I emphasize Macintosh user interface features. MS-BASIC 2.0 has done an admirable job of making the Macintosh features available to the "rest of us" as the phrase goes. From MSB 2.0 you can access much of the built-in support for interactive programming: menus, windows, buttons, edit fields, dialog, the mouse, and Quickdraw graphics. And what is more, you don't have to be an "Inside Macintosh" guru to make it all work! You simply use the packaged commands that MSB 2.0 provides and the rest is automatic. Well, almost automatic.

In this chapter, I walk you through some of these capabilities. Along the way, you should get an idea of the kinds of topics that I emphasize in the remainder of the book. If the material here doesn't pique your curiosity, doesn't get your creative juices flowing, doesn't make you want to sit down and start work on your own MS-BASIC applications, then this book is probably not for you. On the other hand, if any of the things I just mentioned does happen to you, then read on! The fun is just beginning.

MENUS

In MS-BASIC 2.0 you get a MENU command with several different variations and a MENU function. You can create your own menus and display them on the menu bar. With the MENU ON and ON MENU statements, you can detect when the user has made

a selection from one of your menus. The MENU function enables you to detect which menu and which item from that menu was selected. From there, you can route your program to subroutines that respond to the various items you have incorporated.

Figure 1-1 illustrates the use of menus in MS-BASIC 2.0 programs. It is taken from one of the programs discussed in detail later in the book. Notice that you may have up to 20 items in your own menus, just as in the "real thing." Some features are not supported directly from MS-BASIC. You cannot change the type face or font in a menu. You can only place a check mark by an item—no other symbol. You cannot check a dimmed menu item. You cannot set up command key equivalents that are automatically detected by BASIC (although you can program this capability with extra BASIC statements—see Chapter 4). Finally, you cannot install items in the system menus. In particular you cannot install your own desk accessories from an MS-BASIC program.

WINDOWS

MS-BASIC 2.0 provides you with up to four different windows, which may be of four different types: document, shadowed, one-line border, or framed. Figure 1-2 illustrates these. The WINDOW command allows you to create and open windows on the screen. The WINDOW CLOSE command allows you to close an existing window. The WINDOW function provides your program with valuable information it may need while manipulating its windows.

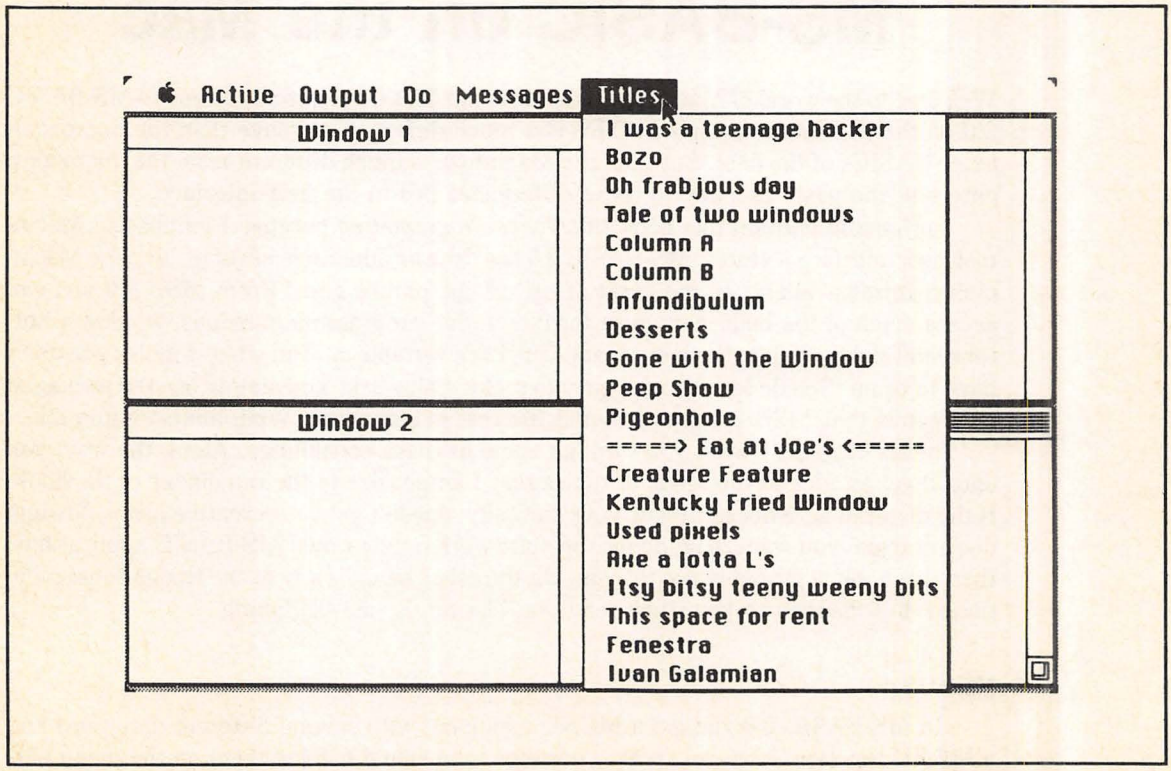


Fig. 1-1. An MS-BASIC menu with twenty items.

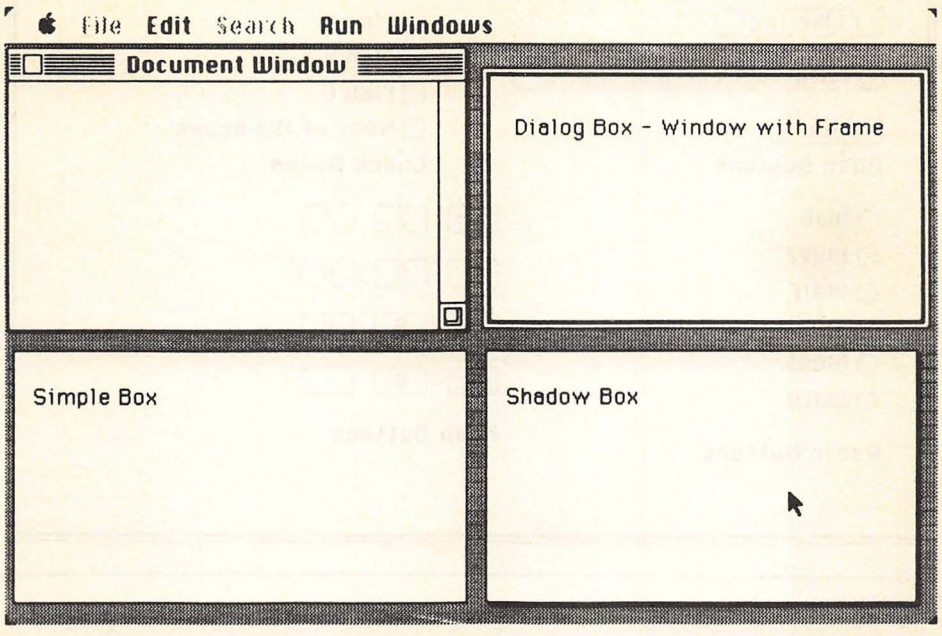


Fig. 1-2. The four types of MS-BASIC windows (clockwise from the upper left): document window, framed window or dialog box, shadowed window, window with a one-pixel border.

BUTTONS

Buttons are a most familiar part of the Macintosh landscape. With MSB 2.0, you can create push buttons, check boxes, and radio buttons. Then using the DIALOG function, you can detect when the user pushes one of them and make your program take the appropriate actions. You can design dialog boxes or use buttons in a number of creative ways in your programs. Figure 1-3 shows a window with lots of different buttons. It was created with the Dialog Design application discussed in Chapter Twenty-five.

EDIT FIELDS

Many of you may be interested in creating your own data processing type applications. The EDIT FIELD statement enables you to create fields on the screen into which a program user can enter data. The data can be freely edited using typical Macintosh editing capabilities. The EDIT$ function can be used to return the contents of an EDIT FIELD at any time. Your program can take any extra actions it needs to in order to reformat, validate, convert, or otherwise manipulate the data in the field. There are simple examples of the use of edit fields in several of the programs in the book.

Figure 1-4 shows an example of a simple form for entering names, addresses, and telephone numbers.

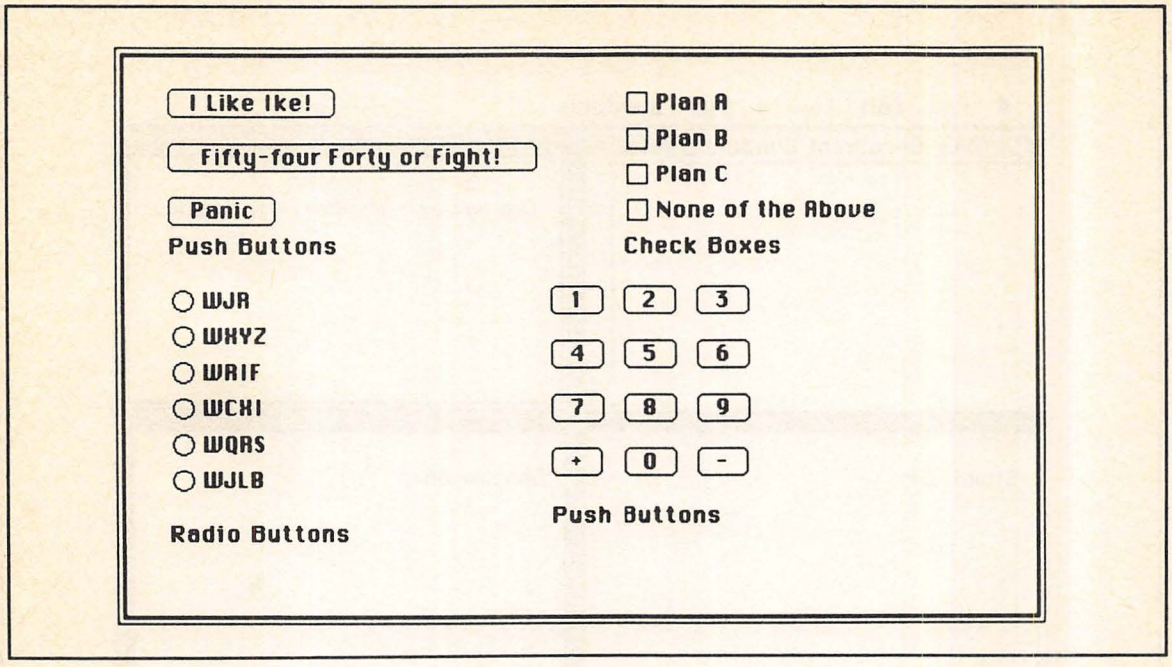


Fig. 1-3. Buttons galore.

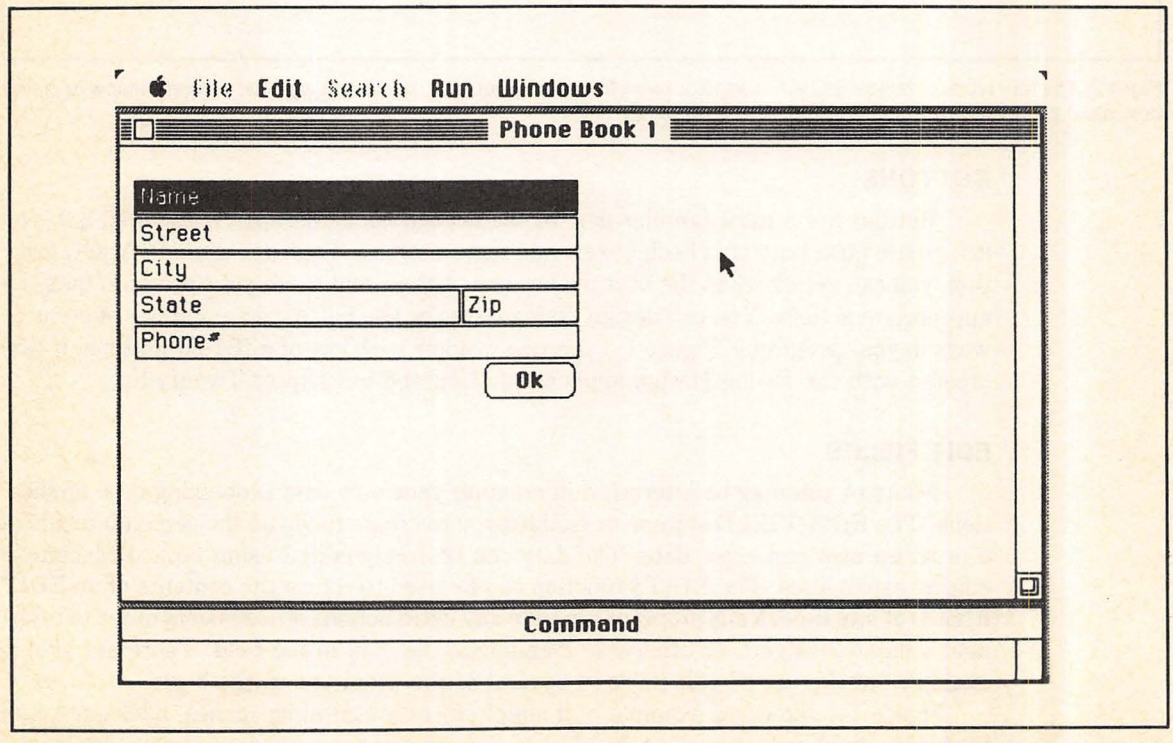


Fig. 1-4. A simple input form using edit fields.

DIALOG

The term *dialog* refers to most of the interaction between a user and a Macintosh program involving buttons, edit fields, and windows. MS-BASIC 2.0 provides a DIALOG function, which stores information about what *events* have occurred in your program. You can query this function at any time to find out the identity of the oldest event that your program has not yet responded to. You can also use the ON DIALOG statement. Its use will cause your program to be sent to a subroutine as soon as any event that the DIALOG function can recognize occurs. I discuss both ways of using the DIALOG function in the course of the sample programs and applications.

THE MOUSE

Ah, the mouse, that "wee, sleekit, cow'rin, tim'rous beastie." The mouse is indeed the most visible, most obvious, most publicized feature that sets the Macintosh apart from other microcomputers. You either love it or you hate it. There seems to be no indifference to the mouse.

What would programming the Macintosh be like if you couldn't access the mouse? One shudders to contemplate. Fortunately, MS-BASIC comes through when it comes to the mouse. It provides the MOUSE function with everything you ever wanted to know about the mouse. It will tell you where the mouse cursor is at all times and what the button has been doing since you last checked. The ON MOUSE, MOUSE ON, and MOUSE OFF statements enable you to GOSUB to a specific subroutine as soon as the mouse is clicked. These statements enable you to detect mouse clicking without having to check for it in every other line of your program.

Even with the MOUSE function and the ON MOUSE statement, there is a lot of trickery involved in programming with the mouse. Examples of mouse use pervade the programs I have included. If you study them all, you will have built a firm foundation in mouse techniques. You can then continue your career by inventing your own.

MS-BASIC GRAPHICS AND QUICKDRAW GRAPHICS

If the mouse is the heart and soul of the Macintosh, then graphics must be its beautiful face and body. MS-BASIC 2.0 provides both built-in BASIC commands and interfaces to the Macintosh Quickdraw ROM routines.

The graphic features in MS-BASIC 2.0 are almost too extensive to survey. BASIC commands are provided to draw lines and circles, scroll areas of the screen in various directions, and GET and PUT areas of the screen. The GET and PUT commands use BASIC arrays to store the screen information in a form known as a *bit-map*. With GET and PUT you can do amazing things. The Quickdraw interface is documented in the MS-BASIC Interpreter Manual. You should read that carefully as an introduction to those capabilities.

I have made extensive use of graphics in the programs included herein. Output from these programs is generously illustrated in the figures that accompany those programs.

IMPROVED STRUCTURE AND STYLE

MS-BASIC 2.0 has a totally different look to it than most microcomputer BASIC implementations. The most startling difference is the absence of line numbers. Not that they

cases where you need a line number as the destination of a GOTO or GOSUB, MS-BASIC 2.0 allows you to use a *label* instead. A label is a simple BASIC identifier or name. How much nicer it is to read:

```
GOSUB initialize
GOSUB get.program.data
GOSUB execute
```

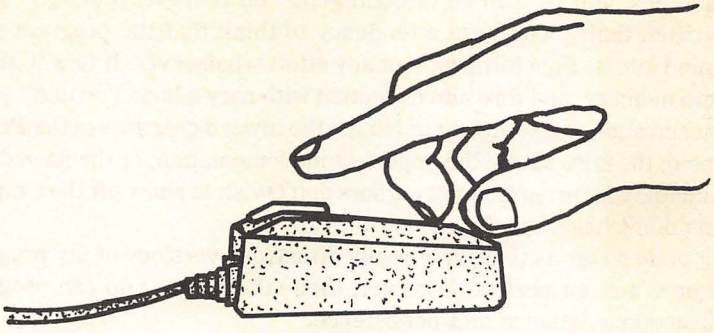
than

```
GOSUB 1000
GOSUB 2000
GOSUB 3000
```

In addition to the improvement in the readability of programs introduced by this feature, there is a profound difference in the program development process. No longer do you have to worry whether leaving 10 lines of space between adjacent BASIC statements will be "enough." No longer do you have to worry about RENUMBERing. No longer do you have to put up with one dimensional, ugly looking program listings. Welcome to the world of structured code. If you insist upon it, you can stay stuck in your old rut of line numbers and GOTOs, but there really is a better way and you should try it. Almost all the programs included in this volume have no line numbers at all. When you contrast the one or two that do with the rest, I feel confident you will want to give up your line numbering ways as well.

The freedom from line numbers allows a much nicer program style as well. It encourages the use of indentation to reveal program structure. This results in much cleaner looking listings. The increased readability makes such programs easier to study, understand, and learn from. Spend time with the programs and see it you don't agree.

Chapter 2



Programs, Programs, Programs

I have always felt that a book about programming should contain programs. This one does. There are fifteen sample programs and four applications. They are all programs that I wrote, some for my own use and some specifically for this book. My only regret is that there are not more programs and longer programs. But the limitations of a few hundred page book dictate otherwise.

I am a programmer and I will assume that you are a programmer or wish to be a programmer. I won't write your programs for you, I will make an effort to show you how to write them. Even that is a tall order! Programming is an open-ended topic. There is always something new to learn about it. I can't tell you everything there is to know about programming. I can't even tell you everything there is to know about interactive programming in MS-BASIC on the Macintosh.

SOME CAUTIONS

I have tried to be as broad in my coverage as I could. If some topics seem to have more emphasis than others, I apologize. Some of you would like to see 25 programs that deal with data entry and processing. Others would like to see 25 game programs or 25 utility programs and so on. Rather than concentrate on only one type of program, I have tried to include something for everyone.

The programs vary in their state of completion. I personally feel that no program is ever complete. To a programmer, the ideal program is like the perfect wave to a surfer. It exists in the mind only, and while we never stop looking for it, it is never found. Most of the programs here work in one way or another. You can run them and they do something

entertaining or useful. On the other hand, many of them could stand cleaning up, extending, modifying, and so on.

I think that when you pick up a book with a fancy program that is all spit and polish and no rough edges, you tend to be discouraged. You think to yourself, "Boy, I could never have written that." You have a tendency to think that the program sprang from the author's mind into its final form without any effort whatsoever. It flew to the keyboard, threw itself into memory, and tore into execution with nary a bug. You don't see the hours of intense concentration, the wringing of hands, the myriad choruses of the Programmer's National Anthem, the false starts, the imperfect implementation, or the flawed statements. It is understandable that programming authors don't wish to show off their bugs in public. Nice programs don't have bugs!

I have not gone so far as to include really disastrous versions of my programs. If the ones you see here are not perfect, however, then take heart. You can program as well as I can—it just takes patience and persistence.

128K Versus 512K

Unfortunately, I have a 512K Macintosh. I say it is unfortunate, since I have not been able to guarantee that all the programs here will run on a 128K machine. In fact, I can guarantee that at least one or two of them will not run on a 128K machine. To ameliorate this situation a bit, I include a brief discussion in the chapter concerning how to cut down fat programs to fit on a skinny Macintosh. I would heartily recommend that you get the 512K upgrade, unless your interest in programming is desultory at best. With 512K you just don't have to worry about running out of memory.

The remainder of this chapter is devoted to a discussion of how to get the most out of this book.

USE THE PROGRAMS

By using the programs I have included, you will accomplish two things:

1. You will see MS-BASIC in action. This will help you become familiar with the interactive features that the language supports.
2. You will think of other things to try out. When a particular program doesn't do exactly what you want it to, then you will have a ready made challenge.

READ THE PROGRAMS

Reading programs is one of the better ways to increase your programming skills. As a programmer you should think of yourself as a writer. Writers that never read would probably not write anything of value. Programmers who don't read other people's programs stand in the same position in my opinion. Of course there are exceptions to every rule!

As you read programs, your stock of programming techniques increases. Your ability to understand how programs work increases. You will be quicker to grasp the structure of a program, and you will be better equipped to isolate problems in your own programs.

Reading programs can actually be fun as well. It is immensely satisfying to read a long and complicated piece of code and figure out what it is doing and how.

TEST THE PROGRAMS

As you use the programs in the book, try to bend and stretch them wherever possible. Only the simplest programs are entirely bug free. Even programs that have been thoroughly tested have obscure bugs lurking within them. By trying to surface bugs in my programs, you will be training yourself to debug your own.

I have deliberately left certain problems in some of the programs of the book. I will mention them as the corresponding chapters come along. You should always try to find them first, however. There may also be bugs that I don't know about. Feel free to write and tell me about them. Of course, there is the old problem that one person's bugs are another person's features!

MODIFY THE PROGRAMS

Writing and reading programs are not the only ways to improve your skills as a programmer. You can also modify programs. If you aspire to being a professional programmer, you should be aware that a large percentage of all the programming done in the world takes the form of *maintenance* of existing programs. That word has a bit of a bad name among programmers. It includes debugging existing programs. It also includes changing the way a program works. This might involve making the program more efficient or making the user interface behave differently.

You should try modifying the programs here. If you don't like the way a program behaves, change it. If you don't like the wording of a menu item, change it. If you don't like the way a particular choice is handled, change it. If you are perfectly satisfied with everything, change it anyway. Then by definition you won't be satisfied any longer and you can change it back!

EXTEND THE PROGRAMS

Extending a program is similar to modifying it. You are still working with an existing program rather than writing one from scratch. The difference is that extending means adding features, whereas modifying means making something that is already there work differently.

If you never write a program from scratch, there are enough possible extensions to the programs here that you could spend a good long time just doing them and nothing else. I will be making suggestions for modifications and extensions to the programs as I discuss them. Try out some of these suggestions before you start writing your own applications.

FIX THE PROGRAMS

As I hinted earlier, many of the programs can stand additional work. There are a few cases of known bugs that I either have not solved or have not had the time to remove. This is your chance. Put on your debugging cap and have at it. I will mention the problems that I know about in the discussions of the programs.

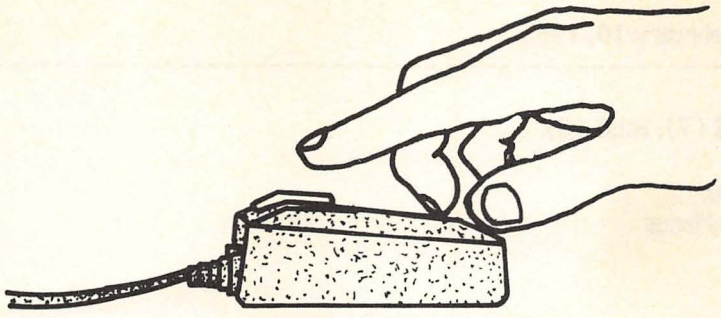
READ THE NOTEBOOK

Section Three of the book is a collection of short notes about programming in MS-

BASIC. Many of them deal with fine points of usage that I spent considerable time digging out while developing the code for this book. Others deal with bugs that can occur and tell you how to understand, recognize, and avoid them. Still others present observations about programming in general. For a good example of this last category, see the entry entitled "The Programmers' National Anthem."

Section Three could easily have been two or three times as large as it is. Browse in it. Scratch your own comments and reactions in the margins. Use the hints and avoid the bugs. Above all, enjoy.

Chapter 3



Menu Trainer

I wrote the menu trainer program to demonstrate the actions that may be taken on a menu:

- Activate an item or entire menu
- Dim (deactivate) an item or entire menu
- Check and activate an item

Note that you can't check the title of a menu in the menu bar unless you build the title with the check mark in it directly: `title$ = CHR$(checkmark) + " Menu"` and then use the statement `MENU 1, 0, 1, title$`.

The program is not of any practical use, but by reading the program and the sections of this chapter following it, you will learn the basics of using menus in your programs. Read this chapter after you have studied the MENU and MENU ON/MENU OFF/MENU STOP entries in the MS-BASIC Interpreter manual.

A gold star goes to everyone, who unlike me, remembers all seven of the seven dwarfs! Was it Happy, Hopeful, Silly, . . . ?

Program Listing 3-1
Menu Trainer

```
'-----------------------------------------------------------------
' Menu Trainer
'
' Written by
```

```
' Richard C. Vile, Jr.
' January 17, 1985
'
' Revised - February 10, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM dwarfs$ (7), acts$ (5)

GOSUB Setup
GOSUB MakeMenus


WHILE -1
WEND
END

Handler:
  MENU STOP

  menunumber = MENU (0)
  menuitem   = MENU (1)

  ON menunumber GOSUB dwarfs, actions
  MENU ON
  MENU
  RETURN

dwarfs:
  MENU 1, Which.item, 1, dwarfs$ (Which.item)
  Which.item = menuitem
  MENU 1, Which.item, 1, "***** " + dwarfs$ (Which.item) + " *****"
  RETURN

actions:
  ON menuitem GOSUB check, uncheck, dim.it, activate, quit
  RETURN

check:
  MENU 1, Which.item, 2
  RETURN

uncheck:
  MENU 1, Which.item, 1
  RETURN

dim.it:
  MENU 1, Which.item, 0
```

```
  RETURN

activate:
  MENU 1, Which.item, 1
  RETURN

quit:
  MENU RESET
  END

MakeMenus:
  MENU 1,0,1, "Dwarfs"
  Which.item = 1

  MENU 1,1,1, "***** " + dwarfs$(1) + " *****"
  MENU 1,2,1, dwarfs$(2)
  MENU 1,3,1, dwarfs$(3)
  MENU 1,4,1, dwarfs$(4)
  MENU 1,5,1, dwarfs$(5)
  MENU 1,6,1, dwarfs$(6)
  MENU 1,7,1, dwarfs$(7)

  MENU 2,0,1, "Actions"
  MENU 2,1,1, acts$(1)
  MENU 2,2,1, acts$(2)
  MENU 2,3,1, acts$(3)
  MENU 2,4,1, acts$(4)
  MENU 2,5,1, acts$(5)

  MENU ON
  ON MENU GOSUB Handler

  RETURN


Setup:
  FOR i = 1 TO 7
    READ dwarfs$(i)
  NEXT i
  FOR i = 1 TO 5
    READ acts$(i)
  NEXT i
  RETURN

'-----------------------------------------------------------------
'  Who is that seventh dwarf anyway????
'-----------------------------------------------------------------
```

```
DATA Dopey, Sneezy, Doc, Sleepy, Grumpy, Bashful
DATA "Happy, Silly, Hopeful, Kalamazoo?"
DATA Check, Uncheck, Dim, Activate, Quit
```

DETAILS OF THE PROGRAM

The menu trainer program creates two menus called **Dwarfs** and **Actions**. The idea is for you to choose a Dwarf and then act on how that name is displayed in the menu with various **Actions**. The **Actions** are precisely those that you can take with respect to the appearance of the menus in your programs:

Check
Uncheck
Dim
Activate

A fifth action—**Quit**—is provided to get you out of the program when you have had enough.

Menu Trainer highlights the item currently selected in the **Dwarfs** menu. Note that simply checking that item would be inadequate, since **Check** and **Uncheck** are the actions being illustrated. Rather, the choice is surrounded by asterisks. In order to accomplish this, the program reads the title strings into a string array, dwarfs$. Each entry in the string array dwarfs$ is the title of one of the items in the **Dwarfs** menu.

The program statement:

```
MENU 1, Which.item, 1, "***** " + dwarfs$ (Which.item) + " *****"
```

provides the needed "highlighting." It also illustrates the very important fact that the title string part of the MENU command accepts a string expression. This is true of the other parts of the MENU command as well and can prove to be very useful at times as you will see in the upcoming programs. Figure 3-1 shows the **Dwarfs** menu with the highlighted choice. After the choice indicated in the figure is made, the highlighted dwarf will be **Doc**.

HANDLING MENUS

You may have wondered how a BASIC program can be written to handle choices from menus. The user can pull down a menu any old time he or she feels like it. Up to 200 different choices (10 menus times 20 choices per menu) are potentially possible at any given moment.

The answer lies in the concept of *event handling*. This is another fundamental Macintosh concept and one to which I shall return often in the course of this book. In the case of menus, the event that must be handled is the user's choice of a menu item. The way that the event is *detected* is by a mechanism called *trapping*. When the event occurs, the BASIC program traps it by immediately transferring control to a section of the program that is designed to respond to or *handle* the event. In order for your BASIC program to do trapping, it must first be *enabled*. This is a two step process. First you must tell BASIC that you intend to trap menu events, and then you must tell it where to transfer control when a menu event occurs. This is accomplished by the pair of statements:

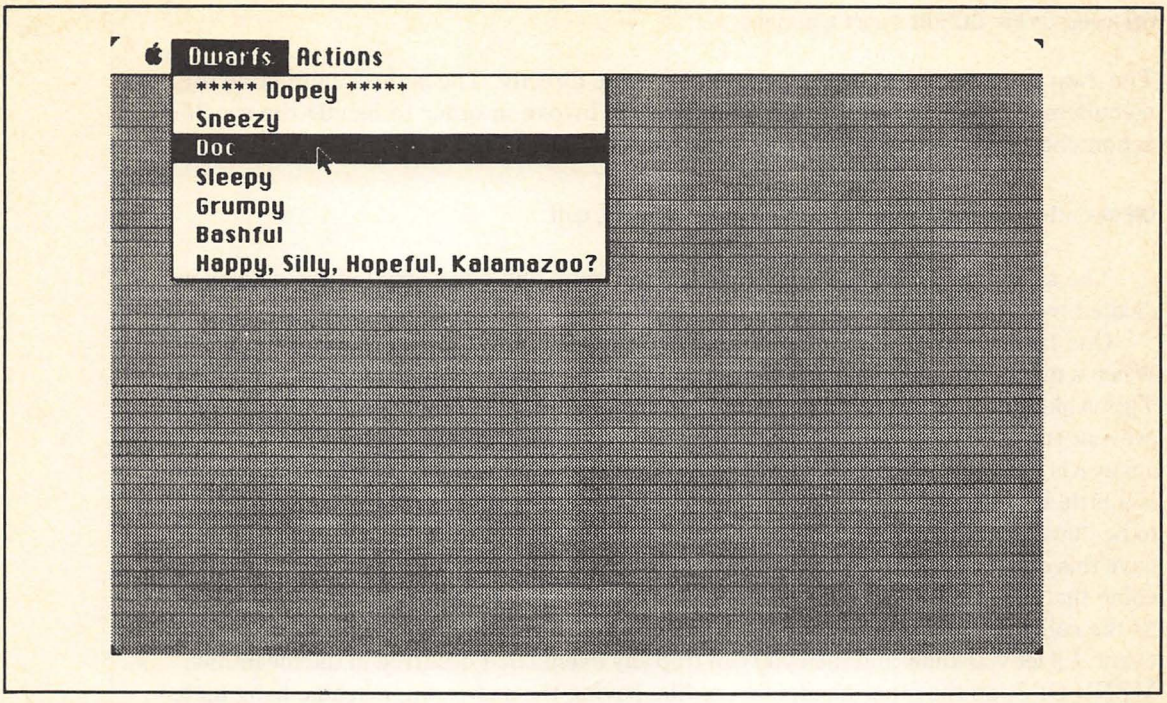


Fig. 3-1. Pick a dwarf, any dwarf!

```
MENU ON
ON MENU GOSUB handle.it
```

The first says to BASIC, "You may now trap menu events." The second says to BASIC, "When a menu event occurs, perform a GOSUB to the subroutine named "handle it." Of course, the name of the subroutine may be anything you choose. The GOSUB is executed from wherever the BASIC program happens to be at the time.

In a simple application, the program sets up menus and then proceeds to *wait* for a menu event to occur while doing nothing. This is called an *idle-loop* and is illustrated in both the Menu Trainer and the Menu Maze programs. This is not always the way a program will be structured, but it is very common nonetheless.

When the BASIC program arrives at the menu handling subroutine, it is necessary for it to find out which item from which menu was chosen. This is also accomplished by two BASIC statements, which use the MENU function:

```
menunumber = MENU (0)
menuitem   = MENU (1)
```

These statements perform the obvious tasks. The use that is made of the information thus gathered is illustrated by the menu programs. The choices from each menu are handled in separate parts of the program. So the first thing that happens is that menunumber is used to decide which part of the program to invoke next. This is done with an ON GOSUB statement:

```
ON menunumber GOSUB dwarfs, actions
```

The dwarfs subroutine processes the choice made directly. The actions subroutine uses menuitem to decide which further subroutine to invoke in order to handle the specific action chosen. This is again accomplished with an ON GOSUB statement:

```
ON menuitem GOSUB check, uncheck, dim.it, activate, quit
```

The subroutines check, uncheck, dim.it, activate, and quit carry out the actions indicated by their names.

One final point about the fundamentals of menu event trapping needs to be made. When a menu event occurs, it should be handled before the next menu event is trapped. This is like finishing your shower before answering the phone. Since you can't physically prevent the user from making another menu choice, how do you accomplish this? The answer is that you tell BASIC not to *perform* any more traps until further notice. This is a little like turning on the answering machine on your phone—you are telling BASIC to be "unavailable at the present time." The statements MENU OFF and MENU STOP have this effect. The difference between them is that MENU STOP is like the phone machine that takes the caller's name and phone number and promises to call back as soon as the callee is free. In effect it says, "as soon as I finish processing the current menu event, I'll let you know and then you can trap any events that occurred in the meantime." MENU OFF, on the other hand, is more like having the answering machine hang up as soon as its recorded message is played—or even better, not answering at all. In order to then turn menu trapping back on again, simply say MENU ON. This is usually done just before returning from the menu handling subroutine.

The final final point about menu event trapping is this: when the user selects a menu item, the title of the menu it belonged to is inverted or highlighted in the menu bar. This highlighting is a visual indication that some choice from that menu is being processed. It is incumbent upon your program to clear that highlighting when the item selected has been processed. This is accomplished by the simple, unadorned MENU command. It is a good idea also to do this just before returning from the menu event handling subroutine:

```
MENU ON
MENU
RETURN
```

A CLEAN MENU BAR IS A HEALTHY MENU BAR

After I wrote this program and a few others involving menus, I realized that something had been subconsciously nagging me. I finally figured out what it was. When I used my own menus in a BASIC program, they got mixed up with the MS-BASIC default menu bar menus. Since I picked up the habit of numbering my menus starting at 1, my menus would replace the BASIC `File Edit Search Run Windows` menus in that order. Since I rarely used as many as five menus at once in my simple beginner's programs, there would be some of my menus and some of BASIC's menus on the menu bar at the same time. This was somewhat ugly or messy or annoying—whatever you want to call it. So I decided that whenever I used one or more menus in my BASIC programs, I would first "deep

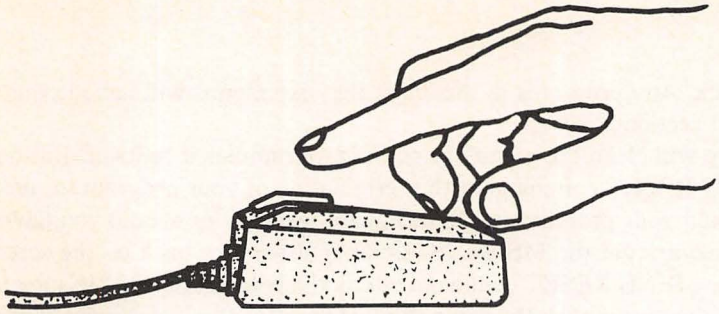
six'' the MS-BASIC menus. Fortunately, this turns out to be easy to do:

```
FOR mn = 1 TO 5
  MENU mn, 0, 0, ""
NEXT mn
```

does the trick. Any programs in this book that use menus will include this loop in their initialization sections.

The loop will clean the menu bar so the program has it to itself. But don't forget to put a MENU RESET command at the termination of your program to restore BASIC's menus. Should your program fail to terminate properly or should you have to interrupt it, you will always get the MS-BASIC Command window back on the screen. You may then type the MENU RESET command interactively to get the BASIC menus back again. Another way to accomplish the same thing is to enter the List window and start editing your program. The first time you press the Return key in the List window, BASIC will restore its own menu bar.

Chapter 4



Menu Maze

I dreamed up this program in a silly moment one day. It seemed like an intriguing idea at the time. Now? Well . . . It does demonstrate some further points about MS-BASIC menus.

The idea of the program is to represent a maze entirely with menus. I did it with one menu whose title keeps changing. The menu title represents the user's current location in the maze. The menu items represent the possible directions. You find your way through the maze by selecting directions from the menu until you get to the finishing location. The program then congratulates you and quits.

The program has been modified from its original form to allow Command-Key equivalents for the directions represented in the menu:

⌘N ===> North

⌘S ===> South

⌘E ===> East

⌘W ===> West

The MENU command does not allow you to do this directly. It takes some trickery on the part of the programmer. I explain this in detail following the listing.

THE PROGRAM LISTING

This program listing contains the symbol ✣. It is used to indicate BASIC statements

that in the listing occupy two lines, but when typed into an MS-BASIC program should occupy only a single line. The reason for this notation is that certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ↵. An an example; if the listing shows

```
IF XO < X THEN    ↵
    inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN  inside = 0 : RETURN
```

Program Listing 4-1
Menu Maze

```
'-----------------------------------------------------------------
' Menu Maze
' Demonstration of MS-Basic MENU command
'
' Written by Richard C. Vile, Jr.
' December 27, 1984
' Revised February 10, 1985
'------ ----------------------------------------------------------
numrooms = 14
DIM rmnames$(numrooms)
DIM map (numrooms,4)

GOSUB Setup
GOSUB Instructions

MENU ON
ON MENU GOSUB Travel

location% = 13
endlocation% = 14
GOSUB NewLoc

WHILE -1
  key$ = INKEY$
  IF key$ <> "" THEN GOSUB MapKey
WEND

Travel:
  MENU STOP
  menunumber = MENU (0)
```

```
  menuitem    = MENU (1)
  GOSUB ChgLoc
  MENU ON
  MENU
  RETURN

ChgLoc:
  nowlocation% = location%
  location% = map (nowlocation%, menuitem)
  IF location% = endlocation% THEN GOSUB finish
  IF location% = 0 THEN PRINT CHR$(7);"Can't go that way" :  +
    location% = nowlocation% : RETURN
  GOSUB NewLoc
  PRINT "You are now in ";rmnames$(location%)
  RETURN

NewLoc:
  MENU 1, 0, 1, rmnames$(location%)
  MENU 1, 1, (1 + (map (location%,1) = 0)), north$
  MENU 1, 2, (1 + (map (location%,2) = 0)), south$
  MENU 1, 3, (1 + (map (location%,3) = 0)), east$
  MENU 1, 4, (1 + (map (location%, 4) = 0)), west$
  MENU ON
  RETURN

MapKey:
  withkey = -1
  menuitem = 0
  IF key$ = CHR$(14) THEN menuitem = 1
  IF key$ = CHR$(19) THEN menuitem = 2
  IF key$ = CHR$(5) THEN menuitem = 3
  IF key$ = CHR$(23) THEN menuitem = 4
  IF menuitem <> 0 THEN MENU OFF : GOSUB ChgLoc : MENU ON
  withkey = 0
  RETURN

Setup:
 FOR i=1 TO numrooms
  READ rmnames$(i)
 NEXT i

DATA Room 1, Room 2, Room 3, Room 4, Room 5, Room 6
DATA Room 7, Room 8, Room 9, Room 10, Room 11, Room 12
DATA Start, End


 FOR i = 1 TO numrooms
```

```
  FOR j = 1 TO 4
   READ map(i,j)
  NEXT j
 NEXT i

DATA 0,2,6,0
DATA 0,3,8,0
DATA 0,4,8,10
DATA 0,5,9,0
DATA 4,12,0,0
DATA 0,0,7,0
DATA 0,0,13,0
DATA 6,4,9,0
DATA 7,5,13,0
DATA 2,4,3,11
DATA 1,5,10,14
DATA 0,13,0,0
DATA 0,1,0,0
DATA 0,0,0,0

north$ = "North   " + CHR$(17) + "N"
south$ = "South   " + CHR$(17) + "S"
east$ =  "East    " + CHR$(17) + "E"
west$ =  "West    " + CHR$(17) + "W"

FOR mn = 2 TO 5
  MENU mn,0,0,""
NEXT mn

RETURN
```

Instructions:

```
PRINT "You are at the start of a maze."
PRINT "Your goal is to find the location called 'END'"
PRINT "You move by selecting a direction from the menu"
PRINT "representing your current location. You should see
PRINT "a menu called 'START' now. Pull it down and you"
PRINT "will see the possible directions. You move by"
PRINT "selecting a direction from the menu. Good Luck!"

RETURN
```

Finish:

```
 PRINT "Congratulations! You got through!!"
 ms = MOUSE(0)
 x = MOUSE(1)
```

```
y = MOUSE (2)
WHILE ((MOUSE (1) = x) AND (MOUSE (2) = y))
  ms = MOUSE (0)
WEND
END
```

COMMAND KEY EQUIVALENTS TO MENU ITEMS

The MENU and MENU ON commands do not provide a direct way to implement so-called command-key equivalents to menu item choices. Some examples of command-key equivalents would be the use of ⌘H to perform a a cut operation or ⌘U to perform a paste operation. In other Macintosh applications, the use of command-key combinations is treated as being equivalent to making the corresponding menu selections with the mouse. Even though MS-BASIC provides no automatic way to trap command-key equivalents, it is possible to simulate such action. This is only convenient if your main program consists solely of an idle loop set up to trap menu selections (later on, this will be explained more generally as applicable to any dialog detectable by MS-BASIC's DIALOG function).

First, you must put the command-key symbol, ⌘, into the appropriate menu items along with the particular character you wish to make command-key equivalent. The printed representation of the command key is referred to by some people as *daisy*, since it looks somewhat like a flower. I prefer to call it *off-ramp*, since the symbol also somewhat resembles a four-way freeway interchange. Whatever you call it, its internal representation in the ASCII code is the value 17. This value may be placed into a string variable by enlisting the aid of the CHR$ function, CHR$ (17). Thus, the statement:

```
north$ = "North " + CHR$ (17) + "N"
```

sets up the string North ⌘N to be used in the MENU command. Let me emphasize that this in no way makes the detection ⌘N automatic. It simply notifies the user of the program's intention to treat ⌘N as equivalent to the selection of North from the menu.

While the program is sitting in its idle loop, it repeatedly checks to see if the user has typed a key. The BASIC function INKEY$ contains the character typed, if any, or an empty string, if no character has been typed. This means that the statements:

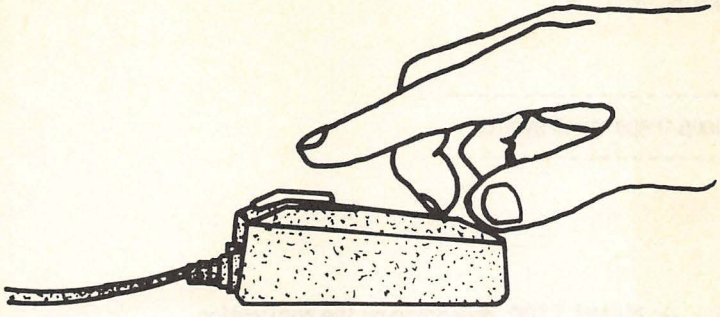
```
key$ = INKEY$
IF key$ <> "" THEN GOSUB MapKey
```

will route the program to the subroutine MapKey whenever the user types any key. The subroutine takes appropriate action if the key typed is a command-key equivalent.

The assignment key$ = INKEY$ is important. Trying to use INKEY$ both in the idle loop and in the subroutine MapKey won't work. The reason is that INKEY$ throws away its value as soon as it is called. It will then return the empty string again until another (possibly different) character is typed.

The subroutine MapKey looks for the characters ⌘N, ⌘S, ⌘E, and ⌘W by using the CHR$ function. The ASCII values of these four characters are 14, 19, 5, and 23, respectively. When MapKey sees any of those four values it sets the variable menuitem to the appropriate equivalent and calls the subroutine ChgLoc. ChgLoc is unaware that it was not reached because a menu selection was made. We brilliant humans have once again out-foxed the stupid computer!

Chapter 5



Prototype Menu Handler

This program is an example of a skeleton program; that is, it is a formula that you can reuse in many of your programs that use MS-BASIC menus. It is the layout of the program that is important—not the specific names used. In fact, I would expect you to change almost all the names in each new program you create. You should choose names in your programs that are suggested of what the program is trying to accomplish. The names I have chosen in the skeleton program suggest the role played by the various pieces of code in the overall structure of the program.

The typical menu-driven program has a *layered* structure to its subroutines that is illustrated by the Prototype Menu Handler. Compare the static organization of the program with its dynamic structure illustrated in Fig. 5-1. You may decide to change the order in which you write the subroutines in your own programs. This is discussed a little bit after the listing.

Program Listing 5-1
Prototype Menu Handler

```
'-----------------------------------------------------------------
' Prototype Menu Handler
'
' Written by
' Richard C. Vile, Jr.
' February 10, 1985
'-----------------------------------------------------------------
```

```
GOSUB makemenus

MENU ON
ON MENU GOSUB handle.em

WHILE -1
  '-------------------------
  ' This idle loop traps menu events
  '-------------------------
WEND

handle.em:

  MENU OFF : ' Or MENU STOP depending on the application

  menunumber = MENU (0)
  menuitem   = MENU (1)

  '-----------------------------------------------------
  ' The following ON GOSUB statement has as many
  ' destination labels as there are different menus
  '-----------------------------------------------------
  ON menunumber GOSUB m1.hand, m2.hand, m3.hand, m4.hand, m5.hand

  MENU ON
  MENU
  RETURN

m1.hand:

  '-----------------------------------
  ' Any special purpose preamble code needed by
  ' more than one item handler for this menu
  '-----------------------------------

  '-------------------------------------------------
  ' The following ON GOSUB statement has as many
  ' destination labels as there are different items
  ' in menu number 1 - there are 5 for illustration
  '-------------------------------------------------

  ON menuitem GOSUB item1.1, item1.2, item1.3, item1.4, item1.5

  '-----------------------------------
  ' Any special purpose postprocessing code needed
  ' by more than one item handler for this menu
  '-----------------------------------
```

```
  RETURN

  item1.1:
    '-----------------------------------
    ' Basic code to handle item 1 of menu 1
    '-----------------------------------
    RETURN

  item1.2:
    '-----------------------------------
    ' Basic code to handle item 2 of menu 1
    '-----------------------------------
    RETURN

    '-------------------------------
    ' Further subroutines as needed to
    ' handle all the items of menu 1
    '-------------------------------

m2.hand:
  '-----------------------------------
  ' Follow same pattern as menu1.handler
  '-----------------------------------

  RETURN


  '-------------------------------
  ' E T C. for the other menu handlers
  '-------------------------------

'-----------------------------------------------------------------------
'                        •
'                        •
'                        •
' Other parts of the Basic program - general purpose routines,
' menu item handler support subroutines, etc.
'                        •
'                        •
'                        •
'-----------------------------------------------------------------------

makemenus:

'-----------------------------------------------------------------------
' Setup the menus used by the program
'-----------------------------------------------------------------------
```

```
'-------------------------------
' First, "hide" the MS-Basic menus
'-------------------------------

  FOR mn = 1 TO 5
    MENU mn, 0, 0, ""
  NEXT mn

  '-------------------------------------
  ' Define mnemonic for first menu
  ' (in practice this will be something like
  ' "filemenu" or "optionmenu" or
  ' "actionsmenu", etc.
  '-------------------------------------
  firstmenu = 1
  '---------------------------
  ' Set up the menu and its items
  '---------------------------
  MENU firstmenu, 0, 1, "First"
  MENU  firstmenu, 1, 1, "item 1.1"

'-------------------------------------
' Other choices - default may be indicated
' by using the value 2 for its itemstate
'-------------------------------------

n1 = 9
MENU firstmenu, n1, 1, "item1.n1"


'---------------------------------
' Set up second and subsequent menus
' in a way analogous to the first
'---------------------------------
secondmenu = 2
MENU secondmenu, 0, 1, "Second"
MENU secondmenu, 1, 1, "item 2.1"

RETURN
```

ARRANGING YOUR SUBROUTINES

The order in which you write your subroutines is a matter of personal preference. The Prototype Menu Handler is organized in the following hierarchy:

Main Program: Idle loop to catch menu events
Menu Handler: GOSUB's to handlers for individual menus
    Menu 1 Handler

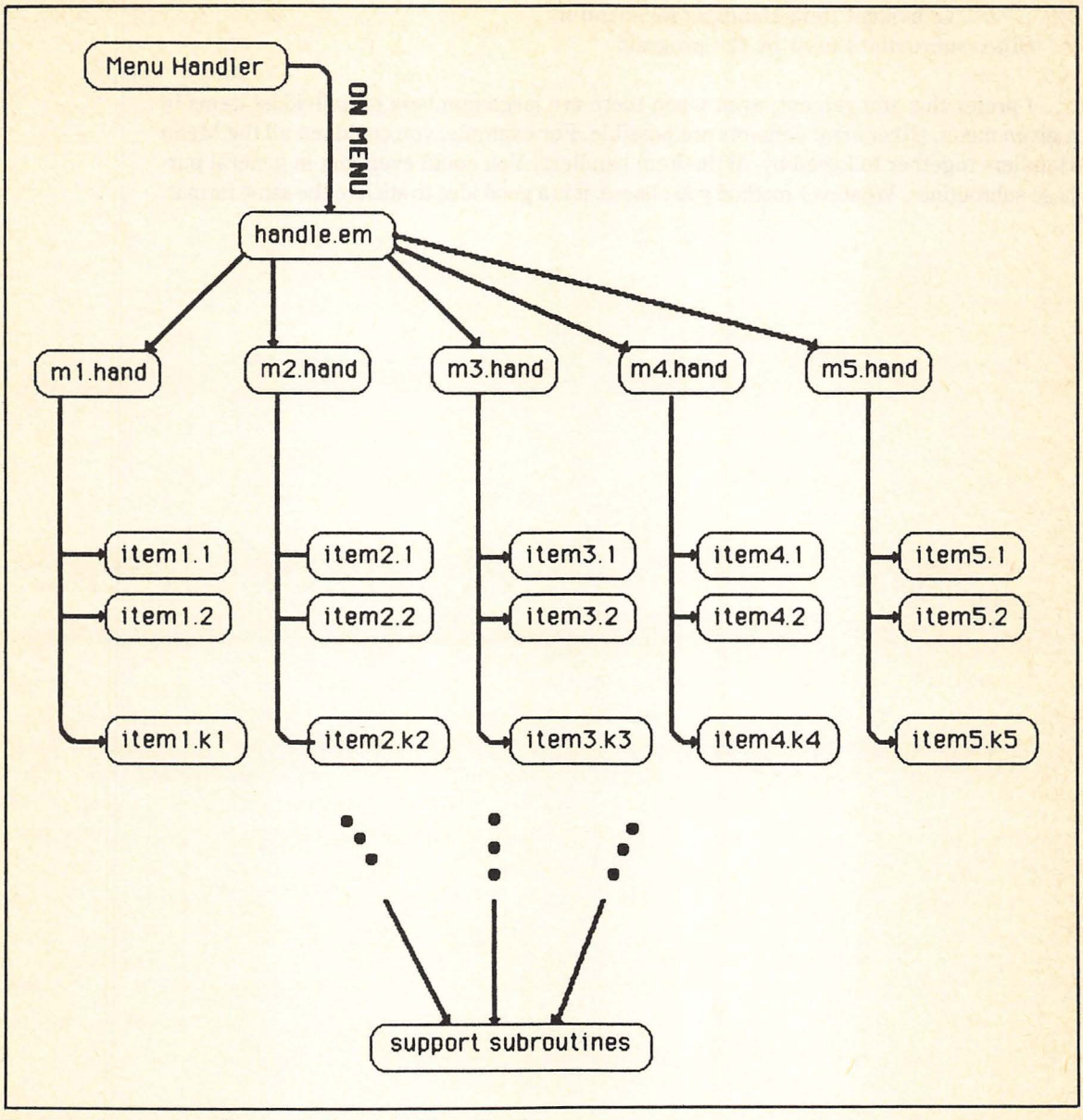


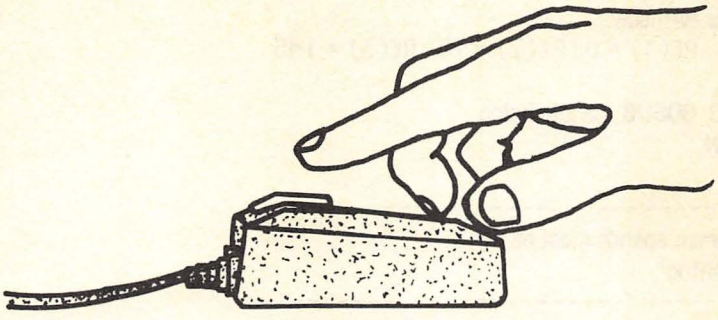
Fig. 5-1. The structure diagram for Prototype Menu Handler.

Individual Item Handlers for menu 1
Menu 2 Handler
Individual Item Handlers for menu 2
•
•
•
Menu n (n < = 10) Handler

Individual Item Handlers for menu n
Other subroutines used by the program

I prefer this arangement, even when there are large numbers of individual items in a given menu. Other arrangements are possible. For example, you could put all the Menu Handlers together followed by all the item handlers. You could even mix in general purpose subroutines. Whatever method you choose, it is a good idea to stick to the same format.

Chapter 6



Pretty Flashy

This program does a little bit of everything. It has a menu, it has multiple windows, and it has an edit field in which a little dialog takes place. It even has a button.

Pretty Flashy demonstrates the handling of multiple windows on the screen. When a given window is clicked, it begins to flash. The window that was flashing before clears itself and sits quietly until the next time it gets clicked. This may seem like an easy program to write, until you sit down to do it! At stake here are the concepts of *active* and *output* window. I suggest that before proceeding, you read the entry in the MS-BASIC Interpreter manual on the WINDOW command and function. Then study the program listing for Pretty Flashy. Finally, read the rest of this chapter for further words of wisdom regarding windows.

Program Listing 6-1
Pretty Flashy

```
'-----------------------------------------------------------------
' Pretty Flashy
' Multiple Output Window Demo
'
' Written by
' Richard C. Vile, Jr.
' January 5, 1985
' Revised February 1985
'-----------------------------------------------------------------
```

```
DEFINT A-Z

  CALL TEXTFONT (0)
  GOSUB makemenu
  GOSUB makewindows
  RE(0) = 0 : RE(1) = 0 : RE(2) = 75 : RE(3) = 145

  ON DIALOG GOSUB handledialog
  DIALOG ON

  '-----------------------------------
  ' This program spends most of its
  ' time "flashing"
  '-----------------------------------
  Flash:
    CALL INVERTRECT (VARPTR (RE(0)))
    FOR J = 1 TO delay% : NEXT J
    GOTO Flash

  handlemenu:

    MENU STOP
    DIALOG STOP
    menunumber = MENU (0)
    menuitem     = MENU (1)
    outputwindow = WINDOW (1)
    ON menuitem GOSUB bump, deck, choose, quit
    WINDOW 4, "", (310,45) - (450,75), 4
    GOSUB disp.delay
    WINDOW OUTPUT outputwindow
    IF outputwindow <> 4 THEN GOSUB disp.flash
    MENU
    MENU ON
    DIALOG ON
    RETURN

    bump:
      delay% = delay% + 50
      RETURN

    deck:
      delay% = delay% - 50
      RETURN

    choose:
      WINDOW 4, "", (10,22) - (180,77), -4
      WINDOW OUTPUT 4
```

```
    EDIT FIELD 1,STR$ (delay%), (5,10) - (105,25)
    BUTTON 1,1,"OK",(110,29) - (160,47)

  Loop:
    d = DIALOG (0)
    IF d = 1 THEN delay% = VAL (EDIT$(1)) : EDIT FIELD CLOSE 1 : GOTO Done
    IF d = 6 THEN delay% = VAL (EDIT$(1)) : EDIT FIELD CLOSE 1 : GOTO Done
    GOTO Loop

  Done:
    PRINT delay%
    WINDOW CLOSE 4

    WINDOW OUTPUT 1
    GOSUB disp.flash
    RETURN

quit:
  MENU RESET
  FOR i = 1 TO 4
    WINDOW CLOSE i
  NEXT i
  END

handledialog:
  MENU STOP
  DIALOG STOP
  outw = WINDOW (1)
  d = DIALOG (0)
  WHILE (d = 3) OR (d = 5)
    IF d = 5 THEN GOSUB refresh
    IF d = 3 THEN w = DIALOG (3)
    d = DIALOG (0)
  WEND
  IF outw = 4 THEN GOSUB disp.delay
  IF outw <> 4 THEN GOSUB disp.flash
  WINDOW w
  MENU ON
  GOSUB clear.dialog
  DIALOG ON
  RETURN

clear.dialog:
  WHILE DIALOG (0) <> 0 : WEND
  RETURN

makemenu:
  FOR mn = 1 TO 5: MENU mn,0,0,"" : NEXT mn
```

```
  MENU 1,0,1, "Flash delay"
  MENU 1,1,1, "Bump"
  MENU 1,2,1, "Deck"
  MENU 1,3,1, "Input"
  MENU 1,4,1, "Quit"
  MENU ON
  ON MENU GOSUB handlemenu
  RETURN

makewindows:
  FOR i = 1 TO 3
    WINDOW i, "", (10 + (i-1)*150, 25 + (i-1)*80)-(5 + i*150, 20 + i*80), 4
  NEXT i
  WINDOW 4, "", (310,45) - (450,75), 4
  GOSUB disp.delay

'---------------------------------
' Setup display - Pretty Flashy msgs
'---------------------------------
  FOR i = 1 TO 3
    WINDOW OUTPUT i
    GOSUB disp.flash
  NEXT i
  WINDOW OUTPUT 4
  RETURN

refresh:
   rw = DIALOG (5)
   WINDOW rw
   IF rw = 4 THEN GOSUB disp.delay
   IF rw <> 4 THEN GOSUB disp.flash
   RETURN

disp.delay:
  CALL MOVETO (20, 20)
  CALL TEXTFACE (1)
  PRINT "Delay: "; delay%;
  CALL TEXTFACE (0)
  RETURN

disp.flash:
  CLS
  CALL MOVETO (30,40)
  CALL TEXTFACE (1)
  PRINT "Pretty Flashy"
  CALL TEXTFACE (0)
  RETURN
```

ACTIVE AND OUTPUT WINDOWS

The *active window* on the Macintosh screen is always the frontmost and highlighted window. If the windows on the screen all have title bars and are overlapping, then it is easy to tell which window is highlighted and frontmost. If the windows do not overlap and if most of them are NOT document windows, that is, they have no title bars, then it may not be so easy.

The *output window* is the window to which output is currently directed. In many cases the active window and the output window are the same. It is possible, however, to have an output window that is not the active window of the moment. This allows you to generate output—text and graphics—in a window without causing it to become active. When there are two document windows involved, this can be a blessing. The output window gracefully accepts its output without its title bar being highlighted.

ACTIVITY IN WINDOWS

Information about what is happening with your windows may be had via the DIALOG function. Here's where the active, output, inactive output, and other such ideas start to be fun. In fact, I had a terrrrrrrrrible time getting Pretty Flashy to work exactly the way I wanted it to. Even when I did, it still took me a long while to figure out WHY. I left a little bug in the listing of Pretty Flashy. You should run the program and play with it awhile—see if you can find the bug. If you find it, see if you can fix it. If you fix it, see if you can explain why the change you made fixed it. Don't worry, I won't leave you hanging. If you can't figure it out, or you don't want to try, just flip to Chapter 27.

Getting back to the DIALOG function. DIALOG (0) returns information about certain kinds of events that are taking place in your program. Events are categorized by the values 1 - 7. The interpretations of these values may be found in the DIALOG entry of the MS-BASIC Interpreter manual. I suggest that you thoroughly familiarize yourself with them. I carried around flash cards for the first couple of months that I used MS-BASIC 2.0 with summaries of all the Macintosh specific commands: DIALOG, MENU, WINDOW, MOUSE, BUTTON, EDIT FIELD, and so on. I still refer to them if I haven't used a particular feature in a while.

As far as windows are concerned, the values of DIALOG that are of interest are 3, 4, and 5. For each of these values, subsidiary information is held in the DIALOG (n) function:

DIALOG (3) = Number of most recently selected *inactive* output window.
DIALOG (4) = Number of the window whose "go-away" box was most recently clicked.
DIALOG (5) = Number of an output window which needs to be refreshed.

Pretty Flashy uses both DIALOG (3) and DIALOG (5). A window needs to be refreshed if it has been covered by another window and then uncovered, for example. In a BASIC program, this has dubious value, since the BASIC program is responsible for remembering exactly what is in each window at all times. In Pretty Flashy, the window contents are always the same, so they are easy to "remember."

DIALOG (0) is what is known in computer science as a *queue.* This is a fancy word for lineup. What it boils down to is that DIALOG (0) will keep on returning nonzero values

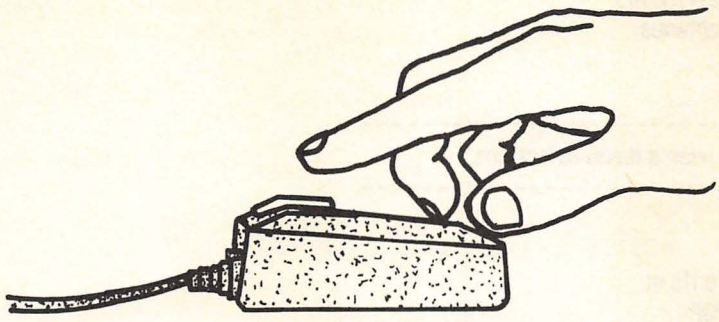
as long as there are events that have occurred. This is both good and bad. It is good in that you can always find out what is happening. It is bad in that you have to remember to ignore the events you are not interested in. The clear.dialog subroutine in Pretty Flashy does just that. It clears out the DIALOG queue after the program has processed the event of interest.

Dialog events may be trapped, just as menu events. The statements:

```
ON DIALOG GOSUB handledialog
DIALOG ON
```

will cause your program to immediately transfer to the handledialog subroutine when a dialog event is detected. In the subroutine, you can detect what events took place by using the DIALOG function. As usual, you should thoroughly familiarize yourself with the relevant entries in the interpreter manual.

Chapter 7



Window Trainer

The Window Trainer program started out life as a very simple idea. I wanted to create four windows that were as nearly equal in size as possible. I accomplished that goal with a little trial and error, but then I kept adding to the program until it reached its current inflated size. It shows some of the things you can do with multiple windows. First interact with it and then study it to see how it does what it does.

Program Listing 7-1
Window Trainer

```
'-----------------------------------------------------------------
' Window Trainer
'
' Written by
' Richard C. Vile, Jr.
' January 19, 1985
'
' Modified to add Dialog
' January 20, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM messages$(20), titles$(20)

RANDOMIZE TIMER
```

```
GOSUB Instructions
GOSUB GetMessages
GOSUB GetTitles
GOSUB MakeWindows
GOSUB MakeMenus

WHILE -1
  '------------------------------
  ' Wait for user's menu selections
  '------------------------------
WEND

This.is.where.its.at:
  MENU STOP
  menunumber = MENU (0)
  menuitem   = MENU (1)

  ON menunumber GOSUB active, woutput, actions, pick.message, pick.title
  MENU ON
  MENU
  RETURN

active:
  now.active = WINDOW (0)
  now.output = WINDOW (1)
  GOSUB Uncheck.Menu1
  MENU 1, menuitem, 2
  GOSUB Uncheck.Menu2
  MENU 2, menuitem, 2
  WINDOW menuitem
  RETURN

woutput:
  now.output = WINDOW (1)
  MENU 2, now.output, 1
  MENU 2, menuitem, 2
  WINDOW OUTPUT menuitem
  RETURN

actions:
  ON menuitem GOTO a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,quit

a1:
  '----------------------
  ' Print current message
  ' goes to output window
  '----------------------
```

```
  PRINT msg$
  RETURN

a2:
  '--------------------
  ' Clear output window
  '--------------------
  CLS
  RETURN

a3:
  '----------------------------
  ' Clear active window -
  ' even if it isn't output window,
  ' unless it's a Basic window
  '----------------------------

  output.window = WINDOW (1)
  active.window = WINDOW (0)
  IF active.window <> 0 THEN WINDOW OUTPUT active.window : CLS
  WINDOW OUTPUT output.window
  RETURN
a4:
  '----------------------------
  ' Interchange active and output
  '----------------------------
  now.active = WINDOW (0)
  now.output = WINDOW (1)
  GOSUB Uncheck.Menu1
  MENU 1, now.output, 2
  GOSUB Uncheck.Menu2
  IF now.active <> 0 THEN MENU 2, now.active, 2 ELSE MENU 2, now.output, 2
  WINDOW now.output
  IF now.active <> 0 THEN WINDOW OUTPUT now.active
  RETURN

a5:
  '-----------------------------------------
  ' Display numbers of active and output windows
  '-----------------------------------------
  PRINT "Active window: "; WINDOW (0)
  PRINT "Output window: "; WINDOW (1)
  RETURN

a6:
  '----------------------------
  ' Retitle active window
```

```
  ' Note: This makes output = active
  '------------------------------
  active.window = WINDOW (0)
  IF active.window = 0 THEN RETURN
  WINDOW active.window, titl$
  GOSUB Uncheck.Menu1
  GOSUB Uncheck.Menu2
  MENU 1, active.window, 2, titl$
  MENU 2, active.window, 2, titl$
  RETURN

a7:
  '----------------------
  ' Retitle output window
  '----------------------
  active.window = WINDOW (0)
  output.window = WINDOW (1)
  WINDOW output.window, titl$

  MENU 1, output.window, 1, titl$
  IF active.window = output.window THEN MENU 1, output.window, 2
  MENU 2, output.window, 2, titl$
  IF active.window <> 0 AND active.window <> output.window THEN WINDOW active.window
  WINDOW OUTPUT output.window
  RETURN

a10:
  '------------------
  ' Show dialog queue
  '------------------
  WHILE -1
    ON DIALOG (0) GOTO noop, noop, inactive, closebox, refresh, noop, noop
    RETURN

  noop:
    GOTO while.end

  inactive:
    PRINT "Inactive window # "; DIALOG (3); " clicked"
    GOTO while.end

  closebox:
    PRINT "Window # "; DIALOG (4); " close box clicked"
    GOTO while.end

  refresh:
    PRINT "Window # "; DIALOG (5); " needs refreshing"
```

```
while.end:
  WEND

a11:
  '------------------
  ' Enable Dialog trap
  '------------------
  ON DIALOG GOSUB a10
  DIALOG ON
  RETURN

a12:
  '-------------------
  ' Disable Dialog trap
  '-------------------
  DIALOG OFF
  RETURN

a8:
a9:
a13:
a14:
a15:
  '-------------------------------
  'Dummy labels to keep Basic happy
  '-------------------------------
  RETURN

pick.message:
  msg$ = messages$ (menuitem)
  RETURN

pick.title:
  titl$ = titles$ (menuitem)
  RETURN

quit:
  FOR i = 1 TO 4
    WINDOW CLOSE i
  NEXT i
  MENU RESET
  END

Uncheck.Menu1:
  FOR i = 1 TO 4
    MENU 1, i, 1
  NEXT i
```

```
  RETURN

Uncheck.Menu2:
  FOR i = 1 TO 4
    MENU 2, i, 1
  NEXT i
  RETURN

MakeWindows:
  WINDOW 1, "Window 1", (2,40) - (253,178)
  WINDOW 2, "Window 2", (2,200) - (253,337)
  WINDOW 3, "Window 3", (257, 40) - (509, 178)
  WINDOW 4, "Window 4", (257, 200) - (509, 337)
  RETURN

MakeMenus:
  FOR mn = 1 to 5
    MENU mn, 0, 0, ""
  NEXT mn

  MENU 1, 0, 1, "Active"
  MENU 1, 1, 1, "Window 1"
  MENU 1, 2, 1, "Window 2"
  MENU 1, 3, 1, "Window 3"
  MENU 1, 4, 2, "Window 4"

  MENU 2, 0, 1, "Output"
  MENU 2, 1, 1, "Window 1"
  MENU 2, 2, 1, "Window 2"
  MENU 2, 3, 1, "Window 3"
  MENU 2, 4, 2, "Window 4"

  MENU 3, 0, 1, "Do"
  MENU 3, 1, 1, "Print current message"
  MENU 3, 2, 1, "Clear output window"
  MENU 3, 3, 1, "Clear active window"
  MENU 3, 4, 1, "Switch active and output"
  MENU 3, 5, 1, "Display window #'s"
  MENU 3, 6, 1, "Retitle active window"
  MENU 3, 7, 1, "Retitle output window"
  MENU 3, 8, 0, "-"
  MENU 3, 9, 0, "-"
  MENU 3, 10, 1, "Show dialog queue"
  MENU 3, 11, 1, "Enable Dialog trap"
  MENU 3, 12, 1, "Disable Dialog trap"
  MENU 3, 13, 0, "-"
  MENU 3, 14, 0, "-"
```

```
  MENU 3, 15, 0, "-"
  MENU 3, 16, 1, "Quit"

  MENU 4, 0, 1, "Messages"
  FOR i = 1 TO 20
    MENU 4, i, 1, messages$ (i)
  NEXT i

  MENU 5, 0, 1, "Titles"
  FOR i = 1 TO 20
    MENU 5, i, 1, titles$ (i)
  NEXT i

  MENU ON
  ON MENU GOSUB This.is.where.its.at
  RETURN

Instructions:
  WINDOW 1, "Window Trainer with Dialog", (2,40) - (509, 337)
  PRINT "Welcome to W I N D O W  T R A I N E R"
  PRINT "Use the mouse and the menus to learn"
  PRINT "about the behavior of windows in MS-Basic"
  PRINT "programs."
  PRINT : PRINT "Resizing and moving windows is ok,"
  PRINT "but if you close a window, you might confuse"
  PRINT "the program."
  PRINT
  GOSUB Click
  CLS
  RETURN

Click:
  PRINT "Click to continue..."
  ms = MOUSE (0)
  WHILE MOUSE (0) = 0 : WEND
  WHILE MOUSE (0) <> 0 : WEND
  RETURN

GetMessages:
  FOR i = 1 TO 20
    READ messages$ (i)
  NEXT i
  msg$ = messages$ (INT (20 * RND (1)) + 1)
  RETURN

GetTitles:
  FOR i = 1 TO 20
    READ titles$ (i)
```

```
NEXT i
titl$ = titles$ (INT (20 * RND (1)) + 1)
RETURN
DATA "Brevity is the soul of wit"
DATA "Stop the world, I want to get off!"
DATA "Quoth the Raven, Nevermore!"
DATA "Fifty four forty or fight!"
DATA "Honi soit qui mal y pense"
DATA "He couldn't catch a bear in a telephone booth."
DATA "The rule is, jam tomorrow and jam yesterday - but never jam today."
DATA "A bird in the hand is safer than two overhead."
DATA "My face I don't mind it, for I am behind it..."
DATA "For the Snark was a Boojum, you see."
DATA "What garlic is to salad, insanity is to art."
DATA "When the roundle is gold, it is termed a bezant."
DATA "There's a sucker born every minute."
DATA "Food, glorious food!"
DATA "And this was the writing that was written: Mene, mene, tekel, upharsin"
DATA "The saddest words are: 'It might have been'"
DATA "Spineless jellyfish: inveterate invertebrate"
DATA "The best things in life are free."
DATA "Pneumonoultramicroscopicsilicovolcanoconiosis"
DATA "Don't believe everything you read in windows!"

'--------------
' Window titles
'--------------
DATA "I was a teenage hacker"
DATA "Bozo"
DATA "Oh frabjous day"
DATA "Tale of two windows"
DATA "Column A"
DATA "Column B"
DATA "Infundibulum"
DATA "Desserts"
DATA "Gone with the Window"
DATA "Peep Show"
DATA "Pigeonhole"
DATA "=====> Eat at Joe's <====="
DATA "Creature Feature"
DATA "Kentucky Fried Window"
DATA "Used pixels"
DATA "Axe a lotta L's"
DATA "Itsy bitsy teeny weeny bits"
DATA "This space for rent"
DATA "Fenestra"
DATA "Ivan Gelamian"
```

THINGS TO DO WITH WINDOW TRAINER

Window Trainer is to play with. Try out different manipulations with the windows. Move them around; change their sizes; close them, and so forth. Try out the different items in the Actions menu. See if you can figure out how to duplicate their effects in your own programs without consulting the listing first. Try adding new items to the Actions menu. There are several suggestions at the end of the chapter.

The **Messages** and **Titles** menus are strictly for fun. Change them if you like. Notice that they both have their full complement of items. The **Messages** menu nearly fills the screen—it still amazes me how fast what is underneath it gets restored when the menu pops back into the menu bar. Quickdraw is indeed aptly named.

128K USERS

This program has too much data in its menus for a 128K Mac. . .Sorry about that. To get it to run, you can try changing the **Messages** and **Titles** items to be shorter.

TAKE NOTICE

As you play with the program, here are some points to notice. They are general facts about windows in MS-BASIC that you need to know when you use windows in your own programs:

- Making a window active also makes it the output window.
- When the output window is different than the active window, messages appear in the output window.
- CLS has the effect of clearing the output window. If the active window and the output window are different, CLS has no effect on the active window.
- WINDOW (0) and WINDOW (1) are used to detect the active and output windows, respectively. Notice their liberal use in the Window Trainer program.
- DIALOG (0) reports on a whole list of items. If you are not automatically trapping dialog events, then whenever you choose the **Show Dialog Queue** item from the **Actions** menu, you will get a list (possibly quite long) of events that have occurred since the last time you chose the **Show Dialog Queue** action.
- Menu events are not part of the dialog mechanism.
- When you close a window and then reopen it (by making it the active window), it retains its size and position on the screen—even if you moved it from its original location or changed its size.
- When you have chosen the action **Enable Dialog Trap** from the **Actions** menu, any dialog event is immediately reported in the output window (active window if active = output).

ADDING TO THE ACTIONS MENU

Here are some actions you might try adding to the **Actions** menu:

- Clear all windows—this clears the visible part of each window. You should do this in such a way that the current output and active windows remain the same after the clearing takes place.

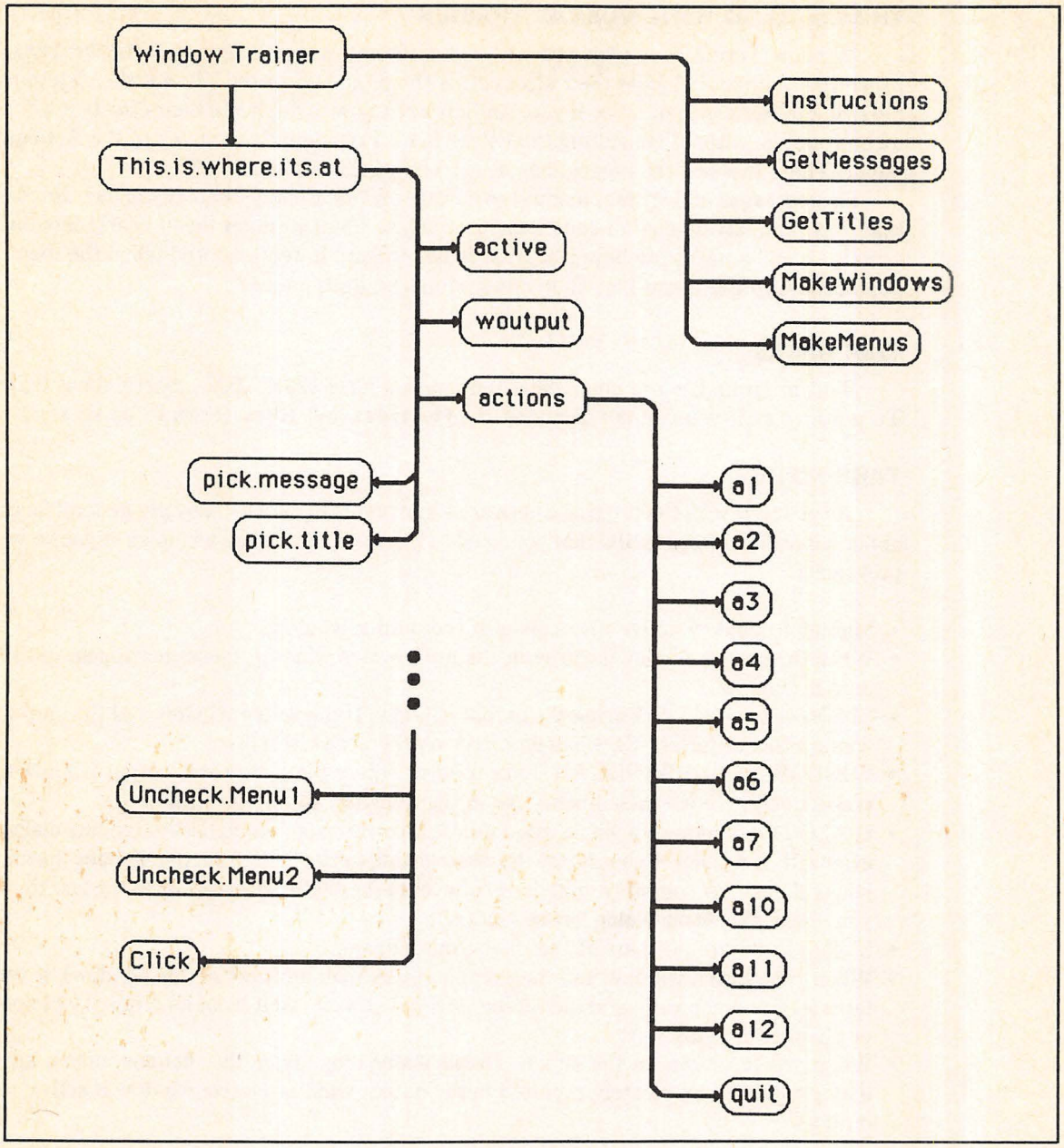


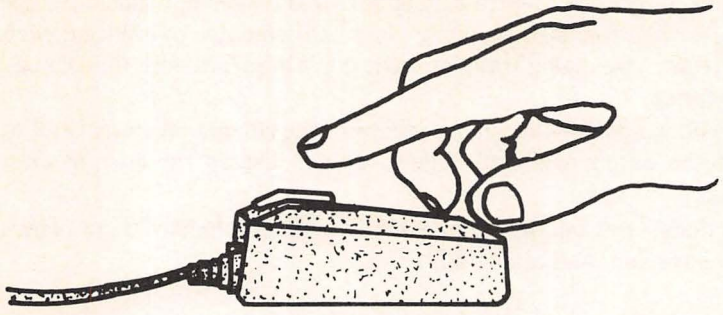
Fig. 7-1. The structure diagram for Window Trainer.

- Hide all windows—this closes all windows that are not already closed.
- Show all windows—this reopens all windows. Do not change the current active and output windows in the process.
- Hide/Show all windows—this combines the function of Hide all windows and Show all windows, but the particular choice showing on the menu depends on the last choice

made. That is, if "Hide all windows" was just chosen, then the menu choice should say "Show all windows." Likewise, if the previous action was "Show all windows," then the menu choice should say: "Hide all windows."

- Hide/Show all windows (extra challenge)—this makes the choice change to "Show all windows" when the user manually closes all windows by clicking each go away box in turn. (Hint: Use dialog trapping—but don't interfere with the Dialog capabilities of the program.)
- Message all windows—this choice will print the current message in all windows in turn (not just the output window). Again, do not disturb the current output and active windows.
- Other actions—you can dream up your own and implement them. The idea is to play, explore, have fun, and above all, learn.

Chapter 8



Elevator

Elevator is a short program that demonstrates the BUTTON statement. It also uses many graphics commands, including SCROLL, so it gives you double for your money. It was conceived as a fun introduction to the subject of buttons. Concentrate on the techniques demonstrated by the program rather than its content.

Program Listing 8-1
Elevator

```
'-----------------------------------------------------------------
' Elevator - Lots of Buttons
'
' Written by
' Richard C. Vile, Jr.
' January 23, 1985
'-----------------------------------------------------------------

DEFINT a-z

GOSUB makewindows
WINDOW OUTPUT 1
GOSUB makebuttons
WINDOW OUTPUT 2
GOSUB draw.elevator
```

```
WINDOW 1
now.floor = 1
times.to.one = 0
ON DIALOG GOSUB pushed
DIALOG ON

WHILE -1
  '----------------------
  ' Care to take a ride?
  '----------------------
  IF times.to.one > 3 THEN WINDOW CLOSE 1 : WINDOW CLOSE 2 : END
WEND

pushed:
  DIALOG OFF
  WINDOW OUTPUT 2
  CALL MOVETO (100, 40)
  d = DIALOG (0)
  IF d <> 1 THEN DIALOG ON : RETURN
  button.number = DIALOG (1)
  IF button.number - now.floor > 0 THEN up = -1 ELSE up = 0
  floors.to.go = ABS (button.number - now.floor)
  now.floor = button.number
  GOSUB ride
  IF (now.floor = 1) AND (floors.to.go > 0) THEN times.to.one = time.to.one + 1
  PRINT CHR$ (7); CHR$ (7); CHR$ (7);
  DIALOG ON
  RETURN

ride:
  IF up THEN dy = -1 ELSE dy = 1
  FOR i = 1 TO floors.to.go * 20
    SCROLL (re(1), tp + i * dy) - (re(3), bt + i * dy), 0, dy
  NEXT i
  tp = tp + floors.to.go * 20 * dy
  bt = tp + 24
  re(0) = tp : re(2) = bt
  FOR i = 1 TO 8
    CALL INVERTRECT (VARPTR (re(0)))
    FOR j = 1 TO 250 : NEXT j
  NEXT i
  RETURN

draw.elevator:
  re(0) = 8 : re(1) = 5 : re(2) = re(0) + 270 : re(3) = re(1) + 50
  CALL FRAMERECT (VARPTR (re(0)))
  re(0) = re(0) + 1 : re(1) = re(1) + 1 : re(2) = re(2) - 1 : re(3) = re(3) - 1
```

```
  CALL FRAMERECT (VARPTR (re(0)))
  re(0) = re(2) - 24 : re(1) = re(1) + 2 : re(2) = re(2) - 4 : re(3) = re(3) - 2
  CALL FRAMERECT (VARPTR (re(0)))
  bt = 275 : tp = bt - 24
  RETURN

makebuttons:
  BUTTON 1, 1, "1st floor", (10, 240) - (130, 256)
  BUTTON 2, 1, "2nd floor", (10, 220) - (130, 236)
  BUTTON 3, 1, "3rd floor", (10, 200) - (130, 216)
  BUTTON 4, 1, "4th floor", (10, 180) - (130, 196)
  BUTTON 5, 1, "5th floor", (10, 160) - (130, 176)
  BUTTON 6, 1, "6th floor", (10, 140) - (130, 156)
  BUTTON 7, 1, "7th floor", (10, 120) - (130, 136)
  BUTTON 8, 1, "8th floor", (10, 100) - (130, 116)
  BUTTON 9, 1, "9th floor", (10, 80) - (130, 96)
  BUTTON 10, 1, "10th floor", (10, 60) - (130, 76)
  BUTTON 11, 1, "11th floor", (10, 40) - (130, 56)
  BUTTON 12, 1, "12th floor", (10, 20) - (130, 36)
  BUTTON 13, 1, "14th floor", (10, 0) - (130, 16)
  RETURN

makewindows:
  WINDOW 2, "", (170,30) - (230, 315), 3
  WINDOW 1, "", (10,40) - (150, 300), 2
  RETURN
```

WHAT ELEVATOR DOES

Elevator simulates an elevator. The window on the left part of the screen shows 13 buttons. They are labelled with floor numbers from 1 to 14. By tradition, the 13th floor does not exist. The elevator is represented by a small rectangle inside a second window on the right part of the screen. This second window represents the elevator shaft. To make the elevator "go," you simply press one of the buttons. The elevator then proceeds from whatever floor it is next to now to the floor corresponding to the button you pushed. When you become bored with riding the elevator, simply return to the first floor enough times and the program quits. Figure 8-1 shows the elevator program in action.

THE ELEVATOR BUTTONS

The elevator buttons are created using the BUTTON command. Of all the interactive Mac programming commands added to BASIC 2.0, this is probably the easiest to use. If you haven't already done so, now is the time to read the description of the BUTTON command in the MS-BASIC Interpreter manual. The use of BUTTON in Elevator is completely straightforward—no tricks, no surprises.

THE ELEVATOR DIALOG

In order to catch the user in the act of pushing one of the elevator buttons, a dialog

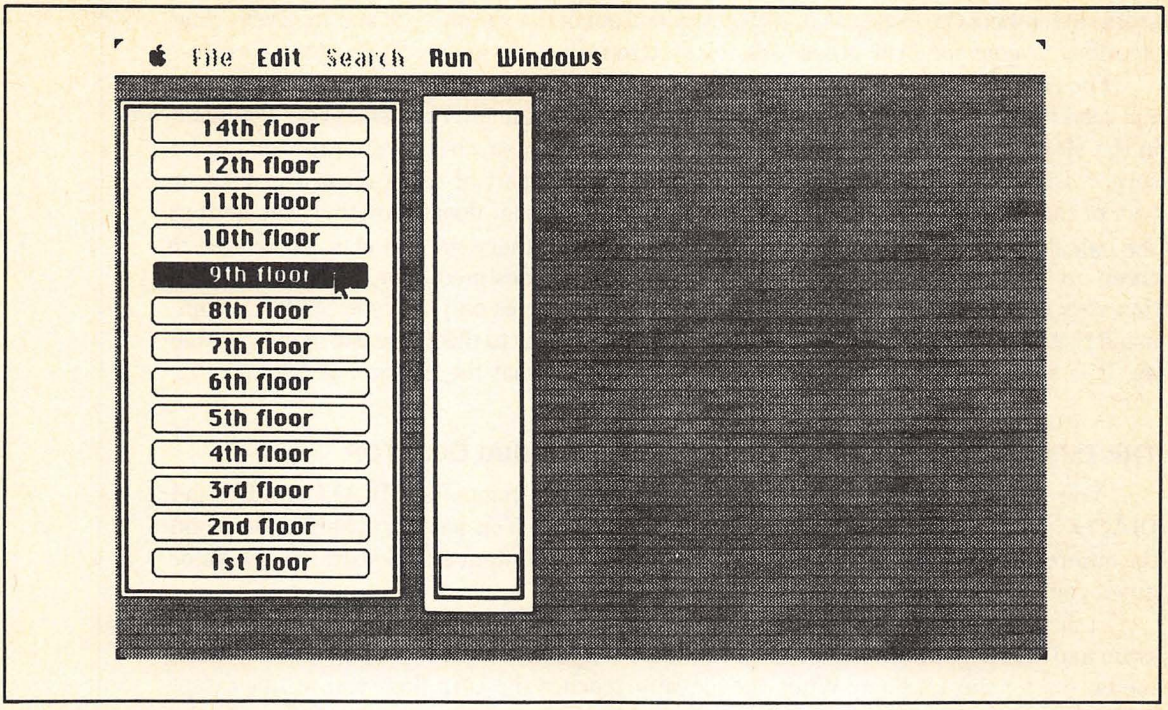


Fig. 8-1. Going up?

trap is set up. This is the same kind of trap discussed in Chapter 6. The purpose here is to catch dialog events involving button pushes. They may be detected by the fact that the value of DIALOG (0) is equal to 1. When that is true, a button has been pushed (clicked, whatever) in the active window. The number of the button that was pushed may be found by consulting DIALOG (1). In Elevator, all this takes place in the subroutine called "pushed." Once the subroutine finds out which button got pushed, it is then a matter of going for an elevator ride. The program remembers which floor the elevator is on now with the variable **now.floor**. The difference between **button.number** and **now.floor** tells the program both how many floors it needs to travel and which direction it needs to go in. The elevator ride is accomplished in the subroutine "ride" which I discuss in the next section. When the destination is reached, the program rings the Mac bell three times.

TAKING THE RIDE

The elevator is drawn using simple calls to the Quickdraw Routines built into the Macintosh ROM. If you have not already done so, please read and study Appendix F of the MS-BASIC Interpreter manual (Access to Macintosh ROM Routines).

The elevator shaft and the elevator itself are drawn with calls to the FRAMERECT ROM routine. The shaft is two lines thick and is drawn by two successive calls to FRAMERECT. In between the calls, the framing rectangle is changed by adding one to the top and left coordinates and subtracting one from the bottom and right coordinates. This is a standard way to achieve a box with a two line thick border. Iterating this process gives boxes with even thicker boundary lines. After the elevator is drawn, the pro-

gram remembers the locations of its top and bottom in the variables **bt** and **tp**. Every time the elevator goes for a ride, these are adjusted to reflect the new location of the elevator.

The elevator ride itself is accomplished with the use of the SCROLL command. You can read its description in the MS-BASIC Interpreter manual. The use of this command in the elevator program is simple minded enough. Even so, there were problems to be solved. One of the problems was how to avoid erasing part of the walls of the shaft or part of the elevator itself when riding to the top or bottom floors. Another was getting the calculation exactly right about how far to scroll—otherwise, the elevator tended to creep up or down and eventually not to stop next to its destined floor. This was a situation very undesirable for passengers! The solution involves only a single SCROLL command that takes care of moving both up and down. The key to this is the use of the variable **dy**. It is set to +1 or −1, depending on the direction that the elevator is taking.

THE DIFFERENCE BETWEEN DIALOG OFF AND DIALOG STOP

You should have read the discussion of the two commands DIALOG OFF and DIALOG STOP in the MS-BASIC Interpreter manual. You have probably understood the difference between them. Just in case you didn't or you want a demonstration, Elevator gives you a chance to see the difference in action.

In the subroutine "pushed" there is a DIALOG OFF statement. Try running the program and pushing the button for the 14th floor. While the elevator is on its way up, push the button for the 1st floor. When the elevator reaches the 14th floor you will be disappointed (but you shouldn't be surprised) to find that it refuses to come back down to the 1st floor until you push the button again. This, of course, is because the DIALOG OFF causes the first push of the 1st floor button to be ignored. The second push occurs after the program has returned from the subroutine "pushed" and has executed a DIALOG ON statement.

Now try changing the DIALOG OFF statement to DIALOG STOP. Then repeat the above experiment. This time the elevator turns right around and comes back down. You don't have to push the button a second time! In fact, if you push other buttons while the elevator is on its way up to the 14th floor, all the floors whose buttons you push will be visited by the elevator in turn until none are left in the dialog queue.

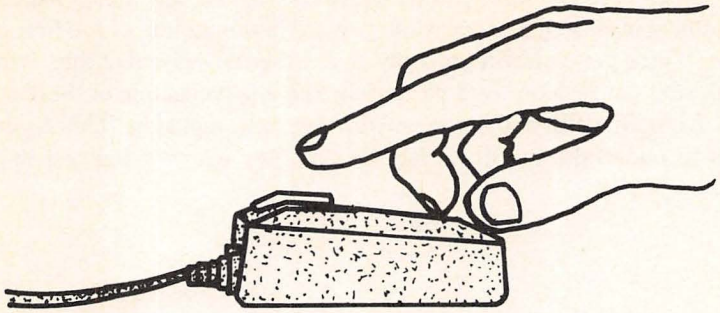
A CHALLENGE

Here is a potentially interesting and challenging project. See if you can get the elevator to stop at a floor whose button is pressed after the button for the floor it is bound for was pressed. It should stop provided that the floor whose button was pressed second is on the way to the floor whose button was pressed first, and that when the second button was pressed, the elevator hadn't already gone by the floor of the second button. Maybe an example will help you understand the insane idea I have in mind here. Let's say the button for the 14th floor is pressed. The elevator starts its journey upward from the 1st floor. Before it passes the 10th floor, you push the button for the 10th floor. Now a real elevator would probably stop at the 10th floor on the way up. For now let's ignore the question of whether the push of the 10th floor button was a request to go UP or to go DOWN—just to keep things simple! Can you reprogram Elevator so that the programmed elevator also stops at the 10th floor under these circumstances? Even better yet, can you

make it so that if the elevator has already passed the 10th floor on the way up, that it keeps on going to the 14th floor and then comes back down.

I haven't tried to meet this challenge myself, so I'm not sure how difficult it will be to carry out. But here is a hint: you will have to set up a new dialog handling subroutine that traps dialog events that occur while you are in the middle of the first dialog trapping subroutine. Worse yet, you will probably have to set up several of these which are visited in a last in—first out fashion. And ultimately, the innermost one of these subroutines will have to set DIALOG OFF and ignore any further button pushes. This is your task, should you accept to undertake it. Otherwise, this chapter will self-destruct in 30 seconds.

Chapter 9



Pattern Maker

There are more buttons in this program than you can shake a stick at. In fact, there are probably more buttons than you would ever think of using in a "real" program. On the other hand, if it's a program just for you and it's in BASIC—well, why not go a little crazy?

The purpose of the program is to generate the data for a Macintosh *pattern*, the kind of pattern you find in MacPaint or on the desktop in the control panel. It could have been implemented with a grid drawn by using calls to the Quickdraw routines. Doing it this way was a lot more fun however.

THE PROGRAM LISTING

This program listing contains the symbol ✣. It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. The reason for this notation is that certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✣. For example, if the listing shows

```
IF XO < X THEN   ✣
    inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN  inside = 0 : RETURN
```

Program Listing 9-1
Pattern Maker

```
'-----------------------------------------------------------------
' Build Pattern Using
' Radio Buttons
'
' Written by
' Richard C. Vile, Jr.
'
' January 31, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM pattern(3), masks(8), bytes(8)

GOSUB make.grid
GOSUB get.mask.data
CALL MOVETO (16,265)
PRINT "Draw a pattern in the grid"
BUTTON 75,1,"Ok",(16,175)-(41,195)
BUTTON 76,1,"Cancel",(16,200)-(81,220)
BUTTON 77,1,"Preview",(16,225)-(100,245)
GOSUB grid.dialog
GOSUB record.pattern
WINDOW CLOSE 1
WINDOW 1
GOSUB print.pattern
END

record.pattern:
  FOR i = 1 TO 8 : bytes(i) = 0 : NEXT i
  FOR col = 1 TO 8
    FOR row = 1 TO 8
      btn = col*8 + row
      IF BUTTON (btn) = 2 THEN GOSUB bit.on
    NEXT row
  NEXT col
  FOR j = 1 TO 8
    k = INT ((j-1)/2)
    IF (j MOD 2) = 0 THEN POKE VARPTR (pattern(k))+1, bytes(j)   +
      ELSE POKE VARPTR (pattern(k)), bytes(j)
  NEXT j
  RETURN

show.pattern:
  re(0) = 0 : re(1) = 0 : re(2) = 350 : re(3) = 500
  CALL PENPAT (VARPTR (pattern(0)))
```

```
  CALL PAINTRECT (VARPTR (re(0)))
  RETURN

bit.on:
  bytes(row) = bytes(row) + masks(col)
  RETURN

print.pattern:
  PRINT "Pattern data:"
  FOR i = 0 TO 3
    PRINT pattern(i)
  NEXT i
  RETURN

grid.dialog:
  done = 0
  WHILE NOT done
    d = DIALOG (0)
    IF d = 1 THEN GOSUB check.buttons
    IF d = 6 THEN done = -1
  WEND
  RETURN

check.buttons:
  btn = DIALOG (1)
  IF btn = 75 THEN done = -1: RETURN
  IF btn = 76 THEN done = -1 : cancel = -1: RETURN
  IF btn = 77 THEN GOSUB preview : RETURN
  IF BUTTON (btn) = 2 THEN BUTTON btn, 1 ELSE BUTTON btn, 2
  RETURN

preview:
  GOSUB record.pattern
  GOSUB click
  GOSUB show.pattern
  GOSUB click
  CLS
  RETURN

make.grid:
  FOR col = 1 TO 8
  FOR row = 1 TO 8
    BUTTON row*8 + col, 1, "",  +
      (row*16,col*16)-(((row+1)*16)-2, ((col+1)*16)-2), 3
  NEXT row
NEXT col
RETURN
```

```
click:
  PRINT "click to continue"
  ms = MOUSE (0)
  WHILE MOUSE (0) = 0 : WEND
  WHILE MOUSE (0) <> 0 : WEND
  RETURN

get.mask.data:
  FOR i = 0 TO 8
    READ masks (i)
  NEXT i
  RETURN

DATA 0
DATA &H80, &H40, &H20, &H10
DATA &H8, &H4, &H2, &H1
```

GENERAL NOTES

This program was originally written with the idea of including it in another program. It was to be used in that program to edit patterns that the program itself made use of. This kind of capability should be familiar to you—compare MacPaint with its pattern editing mode. Consider also the Macintosh control panel with its desktop pattern editor. In fact I have included a program that will use the patterns later in the book—see Chapter 23.

The Pattern Maker is written in such a way that it runs for one pattern and then quits. You can make more patterns by rerunning it. As an exercise, why don't you try to modify it to run in a loop. Make each repetition of the loop produce a new pattern. If you make this change, you can use the program until you get tired of making patterns. The Cancel button might be used to leave the program. In the context of a larger program, Cancel is provided to mean, "Oops, I really didn't want to edit a pattern after all. Please let me out." In a stand-alone version, Cancel could mean Quit. You could change the Cancel button to a Quit button if you prefer.

There are really a lot of buttons in this program. One limit on how many buttons can be created is the size of the Mac's memory. I have a 512K Mac, and I tried the following silly program once. I call it "Button Mania."

Button Mania

```
btn = 1
WHILE -1
  x = INT (RND * 450) : y = INT (RND * 325)
  BUTTON btn, 1, str$ (btn), (x,y)-(x+50,y+16), 1
  btn = btn + 1
WEND
```

This causes buttons galore, all over the screen! After a while, the program starts to spin the disk drives. When that happens, the button creation speed slows a bit. I really don't understand why the disks start to go, but it no doubt has to do with the way that

MS-BASIC keeps track of its buttons. In any case, it became extremely difficult to get a response out of the BASIC interpreter once this started. I eventually had to press the interrupt button on the programmer's switch. Button Mania is not recommended for the faint of heart!

PROGRAM BEHAVIOR AND TECHNIQUES

A Macintosh pattern consists of four sixteen bit integers, stored in an array. The Pattern Maker program keeps its pattern in the array of the same name; **pattern**. The four numbers in the array are interpreted as a bit pattern laid out in 8 rows of 8 bits (or one byte) each. When the pattern is transferred to the screen, if a bit is on, the corresponding screen pixel is turned on. If a bit is off, the corresponding screen pixel is turned off. In the 8×8 layout, each sixteen bit integer is broken into two halves. The half corresponding to the higher order bits of the sixteen bit number appears first. This is illustrated in detail in Fig. 9-1.

BUILDING THE PATTERN

The user simply clicks buttons in the grid to turn bits in the pattern on or off. When a button is clicked, the program uses the button function to detect the current state of the button. If the button is active, it is made inactive and vice versa. When it comes time to record the pattern, the same technique is used in order to detect which bits should be on and which should be off.

The pattern is constructed by calculating each of the eight bytes that correspond to the eight rows of the pattern. The individual bytes are computed by adding up the bit values corresponding to the buttons of that row that are active. Each element of the array **masks** represents a number with exactly one bit turned on. These values are the ones used in the subroutine "bit.on" to compute the final byte value.

In order to be able to use the pattern in calls to Macintosh ROM routines, it must be put into the array **pattern**. This in turn involves combining the two bytes of each pair of adjacent rows in the grid into a single 16-bit integer. Since integers in MS-BASIC are signed numbers, the following technique will not work:

```
pattern (i) = bytes (row*2) + bytes (row*2 + 1) * 256
```

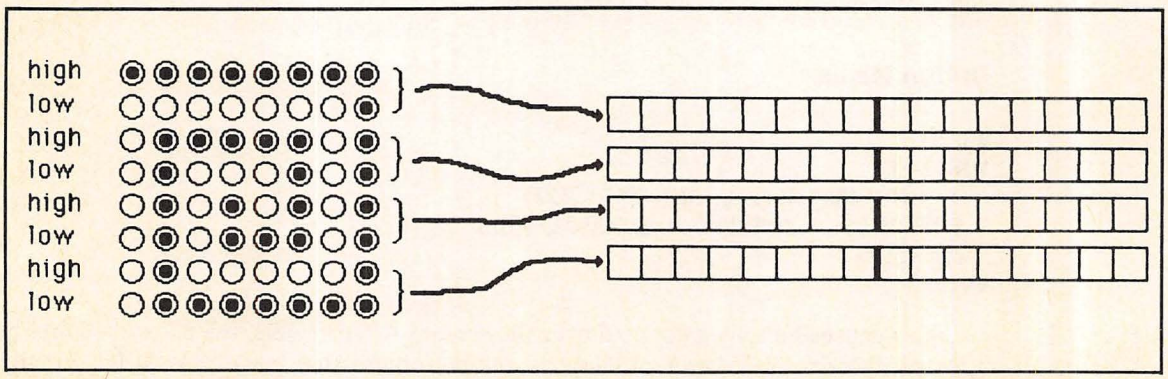


Fig. 9-1. The bits in a BASIC integer illustrated via a pattern.

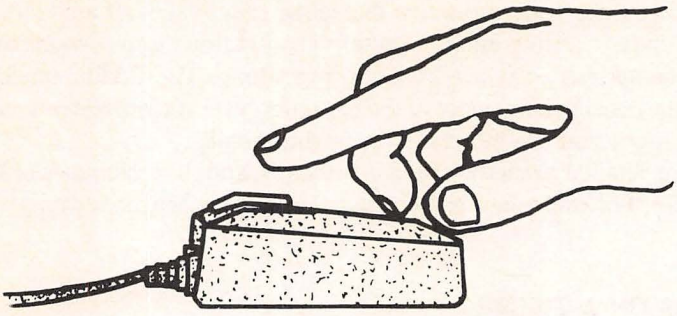
It will not work because values computed with such an equation are potentially greater than 32767. Such computations may result in an *overflow* error in MS-BASIC. There are many ways to avoid the overflow message and still get the 16-bit values into the **pattern** array. I have chosen a rather primitive technique using pokes. It amounts to copying the **bytes** array a byte at a time into the **pattern** array, taking into account the fact that the elements of the **pattern** array are each two bytes long. The BASIC function that allows you to locate the position in memory of the elements of the **pattern** array is called VARPTR. Study the listing if you are interested in further details.

For a more detailed discussion of bits, bit-maps, and the techniques of Pattern Maker, see Chapter 22. For more figures showing the Pattern Maker technique in action, see Chapter 23.

VARIATIONS ON A THEME

The way the program works now, when the Preview button is clicked, the right half of the screen is filled with replication of the pattern. If you change the assignment `re(1) = 175` to `re(1) = 0` in the subroutine show pattern, the preview will paint the entire screen. Notice how this obliterates all the buttons. Notice further how the buttons regenerate themselves after the pattern is erased. The only BASIC statement that is executed when this happens is CLS. The buttons reappear automatically because the Macintosh knows that they should be there. When the screen is cleared, any buttons that were previously obscured will magically reappear.

Chapter 10



Address Book

Here is another program which could be made a part of a larger program. It is the data entry segment of an online Macintosh Address Book. It allows you to fill in the usual Name, Address, Phone number, and so on. It illustrates a number of points about edit fields and dialog involving edit fields. The form used by the program is illustrated in Fig. 10-1.

Program Listing 10-1
Address Book Entry

```
'-------------------------------------------------------------------
' Address book program
'
' Written by
' Richard C. Vile, Jr.
' January 15, 1985
'-------------------------------------------------------------------
DEFINT a-z

Entries = 25
DIM Person$ (Entries)
DIM  Street$ (Entries), City$ (Entries), State$ (Entries), Zip$ (Entries)
DIM Phone$ (Entries)
```

```
' Test data entry format

GOSUB screen.form
pn = 1
active.field = 1
EDIT FIELD 1
DIALOG ON
ON DIALOG GOSUB next.person

WHILE -1
WEND

next.person:
   d = DIALOG (0)
   ON d GOTO button.press, edit.field, none, none, none, return.press, tab.press
   GOTO next.person

button.press:
   button.selected = DIALOG (1)
   IF button.selected = 1 THEN GOSUB display.results
   GOTO done

edit.field:
   active.field = DIALOG (2)
   EDIT FIELD active.field
   GOTO next.person

tab.press:
   active.field = (active.field MOD 6) + 1
   EDIT FIELD active.field
   GOTO next.person

none:
   GOTO next.person

return.press:
   k$ = INKEY$
   IF k$ = CHR$(3) THEN GOSUB display.results
   GOTO tab.press

done:
   pn = pn + 1
   GOSUB screen.form
   RETURN

screen.form:
   CLS
```

```
  BUTTON 1, 1, "Ok", (200, 120) - (250, 140)
  EDIT FIELD 6, "Phone#", (10, 100) - (250, 116)
  EDIT FIELD 5, "Zip", (190, 80) - (250,96)
  EDIT FIELD 4, "State", (10,80) - (185, 96)
  EDIT FIELD 3, "City", (10, 60) - (250, 76)
  EDIT FIELD 2, "Street", (10, 40) - (250,56)
  EDIT FIELD 1, "Name", (10,20) - (250, 36)
  RETURN

display.results:
  Person$ (pn) = EDIT$ (1)
  Street$ (pn) = EDIT$ (2)
  City$ (pn) = EDIT$ (3)
  State$ (pn) = EDIT$ (4)
  Zip$ (pn) = EDIT$ (5)
  Phone$ (pn) = EDIT$ (6)
  FOR i = 1 TO 6
    EDIT FIELD CLOSE i
  NEXT i
  BUTTON CLOSE 1
  CLS
  PRINT "Name:"; TAB (20);Person$ (pn)
  PRINT : PRINT "Street:"; TAB (20); Street$ (pn)
  PRINT "City:"; TAB (20); City$ (pn)
  PRINT "State:"; TAB (20); State$ (pn); TAB(45); "Zip:    "; Zip$ (pn)
  PRINT : PRINT "Phone#:"; TAB (20); Phone$ (pn)
  WHILE MOUSE (0) = 0 : WEND
  WHILE MOUSE (0) <> 0 : WEND
  RETURN
```

GENERAL NOTES

Address Book illustrates how a simple form is filled out. EDIT FIELD statements are used to set up the form on the screen. Then DIALOG (0) and DIALOG (1) are used to manage the interaction with these fields. Finally, the EDIT$ function is used to retrieve the contents of each field for recording elsewhere in the program. And that, as they say, is that!

RETURN OR ENTER AND WHAT TO DO WHEN EITHER IS PRESSED

When filling out a form, the user has a variety of choices for moving between fields. The program must also decide on how to respond to certain activities.

The DIALOG (0) function is capable of detecting the following situations:

- The user presses Tab while in the process of entering information into an edit field.
- The user clicks the mouse in a different edit field than the one currently being edited.
- The user presses the Return key in an edit field that was defined so as not to allow

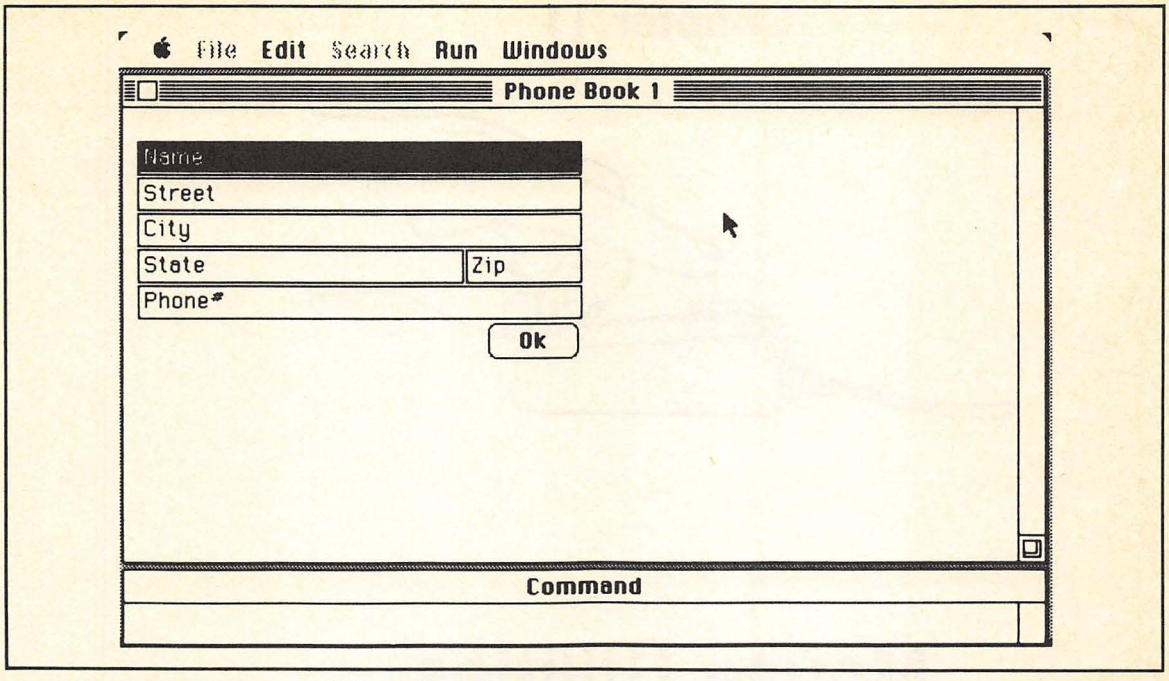


Fig. 10-1. A simple input form using edit fields.

Return keys. What that means is that no explicit carriage returns are allowed to be embedded in the text that is entered in that field. This is usually the case for short fields in a form like the one used in this program. The Enter key is considered to be equivalent to the Return key for this purpose.

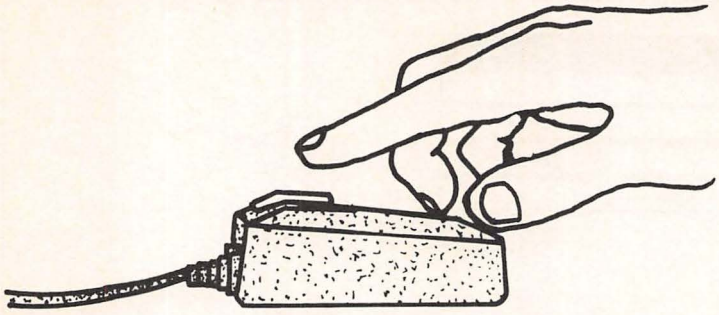
Unfortunately, there seems to be no way to distinguish between the user pressing the Return key and the user pressing the Enter key while in an edit field. This is too bad considering that you can distinguish between the two keys in "raw" input. The BASIC commands:

```
k$ = INKEY$
PRINT ASC (k$)
```

produce a value of 13 for the Return key and a value of 3 for the Enter key. It would be nice to allow Return to be equivalent to Tab in some situations and require Enter in order to stop filling in the form. However, the equating of Enter and Return inside edit fields makes this impossible. The subroutine return.press in the current example was an attempt to make the Enter key work this way. Of course, it failed. The program does treat Return as equivalent to Tab, since the subroutine transfers control to the Tab key handling subroutine.

All of this is merely to point out that you have a choice in your programs. You are not compelled to treat Return as equivalent to the Ok button.

Chapter 11



Mouse Utilities

Menus, windows, buttons, and so on are all nice features of the Macintosh user interface. But the heart and soul of the Mac is the mouse. Almost every advertisement you see for the Macintosh emphasizes pointing and clicking with the mouse. Naturally, you will want to make the use of the mouse an integral part of your programs. Event trapping for windows, menus, dialog, and so on gives you many opportunities to use the mouse in an indirect manner. In this chapter you will find a collection of short programs and subroutines that involve using the mouse more directly.

MS-BASIC gives you access to information about the mouse via the MOUSE function. The code provided here makes ample use of that information. I have no doubt that you will find many ways to use the mouse techniques provided in this chapter in your own MS-BASIC applications.

Program Listings 11-1 to 11-5

Mouse Utilities

Listing 11-1

```
'-----------------------------------------------------------------
' Request user to Click mouse
' before program will continue
'-----------------------------------------------------------------
Click:
  PRINT "Click to Continue..."
  ms = MOUSE (0)
```

```
   WHILE MOUSE (0) = 0 : WEND
   WHILE MOUSE (0) <> 0 : WEND

   RETURN
```

Listing 11-2

```
'-------------------------------------------------------------------
' Position x-y axes (Crosshairs)
'
' Written by
' Richard C. Vile, Jr.
' February 3, 1985
'-------------------------------------------------------------------
DEFINT a-z
DIM drag.vertical (400), drag.horizontal (400)

CLS
FOR i = 1 TO 75 STEP 5
  FOR j = 1 TO 5
    PRINT "This is a test";
  NEXT j
  PRINT
NEXT i

WHILE MOUSE (0) = 0 : WEND
WHILE -1

  x1 = MOUSE (3) : y1 = MOUSE (4)

loop:

    GOSUB get.lines
    GOSUB draw.axes
    xold = x1 : yold = y1
    IF MOUSE (0) >= 0 THEN freeze
    GOSUB erase.axes
    GOSUB redraw.screen
    x1 = MOUSE (5) : y1 = MOUSE (6)
    GOTO loop

freeze:
  WHILE MOUSE (0) = 0 : WEND
  GOSUB erase.axes
  GOSUB redraw.screen

'WEND

END
```

```
erase.axes:
  LINE (x1, 0) - (x1, 350), 30
  LINE (0,y1) - (500,y1), 30
  RETURN

redraw.screen:
  PUT (x1, 0) - (x1, 350), drag.vertical, OR
  PUT (0,y1) - (500,y1), drag.horizontal, OR
  RETURN

draw.axes:
  LINE (x1, 0) - (x1, 350)
  LINE (0,y1) - (500, y1)
  RETURN

get.lines:
  GET (x1, 0) - (x1, 350), drag.vertical
  GET (0, y1) - (500, y1), drag.horizontal
  RETURN
```

Listing 11-3

```
'-----------------------------------------------------------------
' Position x-y axes (Crosshairs)
' "Flying" Crosshairs version - Crosshairs follow cursor until click
'
' Written by
' Richard C. Vile, Jr.
' February 3, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM drag.vertical (400), drag.horizontal (400)

CLS
FOR i = 1 TO 75 STEP 5
  FOR j = 1 TO 5
    PRINT "This is a test";
  NEXT j
  PRINT
NEXT i

WHILE MOUSE (0) <> 0 : WEND
WHILE -1

  x1 = MOUSE (3) : y1 = MOUSE (4)

loop:

    GOSUB get.lines
```

```
    GOSUB draw.axes
    xold = x1 : yold = y1
    IF MOUSE (0) < 0 THEN freeze
    GOSUB erase.axes
    GOSUB redraw.screen
    x1 = MOUSE (1) : y1 = MOUSE (2)
    GOTO loop

freeze:
  WHILE MOUSE (0) <> 0 : WEND
  GOSUB erase.axes
  GOSUB redraw.screen
  WEND
  END

  erase.axes:
    LINE (x1, 0) - (x1, 350), 30
    LINE (0,y1) - (500,y1), 30
    RETURN

  redraw.screen:
    PUT (x1, 0) - (x1, 350), drag.vertical, OR
    PUT (0,y1) - (500,y1), drag.horizontal, OR
    RETURN

  draw.axes:
    LINE (x1, 0) - (x1, 350)
    LINE (0,y1) - (500, y1)
    RETURN

  get.lines:
    GET (x1, 0) - (x1, 350), drag.vertical
    GET (0, y1) - (500, y1), drag.horizontal
    RETURN
```

Listing 11-4

```
'-------------------------------------------------------------------
' Detect whether co-ordinate
' pair is inside a given rectangle
' Method Number 1
'
' Co-ordinates in x0, y0
' Rectangle specified by:
'    tp0, lf0, bt0, rg0
'-------------------------------------------------------------------

Inside.Rect:
  inside = -1
```

```
  IF (x0 <= lf0) OR (x0 >= rg0) THEN inside = 0
  IF (y0 <= tp0) OR (y0 >= bt0) THEN inside = 0
  RETURN
```

Listing 11-5

```
'-----------------------------------------------------------------
' Dragging open a rectangle
'
' Written by
' Richard C. Vile, Jr.
' January 24, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM top.line(400), bottom.line(400), left.line(400), right.line(400)

CLS
FOR i = 1 TO 75 STEP 5
  FOR j = 1 TO 5
    PRINT "This is a test";
  NEXT j
  PRINT
NEXT i

WHILE MOUSE(0) = 0 : WEND
WHILE -1

  x0 = MOUSE(3) : y0 = MOUSE(4)
  x1 = x0 : y1 = y0

loop:

    GOSUB get.lines
    GOSUB draw.rect
    xold = x1 : yold = y1
    IF MOUSE(0) >= 0 THEN freeze
    GOSUB erase.rect
    GOSUB redraw.screen
    x1 = MOUSE(5) : y1 = MOUSE(6)
    GOTO loop

freeze:
  WHILE MOUSE(0) = 0 : WEND
  GOSUB erase.rect
  GOSUB redraw.screen

WEND
END
```

```
erase.rect:
  LINE (x0,y0) - (x1,y0), 30
  LINE (x0,y0) - (x0,y1), 30
  LINE (x1,y0) - (x1,y1), 30
  LINE (x0,y1) - (x1,y1), 30
  RETURN

redraw.screen:
  PUT (x0,y0) - (x1,y0), top.line, OR
  PUT (x0,y0) - (x0,y1), left.line, OR
  PUT (x1,y0) - (x1,y1), right.line, OR
  PUT (x0,y1) - (x1,y1), bottom.line, OR
  RETURN

draw.rect:
  LINE (x0,y0) - (x1,y0)
  LINE (x0,y0) - (x0,y1)
  LINE (x1,y0) - (x1,y1)
  LINE (x0,y1) - (x1,y1)
  RETURN

get.lines:
  GET (x0,y0) - (x1,y0), top.line
  GET (x0,y0) - (x0,y1), left.line
  GET (x1,y0) - (x1,y1), right.line
  GET (x0,y1) - (x1,y1), bottom.line
  RETURN
```

CLICK TO CONTINUE

How many times have you seen the words **Click to Continue** on your Macintosh screen? Now you can use them in your own programs. The Subroutine Click shown in Listing 11-1 uses the MOUSE function to provide the capability.

MOUSE (0) is always 0 when the button is up and nonzero when it is down. The specific nonzero values can be used to distinguish between various situations, but that is described later in the book. For now, you simply need to detect first the click and then the "unclick." The subroutine does this with two WHILE loops. The first waits for the value of MOUSE(0) to become nonzero (click). The second then waits for the value of MOUSE(0) to become zero again (unclick).

The statement ms = MOUSE (0) is used to "clear the mouse." Since the MOUSE function measures what took place since the last time it was called, this is simply to ensure that the subroutine is reflecting current events.

The reason for waiting for the unclick is to prevent the same click from being processed twice. This is relatively unimportant in situations where the program is waiting for the click in order to proceed; however, in other cases, a click may signal the program to take an action that "toggles" between two states. If the same click is processed as two clicks, it can be difficult for the user to get the state he or she wants.

CROSSHAIRS AND FLYING CROSSHAIRS

These programs are interesting because they illustrate what you can do with combinations of MS-BASIC statements. In this case GET and PUT team up with the MOUSE function. Both programs enable you to position a pair of coordinate axes represented by a vertical and a horizontal line somewhere on the screen.

The first variation requires you to click the mouse and drag the coordinates into the desired position. After you click, the crosshairs follow the cursor around the screen until you unclick. Then the coordinates are frozen on the screen in the position where you released the mouse button.

The second variation puts the crosshairs up immediately and causes them to follow the cursor until the mouse is clicked. Once the mouse is clicked, the axes freeze into position until you unclick at which point they start to "fly" again. In an application, the first click could serve to position them in preparation for some other action. I give an example of this later in the book as part of the Dialog Designer program.

MOUSE INSIDE RECTANGLE

Suppose you write a program that draws rectangles on the screen with either MS-BASIC graphics commands or CALLs to the Quickdraw ROM routines. Suppose further that you want to detect whether or not a pair of coordinates provided dynamically by your program lies inside the rectangle. The coordinates are returned by calls to the MOUSE function under certain circumstances: mouse click, start of mouse drag, end of mouse drag, and so on. In the example, I use x0 and y0 to represent these coordinates. In an application, they would be provided by some other part of the program. The parameters of the rectangle are represented in the subroutine by t0, l0, b0, r0 (top, left, bottom, right). The routine speaks for itself—the comparisons represent a translation into BASIC of the geometric condition of the point (x0, y0) lying inside the rectangle:

x0 is to the right of the left side of the rectangle AND
x0 is to the left of the right side of the rectangle

y0 is below the top of the rectangle AND
y0 is above the bottom of the rectangle

Actually, the program uses the logical *negation* of these conditions to detect when the point lies outside the rectangle:

x0 is to the left of the left side of the rectangle OR
x0 is to the right of the right side of the rectangle

y0 is above the top of the rectangle OR
y0 is below the bottom of the rectangle

The subroutine of the listing is written so that the boundary lines of the rectangle are considered to be outside the rectangle. What change would need to be made to the program so that the boundary lines were considered to be inside the rectangle?

DRAGGING OPEN A RECTANGLE

This example is similar in spirit to the Crosshairs examples. It waits until the mouse is clicked on. The point at which the click occurs is taken as the upper left corner of a rectangle to be formed on the screen. As long as the mouse button is held down, the moving cursor is taken as the lower right hand corner of a rectangle which is alternately formed and erased. The rectangle thus appears to be blinking or shimmering as long as the mouse is held down. When the button is released, the rectangle freezes into position. In an application, this could be used to allow the user to specify a rectangle on the screen for some purpose. Of course, you are already familiar with this idea from MacPaint and the Finder. The rectangles opened in those examples don't shimmer in the same way, but the spirit is similar.

The interesting part of this example as well as the Crosshairs examples is the way that the screen appearance is left unchanged even though various lines go moving about and covering up the pixels on the screen. This is accomplished by having the program remember what lies under the sides of the rectangle as it shimmers around. The part of the screen that gets covered up by the lines of the rectangle or the axes of the crosshairs is restored whenever those lines or axes move to a new position. All of this is accomplished with the GET and PUT commands. The outline for this kind of program is a loop that takes the following actions:

- Remembers what is on the screen now in the places you are about to cover up. This action uses GETs.
- Draws the figure you are moving around in the places whose contents you just remembered.
- Determines the new location of the cursor (which may or may not have actually moved).
- Erases the figure you drew.
- Restores the screen to the way it looked before you covered it up. This uses PUTs.

This loop continues until the appropriate exit condition is met. This condition is usually one of two events: either the mouse is clicked or unclicked. The difference between the two is illustrated by the difference between the Crosshairs and Flying Crosshairs routines.

CHALLENGES

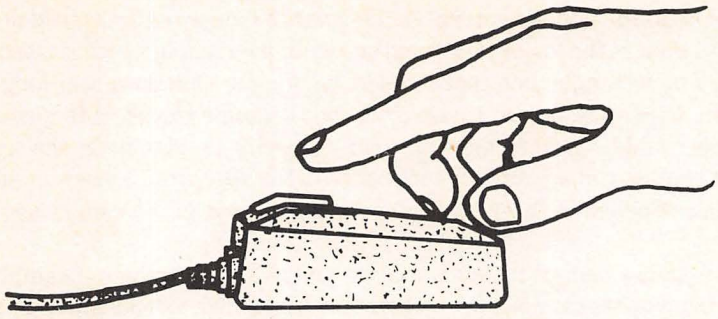
In the spirit of the examples of this chapter, here are a couple of challenges for you to try.

Write a subroutine that does nothing until the mouse is moved. That is, it first detects the current coordinates of the mouse; then it simply waits until the user moves the mouse to a different location. A variation on this theme could be to wait until the cursor is moved out of or into a particular rectangle on the screen.

Write a subroutine that detects whether or not the mouse is clicked inside a given circle on the screen. The circle can be specified by a point, (X0,Y0), and a radius, R.

Write BASIC code that causes the cursor to change when the mouse is moved inside or outside a given rectangle. This sort of thing happens all the time in "real life" on the Mac. For information on redesigning the cursor, see the MS-BASIC Interpreter manual appendix on the Quickdraw interface.

Chapter 12



Quickdraw Trainer

Quickdraw Trainer provides a "Cook's tour" of the Macintosh ROM calls that have been made available from Microsoft BASIC. This program started out as a simple idea and grew like topsy. I have left parts of it incomplete for you to finish if you so desire. You can also change it in many ways if you like. I will suggest some possibilities for modifications and extension before the chapter is over.

As usual I will assume that you have read and studied the relevant sections of the Microsoft BASIC Interpreter manual before you continue reading this chapter. In this case, you should read the discussion of the CALL command as well as Appendix F: Access to Macintosh ROM Routines.

THE PROGRAM LISTING

This program listing contains the symbol ✣ . It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✣ ; for example, if the listing shows

```
IF XO < X THEN     ✣
    inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN  inside = 0 : RETURN
```

Program Listing 12-1
Quickdraw Trainer

```
'-----------------------------------------------------------------
' Quickdraw Trainer
'
' Written by
' Richard C. Vile, Jr.
' March 3, 1985
'-----------------------------------------------------------------
DEFINT a-z

DIM pat(4), ppat(4), bpat(4), re(3), scr(3)
DIM polygon(256)
DIM hlp.title$(20)
DIM hlp.rn(20), hlp.len(20)

GOSUB defaults
GOSUB get.help.data
GOSUB makemenus
ON ERROR GOTO check.data
ON MENU GOSUB program.central
MENU ON

WHILE -1
  '--------------------------
  ' As usual - sit and wait for
  ' the real action to happen
  '--------------------------
WEND
END

program.central:
  MENU STOP
  menunumber = MENU(0)
  menuitem   = MENU(1)

  ON menunumber GOSUB graph, ops, pens, options, patterns, text, help
  MENU ON
  MENU
  RETURN

graph:
  IF (op <> erase.it AND menuitem <> 7) THEN CLS
  IF op = erase.it THEN GOSUB blackout
  ON menuitem GOSUB square, round, oval, arc, poly, lines, prt.text
```

```
  RETURN

square:
  FOR rpt = 1 TO repeats
    FOR de = 1 TO delay : NEXT de
    IF always.clear AND op <> erase.it THEN CLS
    IF always.clear AND op = erase.it THEN GOSUB blackout
    IF draw.rand THEN GOSUB rand.rect ELSE GOSUB drag.rect
    IF op = frame THEN CALL FRAMERECT (VARPTR (re(0)))
    IF op = paint THEN CALL PAINTRECT (VARPTR (re(0)))
    IF op = erase.it THEN CALL ERASERECT (VARPTR (re(0)))
    IF op = invert THEN CALL INVERTRECT (VARPTR (re(0)))
    IF op = fill THEN CALL FILLRECT (VARPTR (re(0)),VARPTR (pat(0)))
  NEXT rpt
  RETURN

round:
  FOR rpt = 1 TO repeats
    FOR de = 1 TO delay : NEXT de
    IF always.clear AND op <> erase.it THEN CLS
    IF always.clear AND op = erase.it THEN GOSUB blackout
    IF draw.rand THEN GOSUB rand.rect ELSE GOSUB drag.rect
    IF op = frame THEN +
      CALL FRAMEROUNDRECT (VARPTR (re(0)), ovalwidth, ovalheight)
    IF op = paint THEN +
      CALL PAINTROUNDRECT (VARPTR (re(0)), ovalwidth, ovalheight)
    IF op = erase.it THEN +
      CALL ERASEROUNDRECT (VARPTR (re(0)), ovalwidth, ovalheight)
    IF op = invert THEN +
      CALL INVERTROUNDRECT (VARPTR (re(0)), ovalwidth, ovalheight)
    IF op = fill THEN CALL +
      FILLROUNDRECT (VARPTR (re(0)), ovalwidth, ovalheight, VARPTR (pat(0)))
  NEXT rpt
  RETURN

oval:
  FOR rpt = 1 TO repeats
    FOR de = 1 TO delay : NEXT de
    IF always.clear AND op <> erase.it THEN CLS
    IF always.clear AND op = erase.it THEN GOSUB blackout
    IF draw.rand THEN GOSUB rand.rect ELSE GOSUB drag.rect
    IF op = frame THEN CALL FRAMEOVAL (VARPTR (re(0)))
    IF op = paint THEN CALL PAINTOVAL (VARPTR (re(0)))
    IF op = erase.it THEN CALL ERASEOVAL (VARPTR (re(0)))
    IF op = invert THEN CALL INVERTOVAL (VARPTR (re(0)))
    IF op = fill THEN CALL FILLOVAL (VARPTR (re(0)),VARPTR (pat(0)))
```

```
  NEXT rpt
  RETURN

arc:
  FOR rpt = 1 TO repeats
    FOR de = 1 TO delay : NEXT de
    IF always.clear AND op <> erase.it THEN CLS
    IF always.clear AND op = erase.it THEN GOSUB blackout
    IF draw.rand THEN GOSUB rand.rect ELSE GOSUB drag.rect
    IF op = frame THEN CALL FRAMEARC (VARPTR (re(0)), stangle, arcangle)
    IF op = paint THEN CALL PAINTARC (VARPTR (re(0)), stangle, arcangle)
    IF op = erase.it THEN CALL ERASEARC (VARPTR (re(0)), stangle, arcangle)
    IF op = invert THEN CALL INVERTARC (VARPTR (re(0)), stangle, arcangle)
    IF op = fill THEN  +
      CALL FILLARC (VARPTR (re(0)), stangle, arcangle, VARPTR (pat(0)))
  NEXT rpt
  RETURN

poly:
  FOR rpt = 1 TO repeats
    FOR de = 1 TO delay : NEXT de
    IF always.clear AND op <> erase.it THEN CLS
    IF always.clear AND op = erase.it THEN GOSUB blackout
    IF draw.rand THEN GOSUB rand.poly ELSE GOSUB draw.poly
    IF op = frame THEN CALL FRAMEPOLY (VARPTR (polygon(0)))
    IF op = paint THEN CALL PAINTPOLY (VARPTR (polygon(0)))
    IF op = erase.it THEN CALL ERASEPOLY (VARPTR (polygon(0)))
    IF op = invert THEN CALL INVERTPOLY (VARPTR (polygon(0)))
    IF op = fill THEN  +
      CALL FILLPOLY (VARPTR (polygon(0)), VARPTR (pat(0)))
  NEXT rpt
  RETURN

prt.text:
  IF always.clear THEN CLS
  FOR rpt = 1 TO repeats
    READ text$
    PRINT text$;" ";
    IF rpt MOD 5 = 0 THEN PRINT
  NEXT rpt
  RETURN

lines:
  FOR rpt = 1 TO repeats
    FOR de = 1 TO delay : NEXT de
    IF always.clear THEN CLS
    IF draw.rand THEN GOSUB rand.rect ELSE GOSUB drag.rect
```

```
    CALL MOVETO (re(1), re(0))
    CALL LINETO (re(3), re(2))
  NEXT rpt
  RETURN

ops:
  MENU operationmenu, op, 1
  op = menuitem
  MENU operationmenu, op, 2
  RETURN

pens:
  ON menuitem GOSUB mode, pnpat, bkpat, penh, penw, dpens
  RETURN

mode:
  RETURN

pnpat:
  FOR i = 0 TO 3
    ppat(i) = INT (RND * 32767)
  NEXT i
  CALL PENPAT (VARPTR (ppat(0)))
  RETURN

bkpat:
  FOR i = 0 TO 3
    bpat(i) = INT (RND * 255)
  NEXT i
  CALL BACKPAT (VARPTR (bpat(0)))
  RETURN

penh:
  CLS
  INPUT "Pen height "; ph
  MENU penmenu, 4, 1, "Pen height: " + STR$ (ph)
  CALL PENSIZE (pw, ph)
  RETURN

penw:
  CLS
  INPUT "Pen width "; pw
  MENU penmenu, 5, 1, "Pen width: " + STR$ (pw)
  CALL PENSIZE (pw, ph)
  RETURN

dpens:
  FOR i = 0 TO 3
```

```
    ppat(i) = -1
    bpat(i) = 0
  NEXT i
  pw = 1 : ph = 1
  CALL BACKPAT (VARPTR (bpat(0)))
  CALL PENPAT (VARPTR (ppat(0)))
  CALL PENSIZE (pw, ph)
  MENU penmenu, 4, 1, "Pen height: " + STR$(ph)
  MENU penmenu, 5, 1, "Pen width: " + STR$(pw)
  RETURN

options:
  ON menuitem GOSUB clr,rand,drg,nrpts,autocl,sarc,sde,svrt,nop,nop,nop,nop,nop,nop,quit
  RETURN

clr:
  CLS
  RETURN

rand:
  draw.rand = -1
  RETURN

drg:
  draw.rand = 0
  RETURN

nrpts:
  CLS
  INPUT "Repeat operation how many times "; repeats
  MENU optionmenu, 4, 1, "Number of repeats: " + STR$(repeats)
  RETURN

autocl:
  IF always.clear THEN always.clear = 0 : MENU optionmenu, 5, 1 : RETURN
  always.clear = -1
  MENU optionmenu, 5, 2
  RETURN

sarc:
  CLS
  INPUT "Arc angle "; arcangle
  MENU optionmenu, 6, 1, "Arc angle: " + STR$(arcangle)
  RETURN

sde:
  CLS
```

```
   INPUT "Delay between operations "; delay
   MENU optionmenu, 7, 1, "Delay: " + STR$ (delay)
   RETURN

svrt:
   CLS
   INPUT "Number of vertices in polygons "; poly.vertices
   MENU optionmenu, 8, 1, "Polygon vertices: " + STR$ (poly.vertices)
   RETURN

nop:
quit:
   MENU RESET
   END

patterns:
   RETURN

text:
   ON menuitem GOSUB txt.siz, txt.fnt
   IF menuitem > 2 THEN GOSUB txt.face
   RETURN

txt.siz:
   CLS
   INPUT "Text size: "; text.size
   MENU textmenu, 1, 1, "Size: " + STR$ (text.size)
   CALL TEXTSIZE (text.size)
   RETURN


txt.fnt:
   CLS
   INPUT "Font: (0 to 11)"; font
   CALL TEXTFONT (font)
   RETURN

txt.face:
   txt.pow = menuitem - 4
   IF txt.pow = -1 THEN face = 0 : CALL TEXTFACE (0) : GOSUB plain : RETURN
   txt.bit  = 2 ^ txt.pow
   IF face AND txt.bit THEN MENU textmenu, menuitem, 1 ELSE    +
      MENU textmenu, menuitem, 2
   IF face AND txt.bit THEN face = face - txt.bit ELSE face = face + txt.bit
   CALL TEXTFACE (face)
   MENU textmenu, 3, 1
   RETURN
```

```
plain:
  FOR item = 4 TO 10
    MENU textmenu, item, 1
  NEXT item
  MENU textmenu, 3, 2
  RETURN

help:
  hn = menuitem
  WINDOW CLOSE 1
  WINDOW 2, "", (50,30) - (455, 310), 2
  PRINT hlp.title$ (hn)
  PRINT
  FOR rn = hlp.rn (hn) TO hlp.rn (hn) + hlp.len (hn) - 1
    GET #2, rn
    PRINT msg$
  NEXT rn
  GOSUB ok
  WINDOW CLOSE 2
  WINDOW 1
  RETURN

rand.rect:
  re (0) = INT (RND * 200)
  re (1) = INT (RND * 300)
  re (2) = re (0) + INT (RND * (338 - re (0)))
  re (3) = re (1) + INT (RND * (508 - re(1)))
  RETURN


drag.rect:
  RETURN



rand.poly:
  polygon (0) = 2 * (7 + 2 * poly.vertices)
  polygon (1) = 0
  polygon (2) = 0
  polygon (3) = 350
  polygon (4) = 512
  FOR i = 0 TO poly.vertices - 1
    polygon (5 + 2*i) = INT (RND * 340)
    polygon (5 + 2*i + 1) = INT (RND * 500)
  NEXT i
  polygon (5 + 2 * poly.vertices) = polygon (5)
  polygon (6 + 2 * poly.vertices) = polygon (6)
  RETURN
```

```
draw.poly:
  RETURN

blackout:
  scr(0) = 0 : scr(1) = 0 : scr(2) = 350 : scr(3) = 512
  CALL PAINTRECT (VARPTR (scr(0)))
  RETURN

defaults:
  WINDOW 1, "", (0,20)-(510,340), 3
  draw.rand = -1
  frame = 1
  paint = 2
  erase.it = 3
  invert = 4
  fill = 5
  op = invert
  face = 0
  CALL TEXTFACE (0)
  repeats = 5
  poly.vertices = 3
  delay = 100
  stangle = 0
  arcangle = 90
  ovalheight = 20
  ovalwidth = 20
  ph = 1
  pw = 1
  CALL PENSIZE (pw, ph)
  text.size = 12
  CALL TEXTSIZE (text.size)
  RANDOMIZE TIMER
  FOR i = 0 TO 3
    pat(i) = INT (RND * 32767)
  NEXT i
  always.clear = 0
  RETURN

makemenus:
  FOR mn = 1 TO 5
    MENU mn, 0, 0, ""
  NEXT mn

  graphicmenu = 1

  MENU graphicmenu, 0, 1, "Graphics"
  MENU graphicmenu, 1, 1, "Plain Rectangle"
```

```
MENU graphicmenu, 2, 1, "Round Rectangle"
MENU graphicmenu, 3, 1, "Oval"
MENU graphicmenu, 4, 1, "Arc"
MENU graphicmenu, 5, 1, "Polygon"
MENU graphicmenu, 6, 1, "Line"
MENU graphicmenu, 7, 1, "Text"

operationmenu = 2
MENU operationmenu, 0, 1, "Operations"
MENU operationmenu, 1, 1, "Frame"
MENU operationmenu, 2, 1, "Paint"
MENU operationmenu, 3, 1, "Erase"
MENU operationmenu, 4, 2, "Invert"
MENU operationmenu, 5, 1, "Fill"

penmenu = 3
MENU penmenu, 0, 1, "Pens"
MENU penmenu, 1, 1, "Mode"
MENU penmenu, 2, 1, "Pen Pattern"
MENU penmenu, 3, 1, "Background Pattern"
MENU penmenu, 4, 1, "Height: " + STR$(ph)
MENU penmenu, 5, 1, "Width: " + STR$(pw)
MENU penmenu, 6, 1, "Restore Defaults"

optionmenu = 4
MENU optionmenu, 0, 1, "Options"
MENU optionmenu, 1, 1, "Clear Screen"
MENU optionmenu, 2, 1, "Random Drawing"
MENU optionmenu, 3, 1, "Drag Rectangle Drawing"
MENU optionmenu, 4, 1, "Number of Repeats: " + STR$(repeats)
MENU optionmenu, 5, 1, "Auto Clear"
MENU optionmenu, 6, 1, "Arc angle: " + STR$(arcangle)
MENU optionmenu, 7, 1, "Delay: " + STR$(delay)
MENU optionmenu, 8, 1, "Polygon vertices: " + STR$(poly.vertices)
FOR mi = 9 TO 14
  MENU optionmenu, mi, 0, "-"
NEXT mi
MENU optionmenu, 15, 1, "Quit"

patternmenu = 5
MENU patternmenu, 0, 1, "Patterns"
MENU patternmenu, 1, 1, "Show Patterns"
MENU patternmenu, 2, 1, "Add Pattern"
MENU patternmenu, 3, 1, "Pick Pattern"

textmenu = 6
MENU textmenu, 0, 1, "Text"
```

```
MENU textmenu, 1, 1, "Size" + STR$ (text.size)
MENU textmenu, 2, 1, "Font"
MENU textmenu, 3, 1, "Face: Plain"
MENU textmenu, 4, 1, "Face: Bold"
MENU textmenu, 5, 1, "Face: Italic"
MENU textmenu, 6, 1, "Face: Underlined"
MENU textmenu, 7, 1, "Face: Outlined"
MENU textmenu, 8, 1, "Face: Shadow"
MENU textmenu, 9, 1, "Face: Condensed"
MENU textmenu, 10, 1, "Face: Extended"

helpmenu = 7
MENU helpmenu, 0, 1, "Help"
FOR hm = 1 TO hlps
  MENU helpmenu, hm, 1, hlp.title$ (hm)
NEXT hm

RETURN
ok:
  BUTTON 1, 1, "Ok", (360, 240) - (390, 260)
  WHILE -1
    d = DIALOG (0)
    IF d = 1 THEN BUTTON CLOSE 1 : RETURN
  WEND

get.help.data:

  prg$ = "QDtrain"
  OPEN prg$ + ".hix" FOR INPUT AS #1
  OPEN "R", #2, prg$ + ".hd", 80
  FIELD #2, 80 AS msg$

  rn = 1
  WHILE NOT EOF (1)
    INPUT #1, hlp.title$ (rn)
    INPUT #1, hlp.rn (rn)
    INPUT #1, hlp.len (rn)
    rn = rn + 1
  WEND
  CLOSE #1
  hlps = rn - 1
  RETURN

check.data:
  IF ERR = 4 THEN RESTORE : RESUME
  ERROR ERR
  END
```

```
DATA The, splendor, falls, on, castle, walls
DATA and, snowy, summits, old, in, story
DATA The, long, light, shakes, across, the, lakes
DATA and, the, wild, cataract, leaps, in, glory!
DATA Blow, bugle, blow, set, the, wild, echoes, flying!
DATA Blow, bugle, answer, echoes, dying, dying, dying
DATA Dishevelled, and, in, tears, go, song, of, mine
DATA To, break, the, hardness, of, the, heart, of, man:
DATA Say, how, his, life, began
DATA From, dust, and, in, that, dust, doth, sink, supine:
DATA Yet, say, the, unerring, spirit, of, grief, shall, guide
DATA His, soul, being, purified
DATA To, seek, its, Maker, at, the, heavenly, shrine
DATA Beware, the, Jabberwock, my, son!
DATA The, jaws, that, bite, the, claws, that, catch!
DATA Beware, the, Jubjub, bird, and, shun
DATA The, frumious, Bandersnatch!
```

THE PROGRAM MENUS

Quickdraw Trainer has seven menus:

Graphics	**Patterns**
Operations	**Text**
Pens	**Help**
Options	

Figure 12-1 shows the general structure of the program. All action is initiated by choices from the **Graphics** menu. It allows the generation of the various geometric figures supported by the Quickdraw routines: rectangles, rounded rectangles, ovals, arcs, polygons, and lines. In addition, it will generate text.

What appears on the screen when a choice is made from the **Graphics** menu depends on what choices have previously been made from the other menus. For example, the **Operations** menu determines what type of graphics operation takes place on the specific figure chosen from the **Graphics** menu. For any of the general categories of geometric figure (except lines), you can FRAME, PAINT, ERASE, INVERT, or FILL.

The **Pens** menu gives choices that allow you to determine various pen characteristics. You can set the pen height and width. You can set the pen pattern, and you can set the background pattern. There is a choice to set the pen mode, although currently the program does not have a routine to handle this operation.

The **Options** menu has choices that allow you to control the way the graphic operations are performed. Some of the choices determine the value of certain parameters to ROM calls. Others define the method of generating specific figures to be used. A brief explanation of each choice follows:

Clear Screen —Clears the output window of all graphics or text.

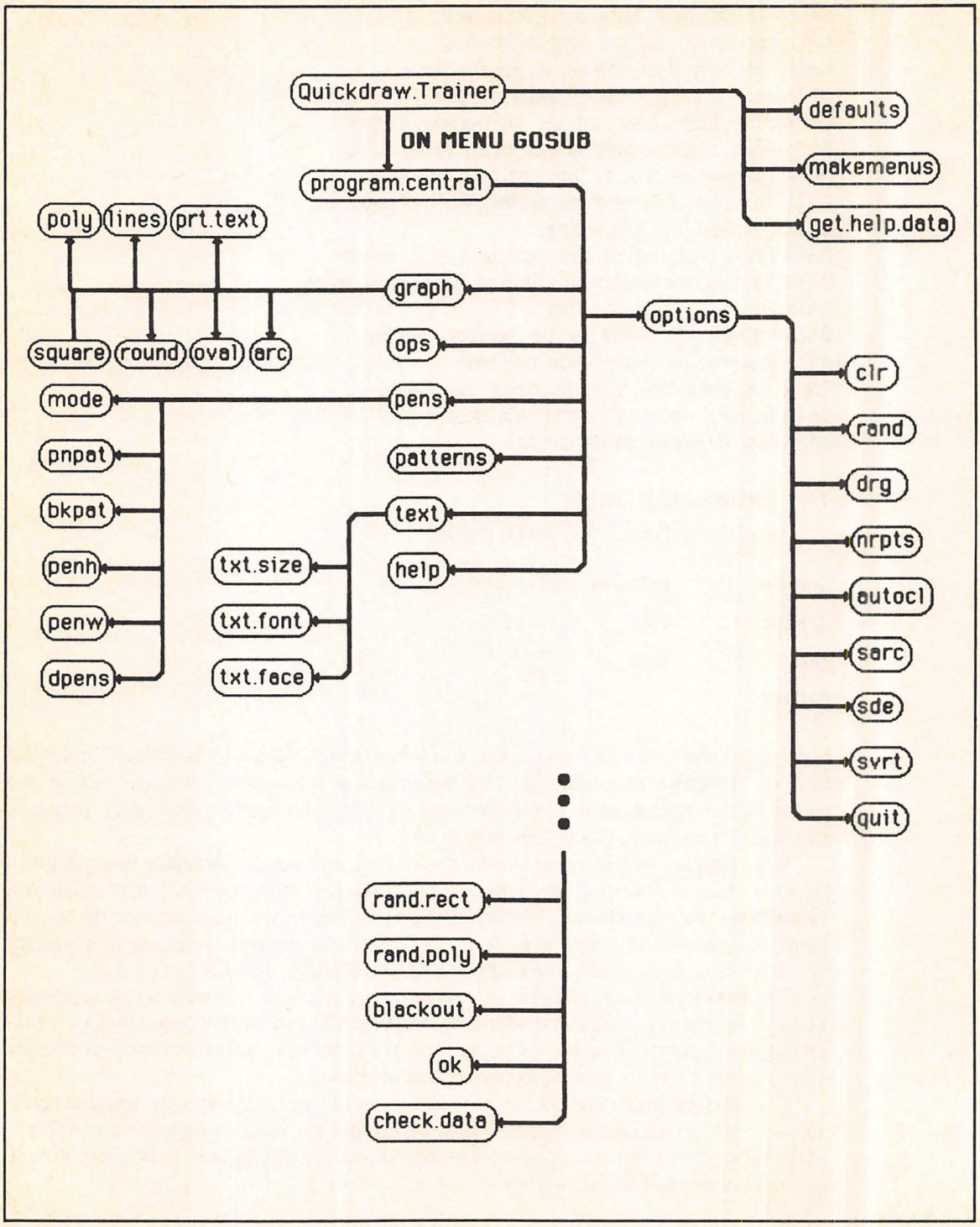


Fig. 12-1. The structure diagram for Quickdraw Trainer.

Random Drawing —Causes all graphic operations to generate random figures.

Drag Rectangle Drawing—Causes all graphic operations to follow user specified instructions for generating figures. The user specifies a rectangle containing each figure by dragging the rectangle open with the mouse.

Number of Repeats —Determines the number of times a given graphic operation will be performed when its choice is made from the **Graphics** menu.

Auto Clear —Determines whether or not the screen is cleared after each repetition of a given graphic operation.

Arc Angle —Indicates the value of the parameter passed to ROM routines for all ARC calls.

Delay —Specifies the upper limit of a delay loop (in BASIC) that occurs between each repetition of a given graphic operation.

Polygon Vertices —Indicates the number of vertices used in drawing each polygon in calls to POLY routines.

I have left several "blank" choices in the **Options** menu for you to add your own options. The last option is **Quit** which gets you out of the program when you feel your eyes getting blurry.

The **Pattern** menu has choices that allow you to manipulate a set of patterns. None of the choices has support in the current implementation. I leave it to you to implement this if you like.

The **Text** menu allows you to control characteristics of any text printed by the program (including prompts resulting from certain menu choices). The program makes no restrictions on the Size parameter, so you can really get carried away if you like. Similarly, it does not check the response to the font setting. The text face choices control the appearance of the printed text—you are probably already familiar with these ideas from MacWrite. The choices on the menu act as toggles. There is an exception to this. If you choose **Face: Plain** all other **Face:** choices become unchecked.

The Help menu contains explanations of various facets of the program to assist the user. The actual text is read from a random access file that is built by the Help File Generator program described later. You may use that program to add your own help messages—or you can refer to Appendix C for listings of the messages I used in my version.

WHAT THE PROGRAM DOES AND DOES NOT DO

The program as implemented is completely menu driven. As explained in the last section, ninety-nine percent of the action of the program is obtainable via the **Graphics** menu. The exceptions to this rule are the **Clear Screen** item in the **Options** menu, which immediately clears the output window, and the entire **Help** menu.

Quickdraw Trainer relies on calls to the Macintosh graphics ROM routines. It does this in a loop according to the value of the program parameter **repeats**. This value can be set with the **Number of Repeats** choice from the Options menu. The effects of the calls made depend on the settings of various other program parameters.

Most of those parameters can be controlled via menu selections. Here is a list of those variables and the menu choices that affect them. See the previous section for an explanation of the effects of each.

Variable	Menu - Item
ph	**Pens - Pen height**
pw	**Pens - Pen width**
ppat (i)	**Pens - Pen Pattern**
bpat (i)	**Pens - Background Pattern**
draw.rand	**Options - Random Drawing**
	Options - Drag Rectangle Drawing
repeats	**Options - Number of Repeats**
always.clear	**Options - Auto Clear**
arcangle	**Options - Arc Angle**
delay	**Options - Delay between operations**
poly.vertices	**Options - Number of vertices in polygons**
text.size	**Text - Text Size**
font	**Text - Font**
face	**Text - Face: ******

The other program parameters that affect the results of various graphic calls are the arrays re(i) and polygon(i). These contain information about the size and location of various graphic figures that are generated by Quickdraw calls. Every graphic call except the POLY calls has the array re passed to it: VARPTR (RE (0)). When random drawing is in effect, the values stored in re (0) – re (3) are determined by the subroutine rand.rect. It selects the values randomly, while guaranteeing that they will result in a rectangle that fits on the screen. The values of re(0) and re(1), which represent the y-coordinate of the top of the rectangle and the x coordinate of the left side of the rectangle respectively, are chosen so that the top left corner of the resulting rectangle will be at least 10 pixels away from the lower right hand corner of the output window.

ERASE calls have no visible effect on a white background. In order to demonstrate them, I made a slightly arbitrary choice. Whenever ERASE is the active choice from the **Operations** menu, the output window is first painted black. The effects of the ERASE operations are then visible in the output window. When the **Auto Clear** choice in the **Options** menu is checked, the window will be blacked out before each call in the repetition loop. When it is unchecked, the window will only be blacked out once, at the beginning of the repetition loop.

When the **Text** item from the **Graphics** menu is chosen, words of text are printed on the screen according to the settings of the current text parameters. The words are READ from DATA statements at the end of the program. When the program reaches the end of the data, the next READ command will cause an Out of Data error to occur. This error is trapped by the program and a RESTORE command is issued, followed by a RESUME command. This has the effect that the program reads the words again and again in a circular pattern.

Figures 12-2 through 12-6 show some of the typical output of the Quickdraw Trainer program.

SUGGESTIONS FOR MODIFYING AND EXTENDING THE PROGRAM

There are a number of parts of the program that remain to be implemented:

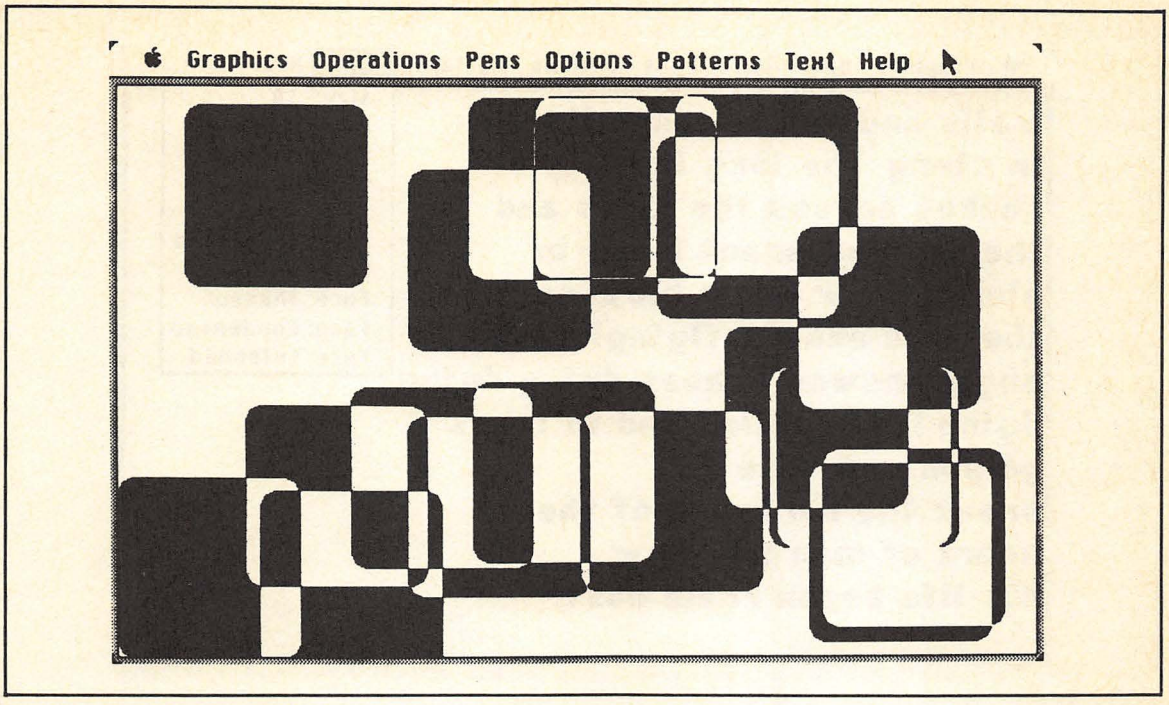


Fig. 12-2. Typical output of the Quickdraw Trainer prograr

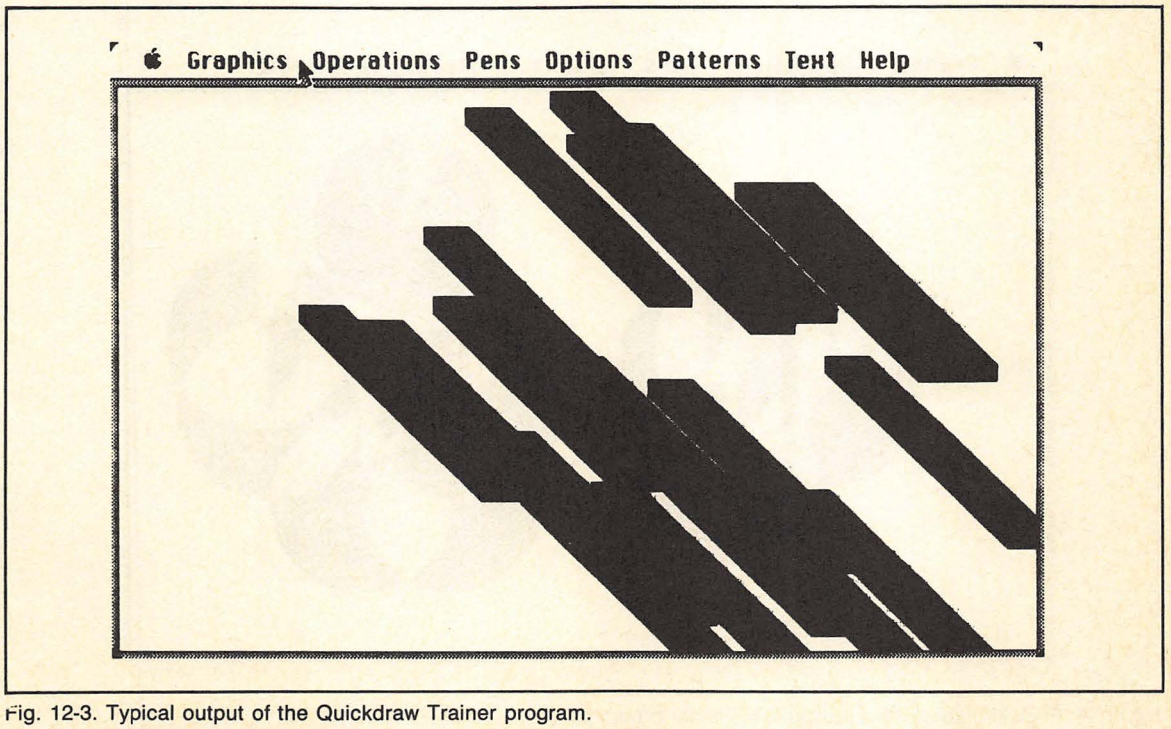


Fig. 12-3. Typical output of the Quickdraw Trainer program.

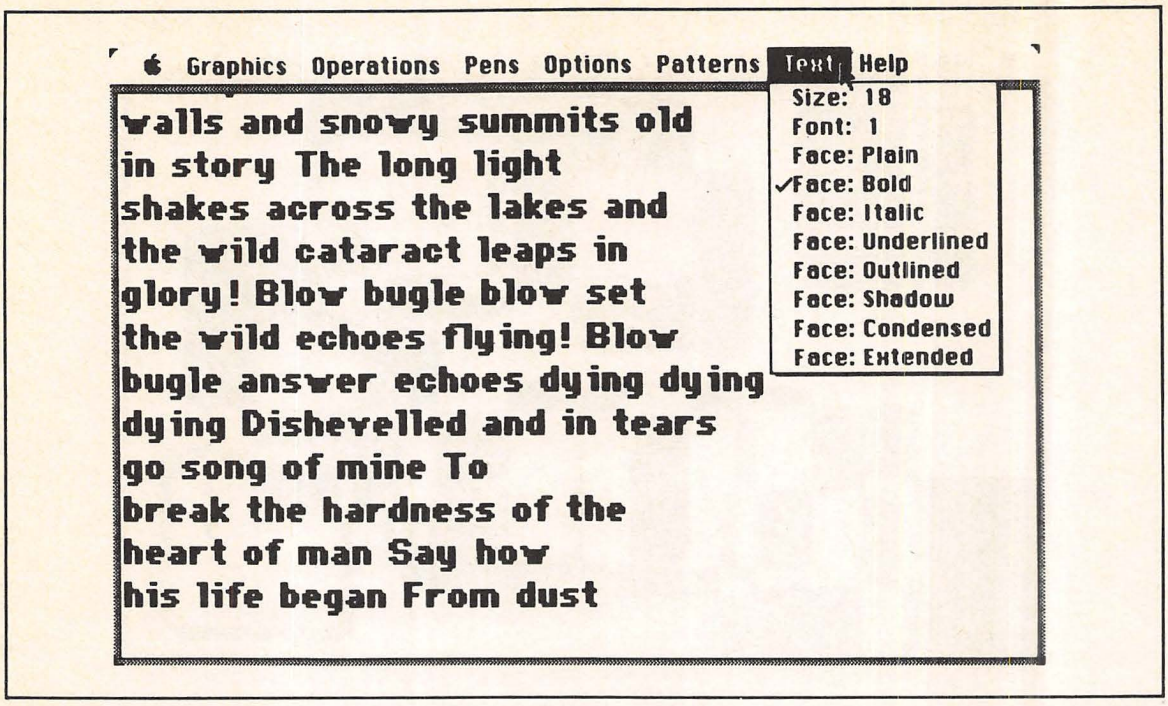


Fig. 12-4. Typical output of the Quickdraw Trainer program.

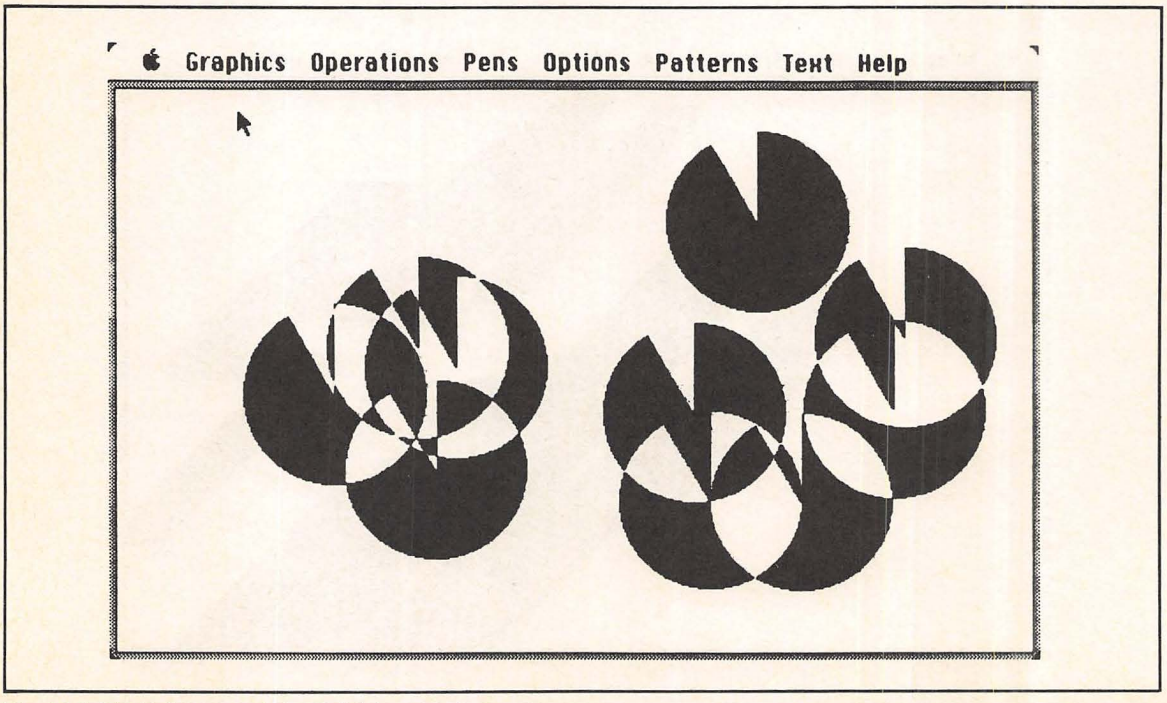


Fig. 12-5. Typical output of the Quickdraw Trainer program.

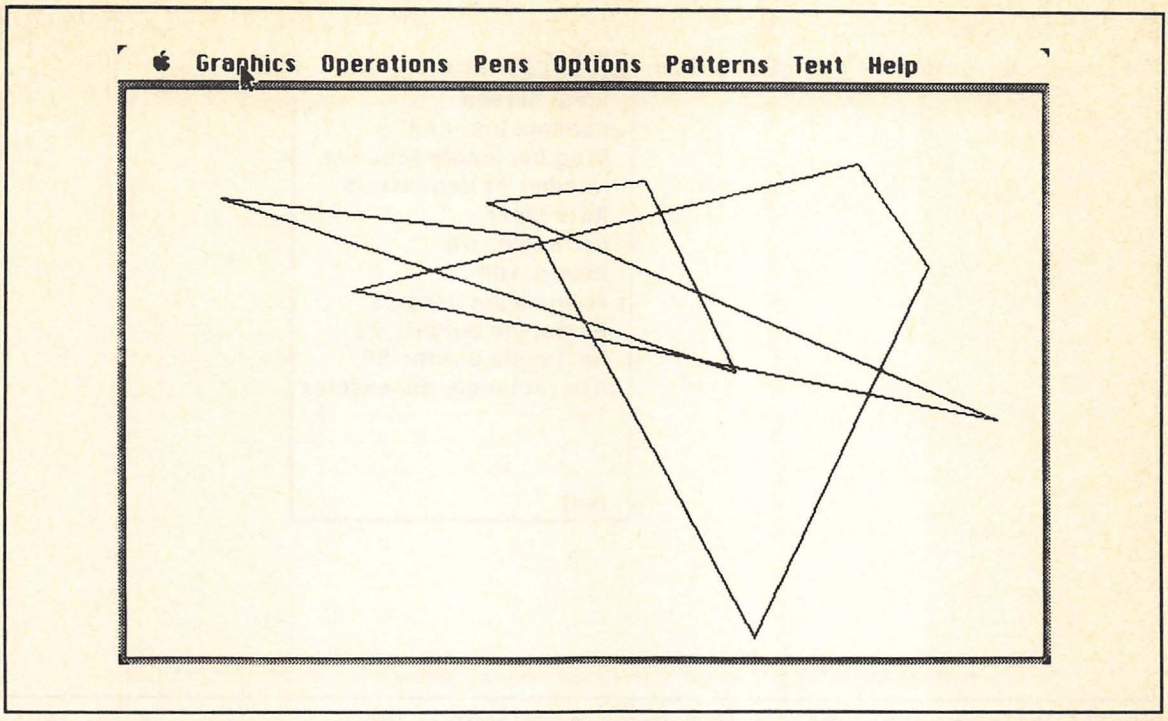


Fig. 12-6. Typical output of the Quickdraw Trainer program.

- The statements in the subroutine get.help.data will fail until you build your help data files: QDtrain.hix and QDtrain.hd. You can use the programs presented later to accomplish this. If you don't care about the help facility, simply remove the code referencing the **Help** menu:

  The statement GOSUB get.help.data
  The label "help" in the ON MENUNUMBER GOSUB statement
  The statements setting up the Help Menu
  The help: subroutine
  The get.help.data: subroutine

- None of the items in the **Patterns** menu work. The original idea was that a set of patterns would be maintained by the program for use in setting the Pen Pattern and Background Pattern. The items in the Pens menu that affect these patterns currently just generate random patterns. Adding a list of patterns to work with would require declaring an array to hold them (using DIM) and providing code to transfer them to the ppat and bpat arrays when appropriate. You could also incorporate radio buttons like the Pattern Builder that was illustrated earlier. The users could then make up their own patterns as well as use the predefined ones provided at startup.
- The Drag Rectangle Drawing capability does not work as yet. You could add this using the code provided in the Mouse Utilities sample program.
- You may wish to expand the **Random Drawing** capability in various ways. One obvious

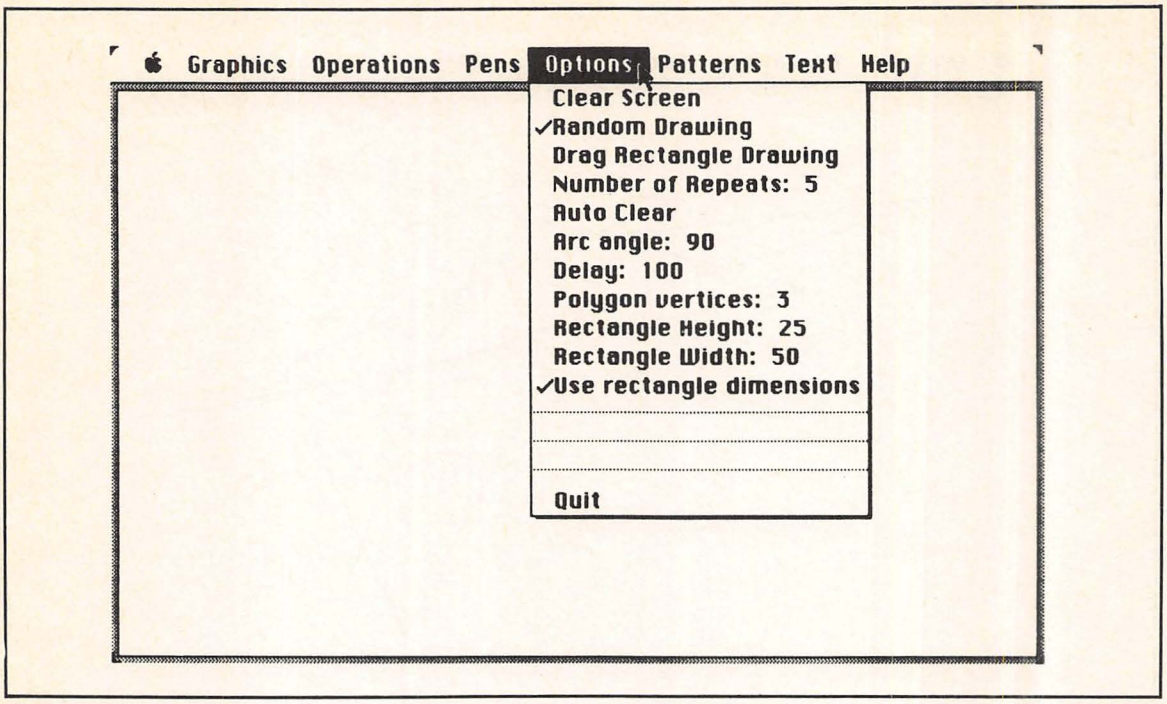


Fig. 12-7. The Options menu: some of the ways you can control Quickdraw Trainer.

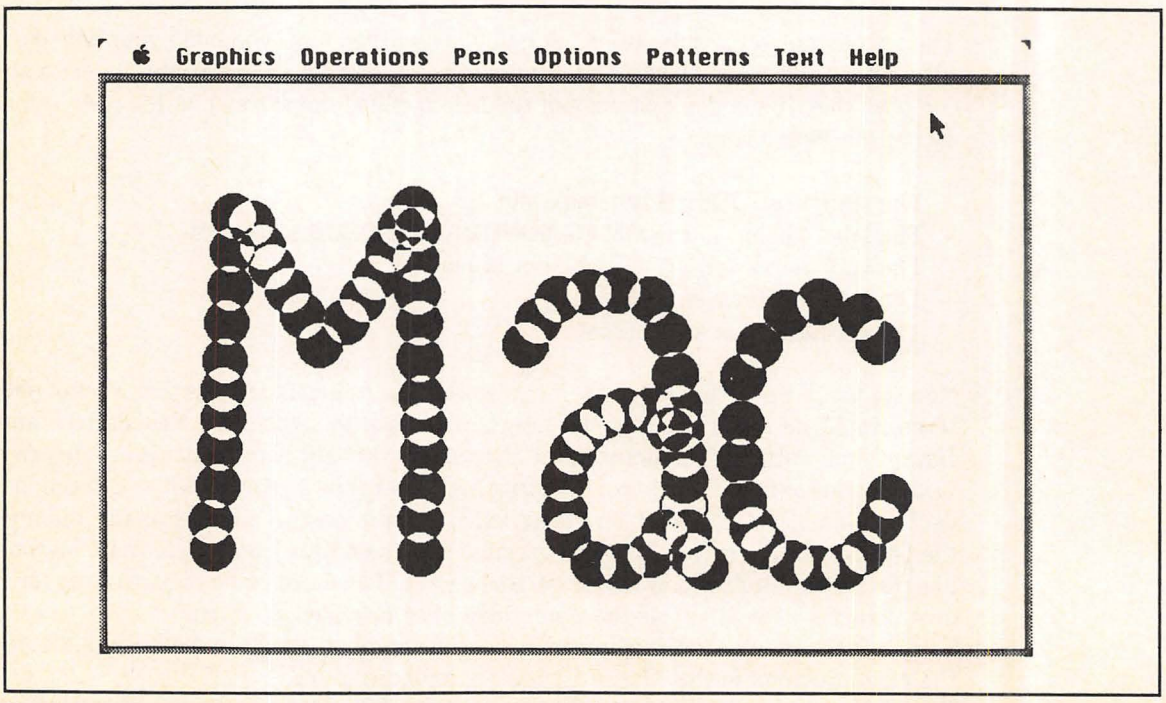


Fig. 12-8. Output of an extended version of the Quickdraw Trainer program.

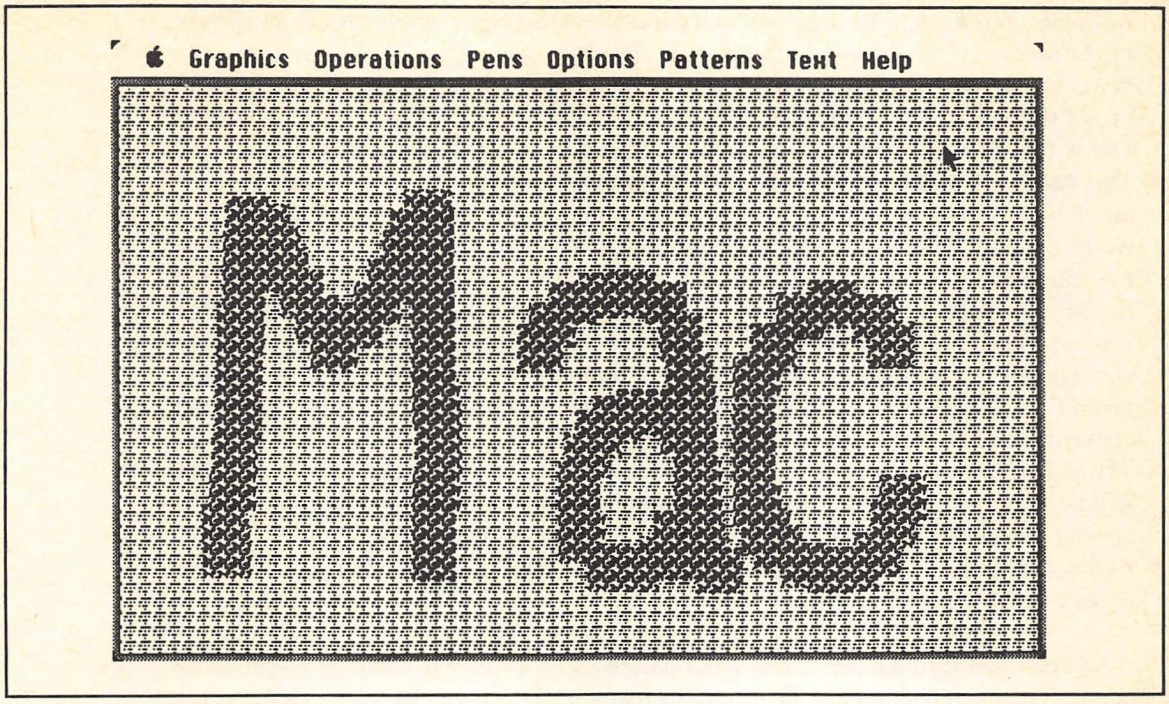


Fig. 12-9. Output of an extended version of the Quickdraw Trainer program.

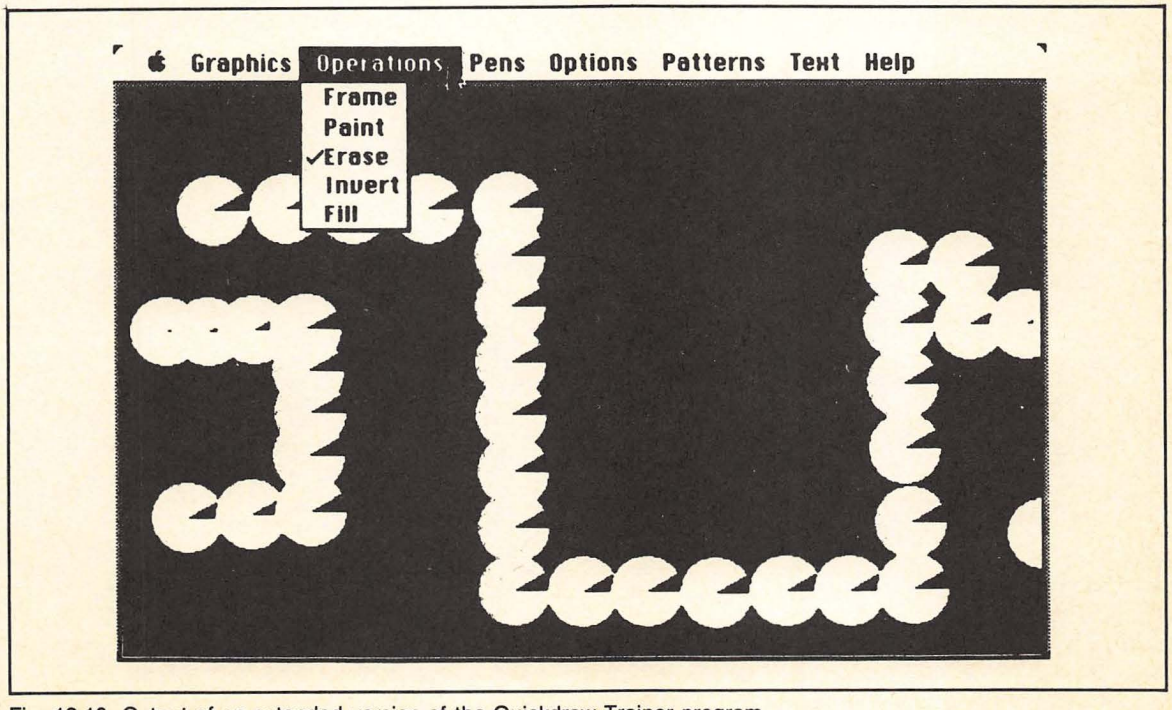


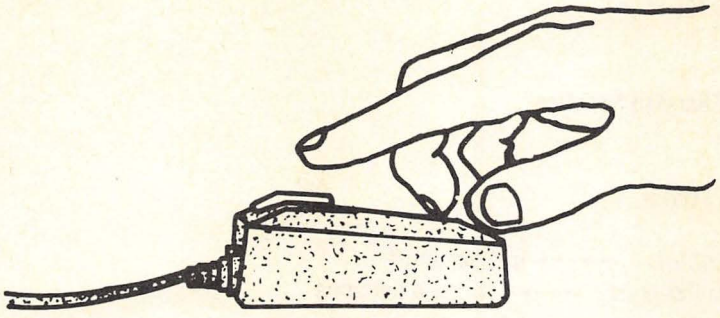
Fig. 12-10. Output of an extended version of the Quickdraw Trainer program.

possibility is the following: add program parameters that represent the width and height of the random rectangles. Add these to the **Options** menu (Fig. 12-7) so that they can be set by the user. Then add another item to the **Options** menu that determines whether or not the random rectangles generated by the program will all have the same height and width. Modify the rand.rect subroutine to take all of this into account.

- The **Random Drawing** and **Drag Rectangle Drawing** choices take up two menu "slots." Since they represent different values of the same program parameter (**draw.rand**), it would be more in the spirit of usual Macintosh practice to have them occupy the same menuitem slot. To make this happen in Quickdraw Trainer, you will have to change the MENU statements that set up the **Options** menu, as well as the subroutines that process the choices. Also, neither of these choices currently ever gets checked, so the user cannot tell simply by pulling down the **Options** menu which is in effect at any given time. While this is a minor defect, it is again contrary to the usual Macintosh conventions. See if you can remedy the situation.
- Try adding the ability to set and use a rectangle path. This is a sequence of points that will be taken as the corners of the successive rectangles. You can allow the user to specify the points in the rectangle path by clicking the mouse.
- Add a parameter to control the start angle for the ARC calls. Currently it is always equal to zero.

Figures 12-8 through 12-10 show the output of an extended version of Quickdraw Trainer. The extended version incorporates many of the suggestions for extension just described.

Chapter 13



Scribble and Video

These two programs are of the light entertainment variety. They show you some of the things that you can do with pens. The first program uses "fat pens" to create random scribblings. My ten year old son thinks they look like neckties—hence the name in the comment at the beginning of the program. Since not everyone may think the results suggest neckties, I decided to use the title Scribble in the chapter title. The second program also uses fat pens, this time to create a moving "test pattern" effect. Hence the designation Video.

Both of these programs were part of a collection of graphics demos I wrote in MS-BASIC 1.1. Each of them uses a CHAIN command to return to a master menu program called Demo-Menu (b). Several of the programs in the demo are also included in upcoming chapters.

Program Listings 13-1 to 13-2
Scribble and Video

Program Listing 13-1

```
'-----------------------------------------------------------------
' Random Neckties
'
' Written by Richard C. Vile, Jr.
' April 1984
' Revised December 28, 1984
'-----------------------------------------------------------------
```

```
DEFINT A-Z

DIM PATTERN (4)
DIM BP(4), PTS(16,4)
DIM CK(11)

WINDOW 1, "Random Neckties"

CLS
RANDOMIZE TIMER

GOSUB InitPenStuff : ' ----- Init pen stuff
GOSUB InitPenPatterns : ' ----- Init pen patterns

CALL BACKPAT (VARPTR (PATTERN(0)))
CLS
CALL MOVETO (350,110)
PRINT "To quit: ";
PUT (400,100), CK
PUT (416,100), CK

FOR I = 1 TO 5000
NEXT I


WHILE -1
  MS = MOUSE(0)
  IF ABS(MS) = 2 THEN CALL PENNORMAL : END

  FOR I = 0 TO 39
    X = INT(RND(1) * 450)
    Y = INT (RND(1) * 275)
    PH = INT (RND(1)*100)
      PW = INT (RND(1) * 100)
      CALL PENSIZE (PW, PH)
      CALL LINETO (X,Y)
'     RP = INT (RND(1) * 2)
'     IF (RP MOD 2) = 0 THEN
      GOSUB SetPattern
    NEXT I
  WEND
  END

  InitPenStuff:

    PATTERN(0) = 0
    PATTERN(1) = 0
```

```
  PATTERN(2) = 0
  PATTERN(3) = 0

  PH = 21
  PW = 21
  CALL PENSIZE (PW, PH)

  FOR II=0 TO 3
    BP(II) = PTS(0,II)
  NEXT II

  CALL PENPAT (VARPTR (BP(0)))
  RETURN

InitPenPatterns:

 FOR I=0 TO 15
   FOR J = 0 TO 3
     READ PTS(I,J)
   NEXT J
 NEXT I

 DATA -1,-1,-1,-1
 DATA 0,0,0,0
 DATA 31,-31,0,0
 DATA -1, 21834, -1, 21834
 DATA  240,240,240,240
 DATA 0, 4095, 4095, 0
 DATA 3855, -3855, 3855, -3855
 DATA 9509, -9509, 9509, -9509
 DATA 101,101,101,101
 DATA -256,0,256,0
 DATA 1000,2000,3000,4000
 DATA -1000,-2000,-3000,-4000
 DATA 747, -747, 747, -747
 DATA 0,255,0,255
 DATA 0,7,255,32767
 DATA 32,-32,128,-128

 FOR J = 0 TO 10
   READ CK(J)
 NEXT J

 DATA 16, 9, 12288, 14336, 7183, 3591, 1807, 925, 504, 240, 96
 RETURN

 SetPattern:
```

```
  RP = INT (RND(1) * 16)
  FOR J = 0 TO 4
    BP(J) = PTS(RP,J)
  NEXT J
  CALL PENPAT (VARPTR (BP(0)))
  RETURN
```

Program Listing 13-2

```
'-----------------------------------------------------------------
' Video Pattern
'
' Written by Richard C. Vile, Jr.
' April 1984
' Revised December 27, 1984
'-----------------------------------------------------------------
DEFINT A-Z

DIM PATTERN (4)
DIM BP(4), PTS(16,4)
DIM CK(11)

WINDOW 1, "Video Pattern"

RANDOMIZE TIMER
PW = 10 : PH = 10 : PD = 7

GOSUB InitPens
GOSUB InitPatterns

CALL BACKPAT (VARPTR (PATTERN(0)))
CLS
CALL MOVETO (350,110) : PRINT "To quit: ";
PUT (400,100),CK : PUT (416,100), CK

WHILE -1
  MS = MOUSE(0)
  IF ABS(MS) = 2 THEN CALL PENNORMAL : END

  FOR I = 0 TO 39
    IF INKEY$ = "" THEN 155
    WHILE INKEY$ = "" : WEND
155 CALL MOVETO (I*PD, 0)
    CALL LINETO (I*PD, 279)
    CALL MOVETO (279 - I*PD, 0)
    CALL LINETO (279 - I*PD, 279)
    CALL MOVETO (0, I*PD)
    CALL LINETO (279, I*PD)
    CALL MOVETO (0, 279 - I*PD)
```

```
      CALL LINETO ( 279, 279 - I*PD)
      RP = INT (RND(1) * 2)
      IF (RP MOD 2) = 0 THEN 175
      GOSUB NewPattern
175 NEXT I
  GOSUB NewPenSize
WEND

999 END

InitPens:
' **************
' * Pen stuff *
' **************
PATTERN (0) = 0 : PATTERN (1) = 0 : PATTERN(2) = 0 : PATTERN (3) = 0
CALL PENSIZE (PW, PH)
FOR II=0 TO 3 : BP(II) = PTS(0,II) : NEXT II
CALL PENPAT (VARPTR (BP (0)))
RETURN

InitPatterns:
' ***********************
' * Init pen patterns *
' ***********************
FOR I=0 TO 15
  FOR J = 0 TO 3
    READ PTS (I,J)
  NEXT J
NEXT I

DATA -1,-1,-1,-1
DATA 0,0,0,0
DATA 31,-31,0,0
DATA -1, 21834, -1, 21834
DATA  240,240,240,240
DATA 0, 4095, 4095, 0
DATA 3855, -3855, 3855, -3855
DATA 9509, -9509, 9509, -9509
DATA 101,101,101,101
DATA -256,0,256,0
DATA -1111, -2222, -3333, -4444
DATA -1000,-2000,-3000,-4000
DATA 747, -747, 747, -747
DATA 0,255,0,255
DATA 0,7,255,32767
DATA 32,-32,128,-128
FOR J = 0 TO 10 : READ CK(J) : NEXT J
```

```
DATA 16, 9, 12288, 14336, 7183, 3591, 1807, 925, 504, 240, 96
RETURN

NewPattern:
' ********************
' * Set pen pattern    *
' ********************

RP = INT (RND(1) * 16)

FOR J = 0 TO 4
  BP(J) = PTS(RP, J)
NEXT J

CALL PENPAT (VARPTR (BP(0)))
RETURN

NewPenSize:
' ************************************
' * New pen width, height, and delta *
' ************************************
PW = INT (RND(1)*50) : PH = INT (RND(1)*50)
CALL PENSIZE (PW, PH)
PD = INT (RND(1) * 20)
RETURN
```

PROGRAM TECHNIQUES

When I wrote both of these programs, I used MS-BASIC 1.1. That version of the interpreter had no MENU, WINDOW, or DIALOG commands. Since the programs are simple graphics demonstrations, I have not bothered to incorporate any menus or windows. The programs still bear some of the stylistic features of MS-BASIC 1.1.

The array CK is used to hold the bit image of a *click* icon. This check mark is used at the beginning of the program to indicate to the user that a double click must be used to quit the program. The condition:

```
ABS (MOUSE (0)) = 2
```

is used to detect the double click. This will usually work, but the user may have to try a couple of times. It is possible that a double click occurs, but the particular value of MOUSE (0) picked up by the program is not 2 or −2. This can happen if the user is unlucky and the call to MOUSE (0) occurs just between the first and second clicks. However, if you keep trying, eventually the program will "hear" you and quit. For more on double clicking, see the Programming Notes chapter.

The patterns used by each program are stored in a two dimensional array; PTS (P, N). Each pattern occupies four entries in the array. The patterns are read from DATA statements. No provision is made for the user to add or change patterns. Light entertain-

ment programs are usually simple like that!

The individual patterns are chosen at random in both programs and copied from the array PTS into the array BP for use in calls to PENPAT.

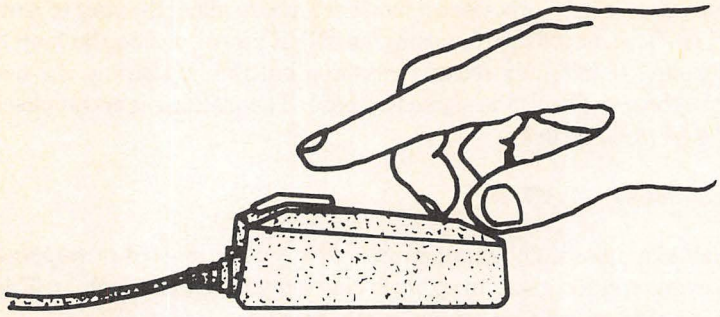
The Video program incorporates a pause feature to allow the user to freeze the screen if desired. This is accomplished by using an INKEY$ call inside the loop that produces the Video display. If INKEY$ returns anything but the null string, the user has struck a key at the keyboard. This is the signal to freeze. The freezing is accomplished by waiting for another key to be struck:

```
WHILE INKEY$ = "" : WEND
```

This is an example of detecting an event by *polling* instead of *trapping*. A program *polls* for an event when it uses an explicit checking statement such as IF INKEY$ < > " " . . . rather than relying on an automatic transfer of control.

Most of the line numbers of the original versions of these programs have been removed. It is an interesting and not very difficult challenge to recode the program to remove the remaining few line numbers.

Chapter 14



Rose Patterns

The program of this chapter draws rose patterns on the screen. Rose patterns may or may not look like roses; however they are all generated using the mathematical function:

$f(\Theta) = a * \sin(n * (\Theta + \Theta 0)) * scale$

where Θ (read "theta") represents an angle measured in radians. Mathematicians refer to the curves produced by these functions as *roses*. Students of calculus and analytic geometry often learn about polar coordinates by painstakingly graphing such curves.

The Rose Patterns program generates successive values of such functions using a spacing by the program parameter **stheta** (step value for theta). The value of f is the radius of the point corresponding to the value of Θ. The pair of numbers $(\Theta, f(\Theta))$ is referred to as the *polar coordinates* of the point on the rose curve corresponding to Θ. Rose Patterns transforms the polar coordinates into a pair of Macintosh screen coordinates in order to plot the point on the program's output window. The program draws several different copies of each individual rose, offset by a certain amount. The overlaying of the individual copies of the rose curves produces the pattern.

Rose Patterns was written using MS-BASIC 1.1. I have edited out most of the line numbers, but there are no menus and no dialog to speak of in the implementation. You may want to rewrite this program using dialog and menus in a more integral fashion.

There is one thing that the program does illustrate. It implements simple analog *controls*. An analog control is a screen element that the user can manipulate to change the value of something. An example familiar to you from your previous Macintosh experience is a scroll bar on a window. Such a control enables you to change your position in a docu-

ment or a mini-finder window. The controls provided by Rose Patterns enable you to change values of parameters using the mouse. The parameters affected combine to control the resulting roses drawn by the program.

THE PROGRAM LISTING

This program listing contains the symbol ✣ . It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✣ ; for example, if the listing shows

```
IF XO < X THEN    ✣
inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN  inside = 0 : RETURN
```

Program Listing 14-1
Rose Patterns

```
'-----------------------------------------------------------------
' Rose Patterns
'
' Written by Richard C. Vile, Jr.
' April 1984
'
' Revised December 28,1984
' Further revised April 1985
'-----------------------------------------------------------------

CLS
DEFINT C,R
DIM RT(12), RL(12), RB(12), RR(12)
DIM CK(20)

WINDOW 1, "", (4,24)-(508,338),3

TOP=100 : RE(0)=TOP
LEFT=250 : RE(1)=LEFT
BOTTOM=240 : RE(2)=BOTTOM
RIGHT=470 : RE(3)=RIGHT

GOSUB ClickIcon : ' Set up click icon

' ======================
```

```
' = Set up to draw figure =
' ======================

DEF FNROSE(THETA) = A* SIN (N*(THETA + THETAO)) * SCALE
GOSUB InitRoseParameters

WHILE 1 : ' +++++++++++++++ Main Program Loop +++++++++++++++
  THETAO=0
  MS = MOUSE (0)
  FOR D = 1 TO 300 : NEXT D
  FOR REPEAT = 1 TO R
    THETAO = THETAO + DTHETA
    RADIUS = FNROSE(0)
    XOLD = RADIUS * COS (0) + 50 : YOLD = RADIUS * SIN (0) + 30
    XSTART = XOLD : YSTART = YOLD
    FOR THETA = 0 TO (2 - (N MOD 2))*3.14159264# STEP STHETA
      RADIUS = FNROSE (THETA)
      X = RADIUS * COS (THETA) + 50 : Y = RADIUS * SIN (THETA) + 30
      LINE (XOLD + 200, YOLD + 130) - (X + 200, Y+ 130)
      XOLD = X : YOLD = Y
      MS = MOUSE (0) : IF MOUSE (0) <> 0 THEN 405
    NEXT THETA
    LINE (XOLD + 200, YOLD + 130) - (XSTART + 200, YSTART + 130)
  NEXT REPEAT
405 GOSUB Clicktocontinue
  CALL MOVETO (350,15) : PRINT "                                "
  CALL MOVETO (350,30) : PRINT "                                "
  GOSUB UpdateParameters : ' Update parameters
  CLS
WEND : ' +++++++++++++++ End of Main Loop +++++++++++++++
END

InitRoseParameters:
  A=10: N=4: DTHETA=.1: R = 2: SCALE = 12: STHETA = .05
  RETURN

Clicktocontinue:
  WHILE MOUSE (0) <> 0 : WEND
  ms = 1
  CALL MOVETO (350,15) : PRINT "More?";
  PUT (400,4),CK
  CALL MOVETO (350,30): PRINT "Quit? ";
  PUT (400,21), CK : PUT (420, 21), CK

  WHILE MOUSE (0) = 0 : WEND
  FOR de = 1 TO 750 : NEXT de
  ms1 = MOUSE (0)
```

```
  IF ABS (ms1) <= 1 THEN RETURN
  CHAIN "Demo-Menu (b)"
  END

DrawReportBox:
  CALL MOVETO (LEFT+5,TOP+15)
  CALL FRAMERECT (VARPTR(RE(0)))
  CALL ERASERECT (VARPTR(RE(0)))
  PRINT "To change a value, click its + or -";

  CALL MOVETO (LEFT + 5, TOP + 35)
  PRINT "a";PTAB (310);A;PTAB (430);"+";PTAB (455);"-"

  CALL MOVETO (LEFT+5,TOP + 55)
  PRINT "n";PTAB (310);N;PTAB (430);"+";PTAB (455);"-"
  CALL MOVETO (LEFT+5, TOP + 75)
  PRINT "delta";PTAB (310);
  PRINT USING "**.**";DTHETA;
  PRINT PTAB (430);"+";PTAB (455);"-"

  CALL MOVETO (LEFT + 5, TOP + 95)
  PRINT "repeat";PTAB (310);R;PTAB (430);"+";PTAB (455);"-"

  CALL MOVETO (LEFT + 5 , TOP + 115)
  PRINT "scale";PTAB (310);SCALE;PTAB (430);"+";PTAB (455);"-"

  CALL MOVETO (LEFT +5, TOP + 135)
  PRINT "step";PTAB (310);
  PRINT USING "**.**";STHETA;
  PRINT PTAB (430);"+";PTAB (455);"-"

  CALL FRAMERECT (VARPTR (RE(0)))
  PSET (RE(3), RE(0))
  FOR I=0 TO 3: RE(I) = RE(I)+1:NEXT I
  CALL FRAMERECT (VARPTR (RE(0)))
  FOR I = 0 TO 3 : RE(I) = RE(I) - 1 : NEXT I
  PSET (RE(1), RE(2))
  FOR DY = 20 TO 120 STEP 20
    LINE (LEFT, TOP +DY) - (RIGHT, TOP + DY)
    LINE (LEFT, TOP + DY + 1) - (RIGHT, TOP + DY + 1)
  NEXT DY
  LINE (LEFT + 50, TOP + 20) - (LEFT + 50, TOP + 140)
  LINE (LEFT + 51, TOP + 20) - (LEFT + 51, TOP + 140)
  LINE (LEFT + 170, TOP + 20) - (LEFT + 170, TOP + 140)
  LINE (LEFT + 171, TOP + 20) - (LEFT + 171, TOP + 140)
  LINE (LEFT + 195, TOP + 20) - (LEFT + 195, TOP + 140)
  LINE (LEFT + 196, TOP + 20) - (LEFT + 196, TOP + 140)
```

```
 RETURN

UpdateParameters:
 GOSUB DrawReportBox
 WHILE 1

    GOSUB GetMouseLocation
    GOSUB DetermineRectangle
    IF WHICH = 0 THEN RETURN
    ON WHICH GOSUB +
1600,1605,1610,1615,1620,1625,1630,1635,1640,1645,1650,1655

 WEND

1600 A = A + 1
    CALL MOVETO (310, TOP + 35)
    PRINT A;
    RETURN

1605 A = A - 1
    CALL MOVETO (310, TOP + 35)
    PRINT A;
    RETURN

1610 N = N + 1
    CALL MOVETO (310, TOP + 55)
    PRINT N;
    RETURN

1615 N = N - 1
    CALL MOVETO (310, TOP + 55)
    PRINT N;
    RETURN

1620 DTHETA = DTHETA + .01
    CALL MOVETO (310, TOP + 75)
    PRINT USING "##.##"; DTHETA;
    RETURN

1625 DTHETA = DTHETA - .01
    CALL MOVETO (310, TOP + 75)
    PRINT USING "##.##"; DTHETA;
    RETURN

1630 R = R + 1
    CALL MOVETO (310, TOP + 95)
    PRINT R;
```

```
    RETURN

1635 R = R - 1
    CALL MOVETO (310, TOP + 95)
    PRINT R;
    RETURN

1640 SCALE = SCALE + 1
    CALL MOVETO (310, TOP + 115)
    PRINT SCALE;
    RETURN

1645 SCALE = SCALE - 1
    CALL MOVETO (310, TOP + 115)
    PRINT SCALE;
    RETURN

1650 STHETA = STHETA + .01
    CALL MOVETO (310, TOP + 135)
    PRINT USING "##.##"; STHETA ;
    RETURN

1655 STHETA = STHETA - .01
    CALL MOVETO (310, TOP + 135)
    PRINT USING "##.##"; STHETA ;
    RETURN

GetMouseLocation:
  MS = MOUSE (0) : ' Clear mouse
  WHILE MOUSE(0) = 0 : WEND : ' Wait for click
  X = MOUSE (3) - 421 : Y = MOUSE (4) - 122 : ' Co-ordinates
  FOR de = 1 TO 35 : NEXT de
  RETURN

DetermineRectangle:
  COL = X \ 24
  ROW = Y \ 20
  IF (X < 0) OR (Y < 0) OR (ROW > 5) OR (COL > 1) THEN which = 0 : RETURN
  RETURN

ClickIcon:
  ck (0) = 16 : ck (1) = 9
  ck (2) = 48*256 : ck (3) = 56*256 : ck (4) = 28*256 + 15
  ck (5) = 14*256 + 7 : ck (6) = 7*256 + 15 : ck (7) = 3*256 + 157
  ck (8) = 256+248 : ck (9) = 240 : ck (10) = 96
  RETURN
```

PROGRAM BEHAVIOR

The program draws roses on the screen. The appearance of the rose pattern is affected by the following parameters:

Parameter	Effect
A	Determines size of patterns
N	Determines the number of "leaves" N odd → N leaves; N even → 2*N leaves
DELTA	The amount THETA0 varies between drawings
REPEAT	The number of times the pattern is drawn
SCALE	Determines size of patterns
STEP	The amount THETA varies between values of f

A wide variety of figures can be obtained by changing the combination of values used for the above parameters. As you will see shortly, each can be changed directly by the user. Figures 14-1 through 14-8 illustrate the variety that you can obtain. Each figure is labelled with the values of the above parameters used to obtain it.

USER CONTROL

The program can be interrupted while it is drawing by a single click of the mouse. When this happens, or when the program finishes drawing the current rose pattern, the

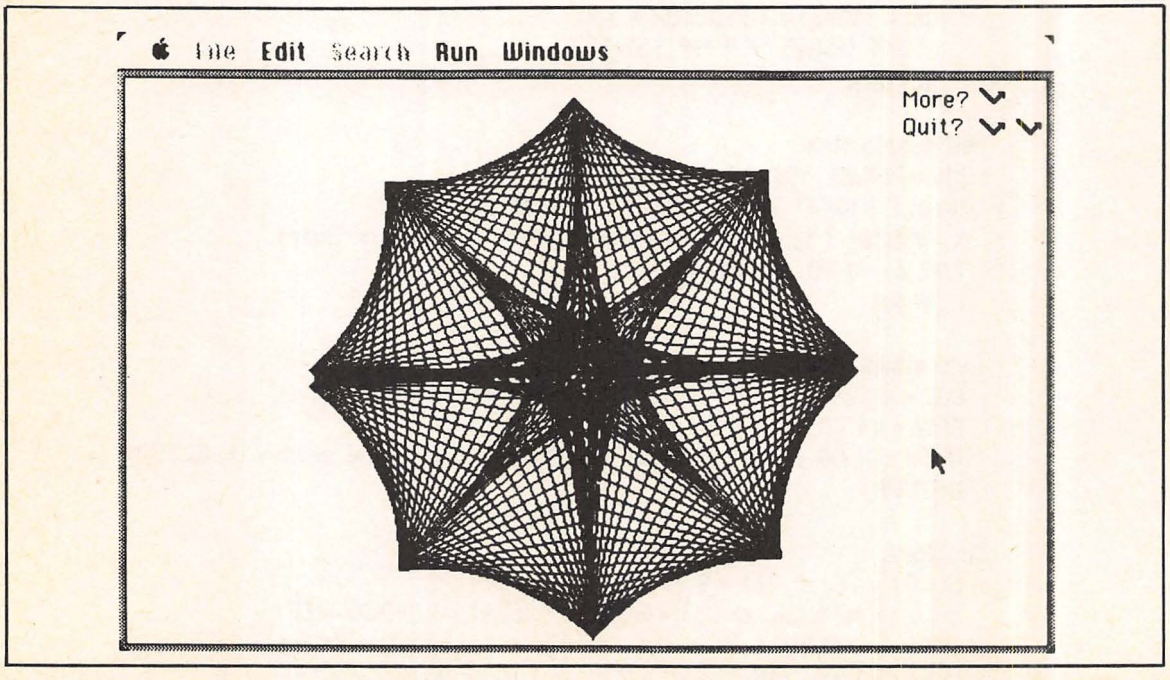


Fig. 14-1. Results of the Rose Patterns program using the parameters A = 15, N = 7, DELTA = 0.01, REPEAT = 90, SCALE = 11, and STEP = 0.77.

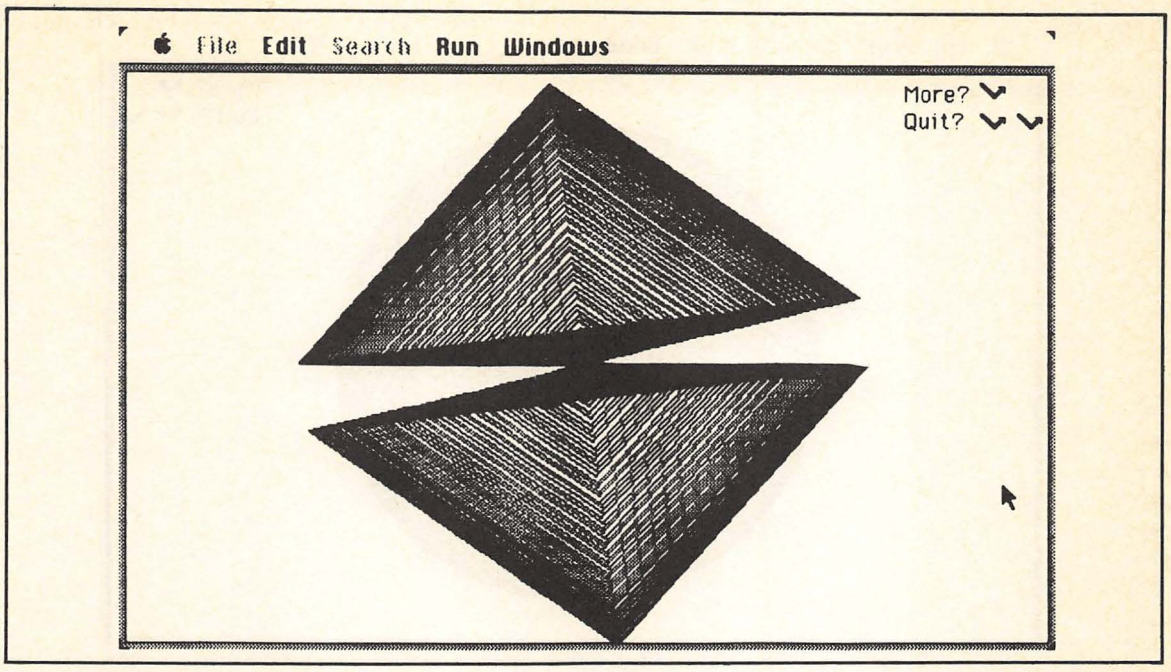


Fig. 14-2. Results of the Rose Patterns program using the parameters A = 26, N = 13, DELTA = 0.01, REPEAT = 148, SCALE = 11, and STEP = 1.45.

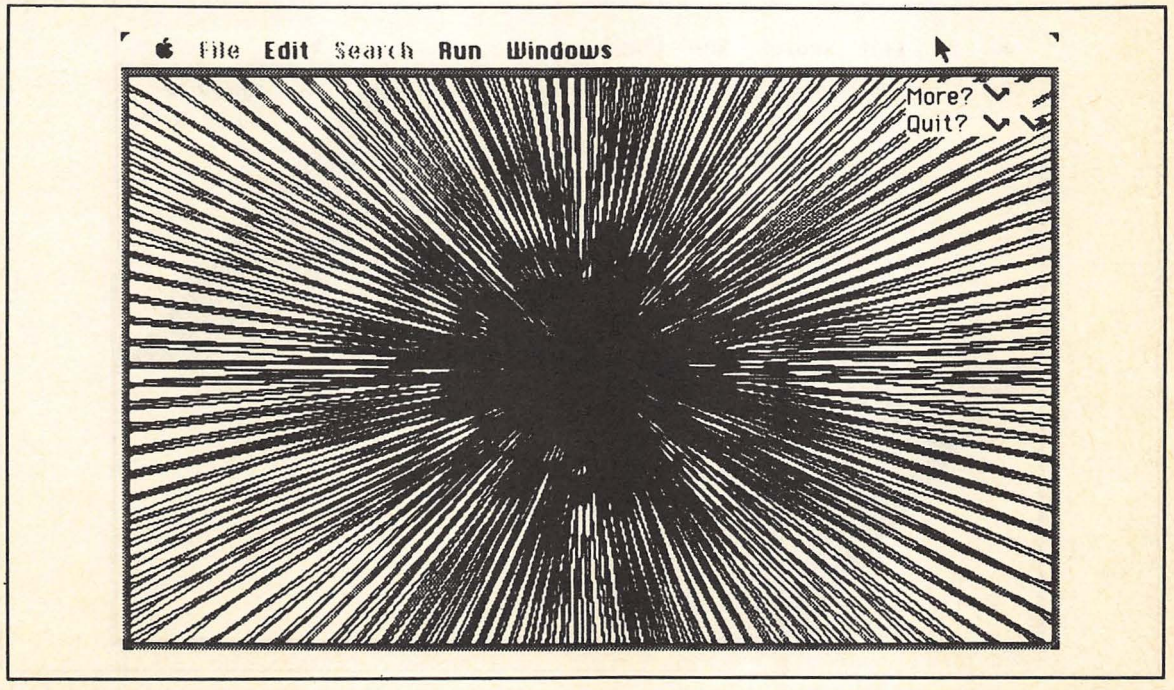


Fig. 14-3. Results of the Rose Patterns program using the parameters A = 20, N= 102, DELTA = 0.05, REPEAT = 5, SCALE = 20, and STEP = 0.05.

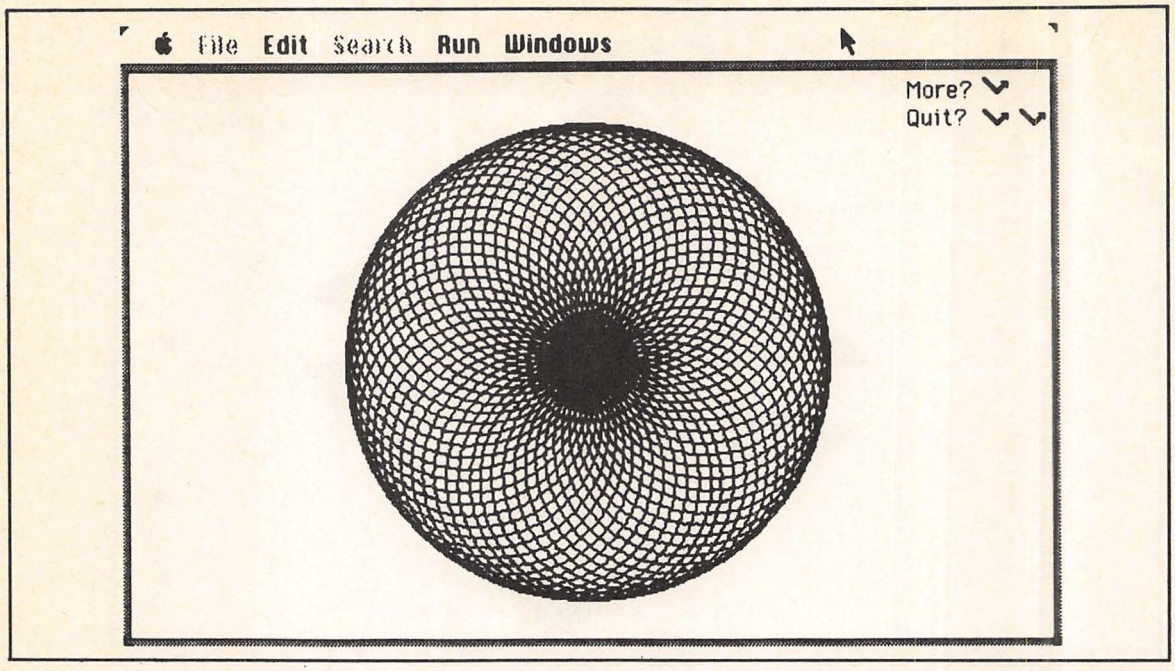


Fig. 14-4. Results of the Rose Patterns program using the parameters A = 10, N = 1, DELTA = 0.10, REPEAT = 63, SCALE = 12, and STEP = 0.01.

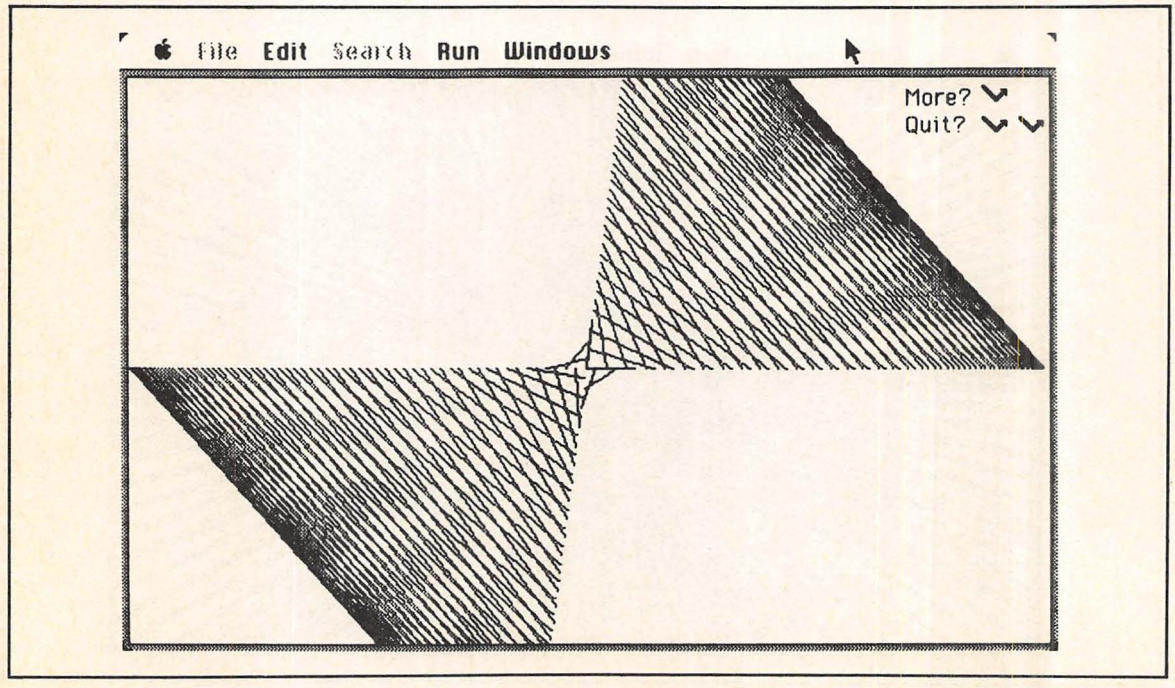


Fig. 14-5. Results of the Rose Patterns program using the parameters A = 25, N = 13, DELTA = 0.03, REPEAT = 148, SCALE = 10, and STEP = 1.70.

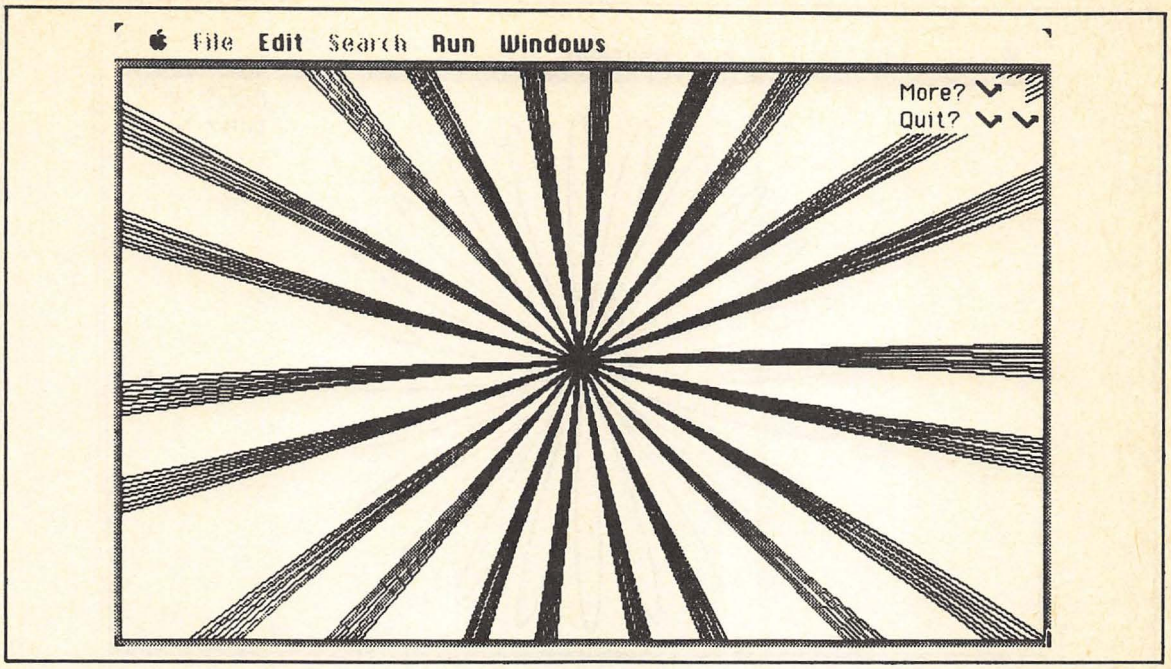


Fig. 14-6. Results of the Rose Patterns program using the parameters A = 50, N = 11, DELTA = 0.01, REPEAT = 8, SCALE = 12, and STEP = 0.01.

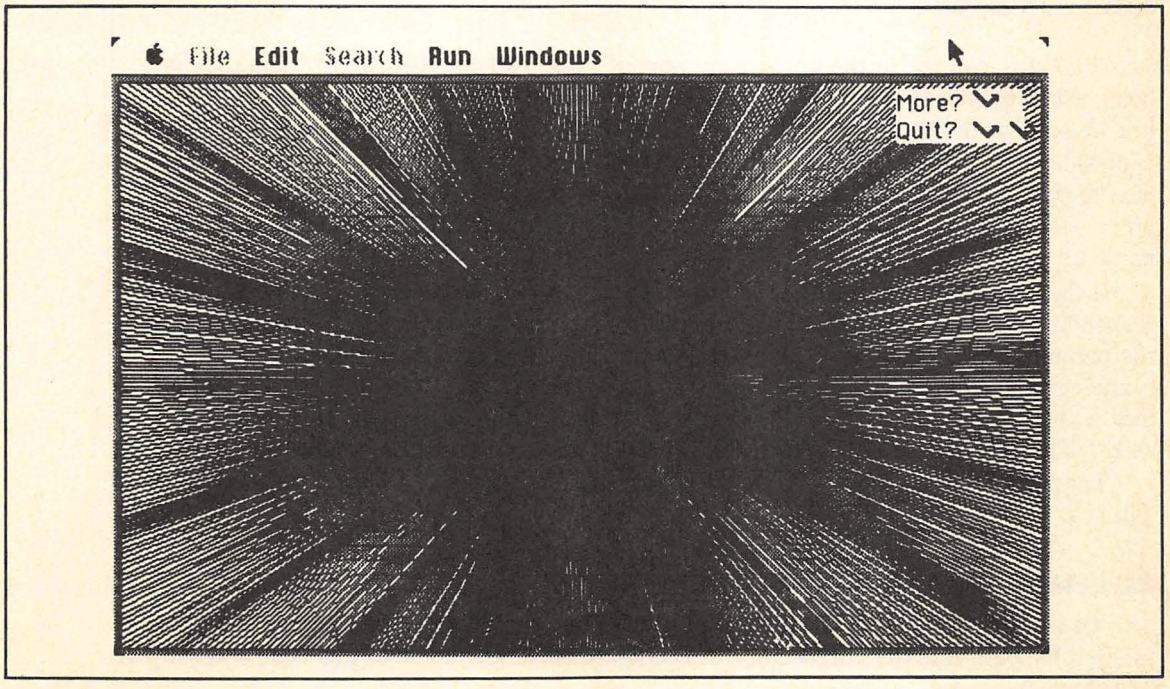


Fig. 14-7. Results of the Rose Patterns program using the parameters A = 200, N = 11, DELTA = 0.01, REPEAT = 34, SCALE = 12, and STEP = 0.01.

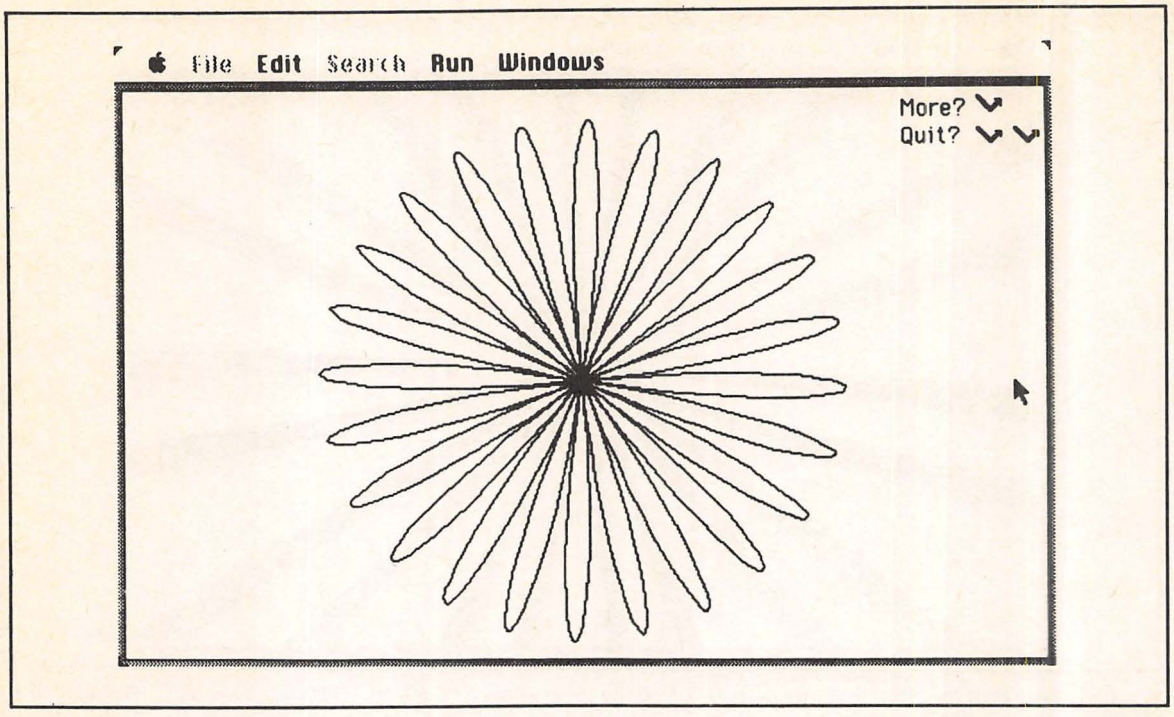


Fig. 14-8. Results of the Rose Patterns program using the parameters A = 12, N = 12, DELTA = 0.10, REPEAT = 1, SCALE = 12, and STEP = 0.01.

two lines that you see in the upper right part of Figs. 14-1 to 14-8 appear. The user is being asked to click once to get more patterns or to click twice to quit the program. Figure 14-9 shows the result of responding with a single click. A grid is drawn on the right hand side of the screen. It displays the current settings of each of the control parameters. On the right side of the grid appear two columns. One column contains + signs and the other – signs. The top row of the grid contains the instruction: To change a value, click its + or –.

If the user clicks the mouse inside any of the squares containing a + sign, the corresponding program parameter will be increased. If the mouse button is clicked and held, the parameter will continue to increase as long as the button is held down. This action is analogous to clicking on one of the arrows at the end of a scroll bar. Likewise, clicking inside a square with a – sign will cause the corresponding parameter to decrease as long as the button is held down. These controls enable you to set any value you like.

Take some time to play with the program. See what radically different shapes you can get.

IMPLEMENTING THE PROGRAM CONTROLS

Of course, all this doesn't happen by itself. There is BASIC code (and lots of it!) to make it happen. It is the purpose of this section to delve into some of this code and see how it works.

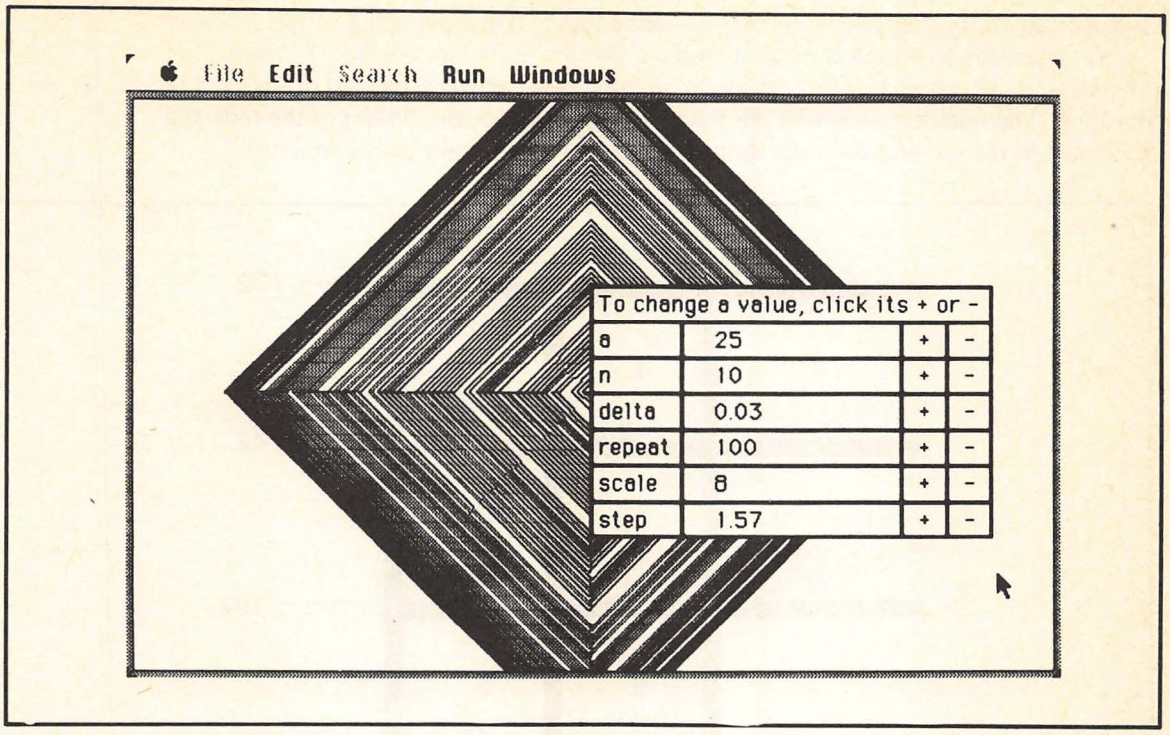


Fig. 14-9. Rose Patterns parameters controls.

Drawing the Grid

The grid is drawn using simple MS-BASIC graphics commands along with Quickdraw calls. The subroutine DrawReportBox accomplishes this task. If you study it, you should have no difficulty seeing what it does. It is not difficult code, but it is tedious.

One point should be specially noted. The text in the grid is drawn *before* the outline of the grid. This is necessary since if you reverse the process, the text includes surrounding white space that destroys parts of the lines of the grid. If you made the boxes of the grid fatter, then you could avoid this destructive white space. But it is just as easy to draw the text first; then the problem goes away. And you don't have to experiment to find out how big you have to make the boxes to avoid the destructive effects.

Where Was the Mouse Clicked?

The bigger problem that must be solved is this: how do you determine which rectangle the mouse was clicked inside. A simple approach could be taken similar to that used in the Mouse Utility in Chapter 10. The code given there to determine whether or not a mouse click was inside a specific rectangle could be repeated for each of the rectangles of the program. This approach is perfectly valid, but there is a slight problem. You will be forced to search the rectangles in some order. This means that the last rectangle in the list will take longer than the first to locate and verify. When the number of rectangles involved gets at all large, this can slow the program down unacceptably. I leave

it to you to verify that twelve rectangles slows things down too much.

A faster way to search through a larger collection of rectangles must be found. Let us examine the screen layout a bit closer. Figure 14-10 shows a blowup of the area in question. The numbers shown in the figure are the x and y coordinates of the tops and left sides of the rectangles in question. Here are two important points to notice:

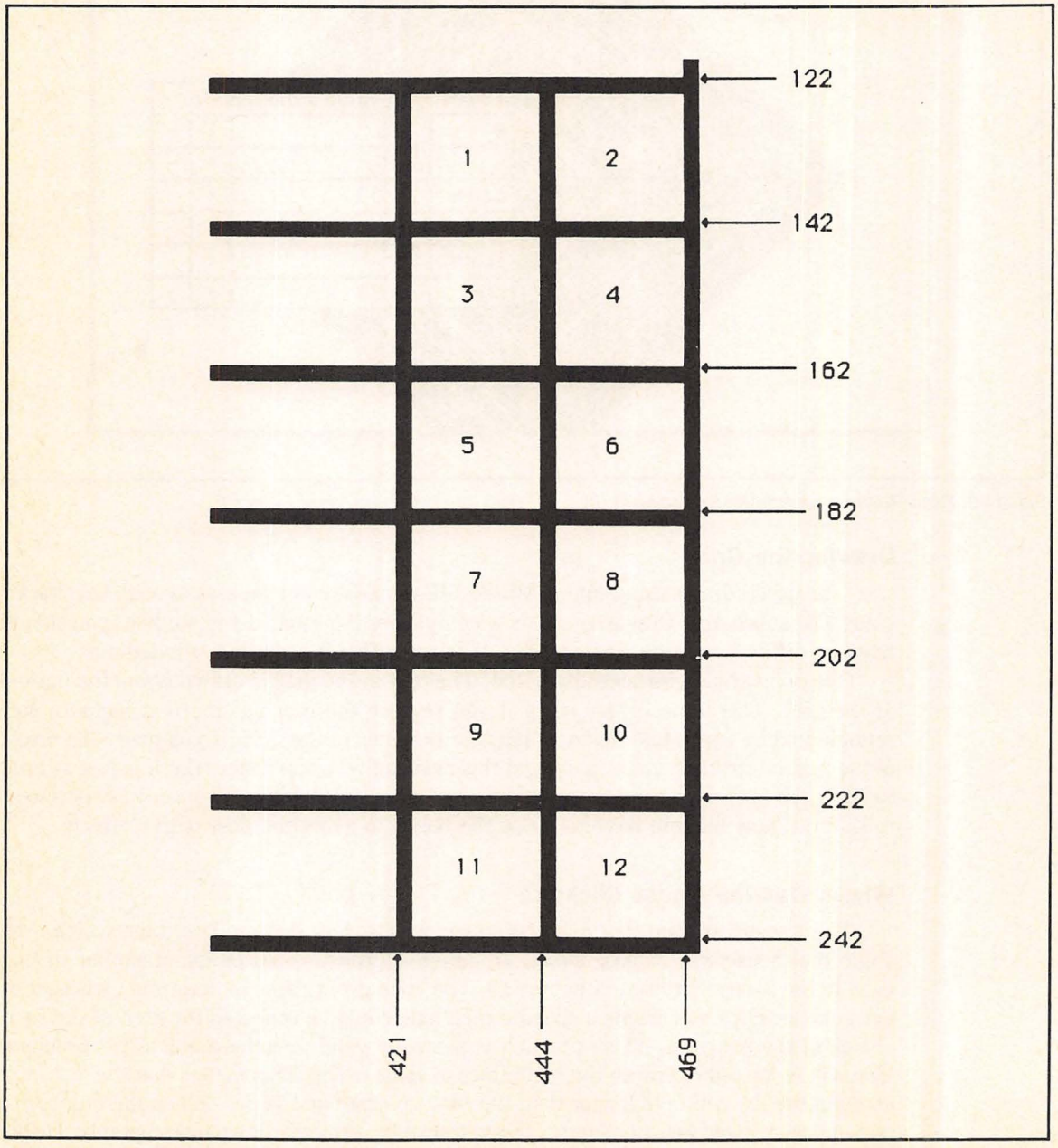


Fig. 14-10. The rectangles to be detected by the Rose Patterns program.

- All the rectangles are the same size.
- The rectangles are adjacent to one another—no space intervenes.

Because of these two features of the layout, it is possible to use simple arithmetic to locate the mouse click.

The first step is to adjust the coordinate values returned by the mouse. The coordinates of the upper left hand corner of the block of rectangles is subtracted from the mouse coordinates:

```
X = MOUSE (3) - 421 : Y = MOUSE (4) - 122
```

Next *modular arithmetic* is used to determine which row and column the rectangle that was clicked lies in. The block of rectangles in the grid forms an array of six rows and two columns. The calculations are as follows:

```
ROW = Y \ 20  COL = X \ 24
```

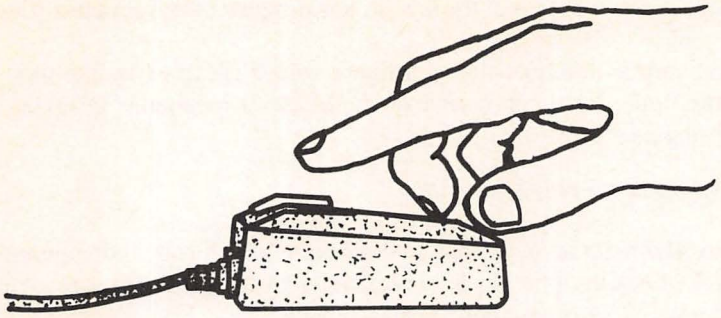
These expressions give the quotients of **Y** and **X** divided by 20 and 24, respectively, with any remainders being ignored. This provides exactly the information of interest. Try a few examples if you don't believe me. All that remains is to convert the ROW and COL numbers into a rectangle number in the numbering scheme indicated in Fig. 14-10. This is done with the formula:

```
WHICH = 2 * ROW + COL + 1
```

This same technique can be used in any situation where there are a number of equally sized rectangles that form a block of rows and columns. The numbers 20 and 24 should be replaced by the height and width of the rectangles in the new situation.

Once the number of the clicked rectangle has been computed, the rest is easy. The **ON which GOSUB** statement invokes a subroutine that adjusts the appropriate parameter and updates the screen to reflect the new value.

Chapter 15



Ups 'N Downs

Ups 'N Downs was written in MS-BASIC 1.1. It is a puzzle or game that turns out to be equivalent to a number of other puzzles. In particular, its solution is mathematically equivalent (or *isomorphic*) to the chinese-rings puzzle.

The game is played with a row of eight tabs. Initially they are all shown below a horizontal line in the middle of the screen. Placing the cursor on the tabs and clicking the mouse button makes them flip above or below the line . . . Sometimes! Most of the time a given tab will not budge and when you click on it the Mac will just beep at you. The trick of the puzzle is to figure out which tabs move and when. Then you must make the correct sequence of moves to get all the tabs above the line. It is challenging and fun.

THE PROGRAM LISTING

This program listing contains the symbol ✣. It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✣ ; for example, if the listing shows

```
IF X0 < X THEN    ✣
   inside = 0 : RETURN
```

then your program should look like:

```
IF X0 < X THEN  inside = 0 : RETURN
```

Program Listing 15-1
Ups 'N Downs

```
'-----------------------------------------------------------------
' Ups 'N Downs - Combinatorial Puzzle Game
'
' Written by
' Richard C. Vile, Jr.
' April 1984
'-----------------------------------------------------------------

1 CLS
5 DEFINT B,L,R,T,C
10 DIM TOP(15), LEF(15), RIG(15), BOT(15)
15 DIM SLOT(7), CK(11)

WINDOW 1, "Ups 'n Downs Puzzle"

60 GOSUB 5000 : REM ***** Introduction *****
65 CLS
100 GOSUB 2000 : REM ***** Set up rectangles *****
105 GOSUB 2100 : REM ***** Draw all rectangles in down position *****
110 CALL MOVETO (175,18) : PRINT "To quit, click here"; :  +
    PUT(300,9), CK
140 GOSUB 2900 : REM ***** Get mouse co-ordinates *****
150 GOSUB 3000 : REM ***** Check which tab chosen *****
155 IF I = 16 THEN PRINT CHR$(7); : GOTO 140
160 GOSUB 2200 : REM ***** Check if ok to move tab *****
162 IF DEBUG THEN PRINT OK
165 IF OK THEN GOSUB 2300 ELSE PRINT CHR$(7); :  +
    REM ***** Move tab *****
175 IF OK THEN MOVES = MOVES + 1 : CALL MOVETO (10, 10) :  +
    PRINT "Moves = ";MOVES ;
180 GOSUB 2400 : REM ***** Check for puzzle solved *****
185 IF NOT DONE THEN 140
190 GOSUB 2500 : REM ***** Congratulations *****
195 PRINT "Click to go again. Double click to quit"
196 MS = MOUSE (0) : WHILE MOUSE (0) = 0 : WEND
197 FOR D = 1 TO 300 : NEXT D
198 MS = MOUSE (0) : IF ABS (MS) = 1 THEN 105
199 IF ABS (MS) = 2 THEN STOP ELSE GOTO 195
2000 FOR I = 1 TO 8
2005 TOP (I-1) = 50 : BOT (I-1) = 145
2010 LEF (I-1) = 50 * I : RIG (I-1) = 50 * I + 40
2015 NEXT I
```

```
2020 FOR I = 8 TO 15
2025 TOP (I) = 155 : BOT (I) = 250
2030 LEF (I) = 50 * (I- 7) : RIG (I) = 50 * (I - 7) + 40
2035 NEXT I
2040 FOR J=0 TO 10 : READ CK(J) : NEXT J
2042 DATA 16, 9, 12288, 14336, 7183, 3591, 1807
2043 DATA 925, 504, 240, 96
2099 RETURN
2100 CLS
2105 LINE (0,150) - (490,150)
2106 LINE (0,151) - (490, 151)
2110 FOR J = 8 TO 15
2120 GOSUB 3300
2125 MOVES = 0
2130 NEXT J
2135 FOR I = 0 TO 7 : SLOT (I) = 1 : NEXT I
2149 RETURN
2200 REM --------------------------
2201 REM - Check validity of move -
2202 REM --------------------------
2205 IF I > 15 THEN PRINT CHR$ (7); : RETURN
2210 OK = 0 : TB = I MOD 8
2215 IF TB = 0 THEN OK = 1 : RETURN : ♣
     REM ***** First tab always ok *****
2220 IF TB <> 1 THEN 2250
2230 OK = (SLOT (0) = 1) : RETURN
2250 IF (SLOT (TB-1) <> 1) THEN RETURN
2255 FOR TT = 0 TO TB - 2
2260 IF SLOT (TT) = 1 THEN RETURN
2265 NEXT TT
2270 OK = 1 : RETURN
2300 REM --------------------
2301 REM - Update display -
2302 REM --------------------
2310 TB = I MOD 8 : IF SLOT (TB) = 1 THEN MVFROM = TB + 8 ♣
     ELSE MVFROM = TB
2315 IF SLOT (TB) = 0 THEN MVTO = TB + 8 ELSE MVTO = TB
2320 J = MVFROM : GOSUB 3100
2325 J = MVTO : GOSUB 3300
2330 SLOT (TB) = 1 - SLOT (TB)
2349 RETURN
2400 REM ----------------------------
2401 REM - Check if solution reached -
2402 REM ----------------------------
2410 SOLVED = 0
2415 FOR S = 0 TO 7 : SOLVED = SOLVED + SLOT (S) : NEXT S
2420 DONE = (SOLVED = 0)
```

```
2449 RETURN
2500 REM ----------------------------------------
2501 REM - Tell player the solution has been reached  -
2502 REM ----------------------------------------
2510 CALL MOVETO (10,165):  +
     PRINT "Congratulations! You reached the solution"
2515 PRINT "It only took you ";MOVES " Moves.
2549 RETURN
2900 MS = MOUSE (0) : REM ***** Clear mouse *****
2905 WHILE MOUSE (0) = 0 : WEND : REM ***** Wait for click *****
2906 WHILE MOUSE (0) <> 0 : WEND : ' Wait for unclick
2910 X = MOUSE (3) : Y = MOUSE (4)
2919 RETURN
3000 REM ------------------------------------------------
3001 REM - Determine which (if any) rectangle was clicked -
3002 REM ------------------------------------------------------
3005 FOR I = 0 TO 15
3010 CHECK = (X > LEF(I)) + (X < RIG(I)) + (Y > TOP(I)) + (Y < BOT(I))
3015 IF CHECK = -4 THEN RETURN
3020 NEXT I
3025 IF (X > 300) AND (X < 310) AND (Y > 9) AND (Y < 20) THEN END
3049 RETURN
3100 REM --------------------
3101 REM - Erase rectangle  -
3102 REM --------------------
3105 RE (0) = TOP (J) : RE( 1) = LEF (J)
3106 RE (2) = BOT (J) : RE(3) = RIG(J)
3110 CALL ERASERECT (VARPTR (RE(0)))
3115 IF DEBUG THEN PRINT "Erasing #";J
3119 RETURN
3200 REM ---------------------------------
3201 REM - Draw a bold empty rectangle     -
3202 REM ---------------------------------
3205 IF DEBUG THEN PRINT "Drawing #";J
3210 RE(0) = TOP (J) : RE( 1) = LEF (J)
3215  RE(2) = BOT (J) : RE(3) = RIG(J)
3220 CALL FRAMERECT (VARPTR (RE(0)))
3225 RE(0) = RE (0) + 1 : RE( 1) = RE( 1) + 1
3230 RE (2) = RE(2) - 1 : RE(3) = RE(3) - 1
3235 CALL FRAMERECT (VARPTR (RE(0)))
3249 RETURN
3300 REM --------------------------------
3301 REM - Draw a solid black rectangle  -
3302 REM --------------------------------
3310 RE(0) = TOP (J) : RE( 1) = LEF (J)
3315  RE(2) = BOT (J) : RE(3) = RIG(J)
3320 CALL PAINTRECT (VARPTR (RE(0)))
```

```
3349 RETURN
5000 REM ----------------
5001 REM - Introduction  -
5002 REM ----------------
5005 CLS : CALL TEXTFACE (1)
6010 PRINT "Welcome to UP's and DOWN's" : PRINT
6012 CALL TEXTFACE (0)
6015 PRINT "This is a game of combinations. The aim is to move a row"
6018 PRINT "of eight tabs from below a horizontal line to above the line"
6020 PRINT "At any given point in the game, some but not all of the tabs"
6022 PRINT "are free to move. Click on a given tab to see if it can move."
6024 PRINT "If it is free, it will change position from up to down or"
6026 PRINT "vice-versa. If it is not, the Mac will beep. "
6028 PRINT "As tabs change position, the tabs which are free to move"
6030 PRINT  +
    "change as well. Remember - the goal is get ALL the tabs above"
6032 PRINT "the line. Good Luck." : PRINT
6090 PRINT "Click to continue"
6092 MS = MOUSE (0) : WHILE MOUSE (0) = 0 : WEND
6099 RETURN
```

PROGRAM DETAILS

Ups 'N Downs faces some of the same problems that Rose Patterns does. It solves them differently, however. The main problem is detecting when the mouse is clicked inside certain screen rectangles.

Figure 15-1 is a snapshot of the programs's puzzle picture. I have superimposed on the display the outlines of eight more rectangles (above the horizontal line), which the program also detects with mouse clicks. The top eight rectangles are numbered (left to right) as 0 to 7. The bottom eight rectangles are numbered (again, left to right) as 8 to 15. Values for the top, left, bottom, and right of each of these 16 rectangles are stored in the arrays: TOP, LEF, RIG, and BOT, respectively. The values are placed in the arrays by the subroutine beginning at line 2000.

MOUSE CLICKS INSIDE RECTANGLES REVISITED

In the Mouse Utilities section I discussed the problem of detecting whether or not a pair of coordinates were inside a rectangle. The coordinates were assumed to be obtained by calls to the MOUSE function after a click was detected. Here we are faced with a similar problem. The difference is that there are many rectangles. One solution to this problem was discussed in the last section. Here is an alternative solution. The solution used in Rose Patterns is convenient when the rectangles are uniform in size and contiguous on the screen. That is what makes the modular arithmetic work. In Ups 'N Downs, the rectangles are uniform in size but there are gaps of space between them that make the modular arithmetic approach less convenient.

In order for a point specified by coordinates (x, y) to be located inside a rectangle specified by TOP (I), LEF (I), BOT (I), RIG (I), four conditions must be simultaneously satisfied (and at the same time!):

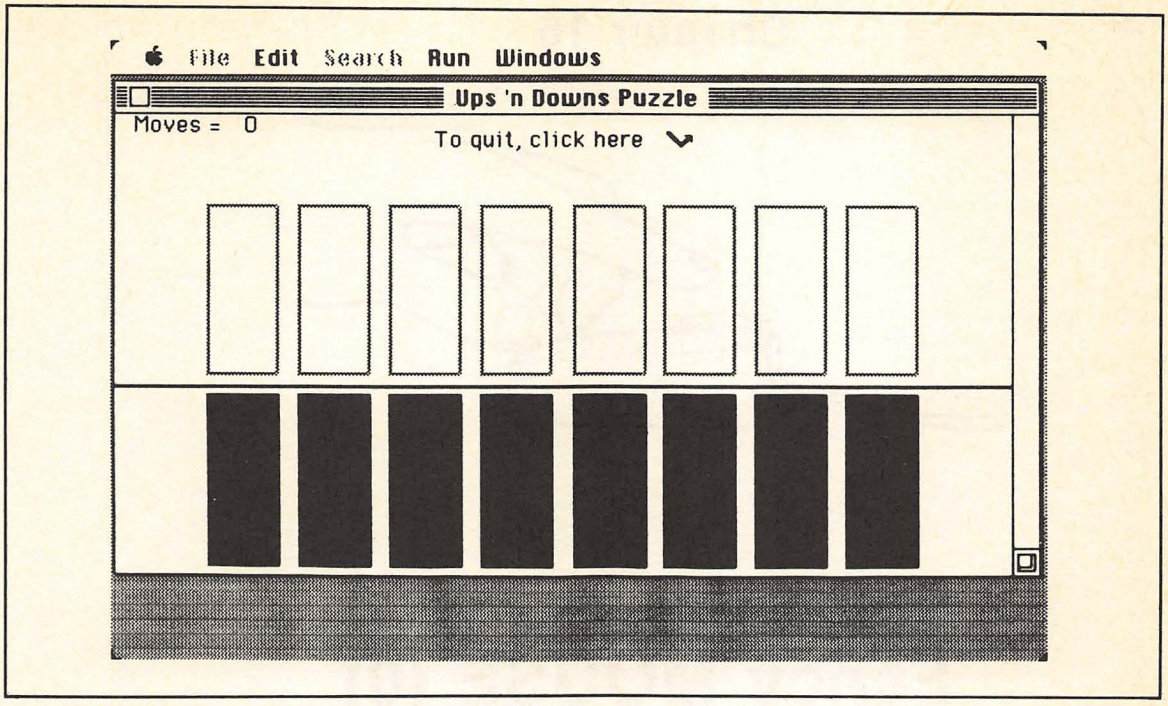


Fig. 15-1. The Ups 'N Downs puzzle display.

```
X > LEF(I) and X < RIG (I) and Y > TOP (I) and Y < BOT (I)
```

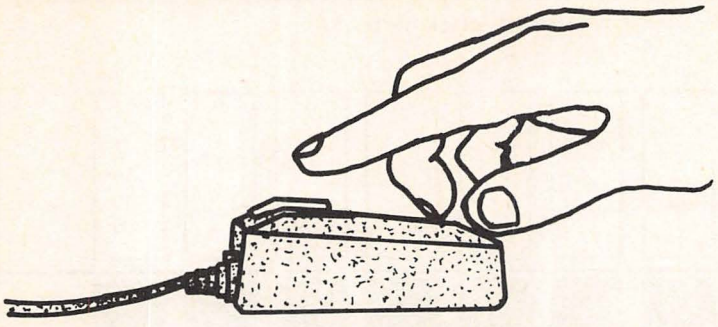
This is similar to the observation made in the discussion of the Mouse Utilities program. Each of the conditions is assigned a value by MS-BASIC. The value 0 is assigned when the condition is false and the value -1 is assigned when the value is true. Therefore, the expression:

```
(X > LEF(I)) + (X < RIG(I)) + (Y > TOP(I)) + (Y < BOT (I))
```

will have the value 0, -1, -2, -3, or -4, depending on how many of the four conditions are true. The point whose coordinates are (X, Y) lies inside the rectangle if and only if the value of the expression is -4. This simple idea is used in the subroutine beginning at the line 3000 of the program. The rectangles are scanned in a sequential search. The first one that satisfies the conditions causes a return from the subroutine to the main program. The number of the rectangle that was clicked is then contained in the variable *i*.

Ups 'N Downs is the only program in the book that has been left entirely in the old style with line numbers everywhere. Why don't you try converting it into the new style.

Chapter 16



Fancy MOUSE (0)

A great deal may be learned by watching the MOUSE (0) function in action. Fancy MOUSE(0), the program in this chapter, allows you to do just that. It is a program to play with and think about as you do. I will discuss some of the more obvious and important facts that may be gleaned.

THE PROGRAM LISTING

This program listing contains the symbol ✥. It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✥; for example, if the listing shows

```
IF XO < X THEN    ✥
inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN inside = 0 : RETURN
```

Program Listing 16-1
Fancy MOUSE (0)

```
'-----------------------------------------------------------------
' Fancy Mouse (0) Demo
'
' Written by
' Richard C. Vile, Jr.
' March 19, 1984
' Adapted to MS-Basic 2.0
' March 10, 1985
'-----------------------------------------------------------------
DEFINT a-z

  CLS
  delay = 100
  dblft = 150
  dbtop = 10

  WHILE -1

    GOSUB draw.delay
    FOR i = 0 TO 13
      FOR j = 0 TO 11
        ms = MOUSE (0)
        IF ms <> 0 THEN GOSUB check.delay :  ←
            CALL MOVETO (dblft + 52, dbtop + 14) : PRINT delay;
        FOR de = 1 TO delay : NEXT de
        CALL MOVETO ((j*32) MOD 384 + 32, (I*16) MOD 256 + 50)
        PRINT ms;
      NEXT j
    NEXT i
    PRINT : PRINT PTAB (50);"Click to continue ";
    ms = MOUSE (0)
    WHILE MOUSE (0) = 0 : WEND
    WHILE MOUSE (0) <> 0 : WEND
    CLS

  WEND

draw.delay:
  CALL MOVETO (dblft + 2, dbtop + 14)
  PRINT "Delay =";
  CALL MOVETO (dblft + 52, dbtop + 14) : PRINT delay;
  CALL MOVETO (dblft + 140, dbtop + 14) : PRINT "+ -";
```

```
  GOSUB draw.frame
  LINE (dblft + 148, dbtop) - (dblft + 148, dbtop + 20)
  LINE (dblft + 136, dbtop) - (dblft + 136, dbtop + 20)
  RETURN

draw.frame:
  re(0) = dbtop : re(1) = dblft : re(2) = dbtop + 20 : re(3) = dblft + 160
  CALL FRAMERECT (VARPTR (re(0)))
  re(0) = re(0) - 1 : re(1) = re(1) - 1
  re(2) = re(2) + 1 : re(3) = re(3) + 1
  CALL FRAMERECT (VARPTR (re(0)))
  RETURN

check.delay:
  x = MOUSE (3) : y = MOUSE (4)
  which = 0
  which = (x > dblft + 136)*(x < dblft + 148)*(y > dbtop)*(y < dbtop + 20)
  which = which + 2*(x > dblft + 148)*(x < dblft + 160)*(y > dbtop)*(y < dbtop + 20)
  IF which = 0 THEN RETURN
  IF which = 2 THEN GOSUB deck.delay : RETURN
  IF delay >= 100 THEN delay = delay + 50 ELSE delay = delay + 10
  RETURN

deck.delay:
  IF delay <= 100 THEN delay = delay - 10 ELSE delay = delay - 50
  RETURN
```

RUNNING THE PROGRAM

When you use the program, you will see how the values of MOUSE(0) change as various events occur. To begin with, until the mouse button is clicked at all, the value of MOUSE(0) will remain equal to 0. Figure 16-1 illustrates this very dull state of affairs.

If the mouse button is clicked once and held down, the value of MOUSE(0) changes from 0 to −1 and stays equal to −1 as long as the button remains down. When the button is released, the value does not immediately change back to 0. Rather, for at least one call to MOUSE(0) it will have the value 1 (see the explanation of MOUSE(0) = 1 in the MS-BASIC Interpreter manual). This is illustrated in Fig. 16-2. During the span of time illustrated in this figure, two separate single clicks of the mouse occurred.

Now things begin to get more interesting. Suppose the mouse is *double-clicked*; that is, the mouse button is quickly clicked twice in a row. If asked, you might say that the value of MOUSE(0) would go directly from 0 to −2. Unfortunately, you would be wrong. Well, that doesn't make you a bad person! Now think about it: in order to perform a double-click, you must first perform a single-click! That means that the values taken on by MOUSE(0) will be as follows:

```
0
0    Mouse button up
```

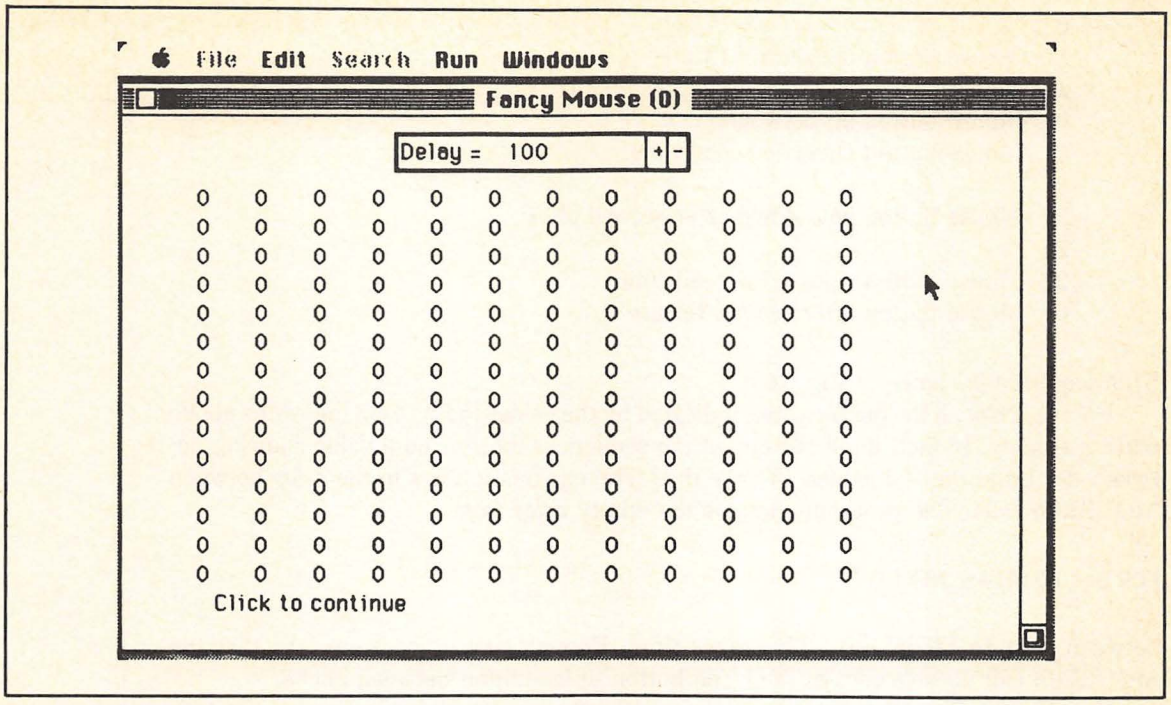


Fig. 16-1. The mouse is quiet (no clicks).

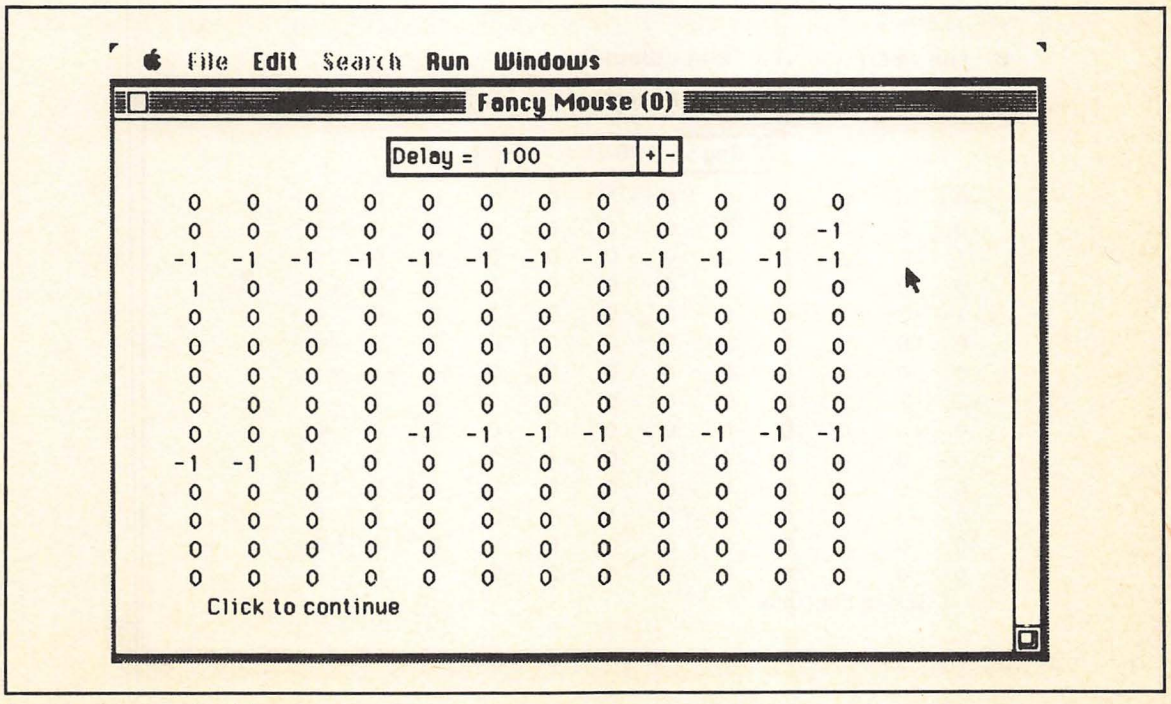


Fig. 16-2. Two single clicks.

0	
−1	Mouse button clicked first time
1	Mouse button released first time
0	Mouse button up between clicks
−2	Mouse button clicked second time
−2	
−2	Mouse button held down after second click
−2	
2	Mouse button released second time
0	Mouse button after second release

This sequence is shown in Fig. 16-3.

But wait! Not all the double clicks indicated by the values in Fig. 16-3 follow the above pattern exactly. In fact, in all but one of the sequences there is no 0 value showing between the 1 and the −2 values. Why is this? The explanation lies in the *delay* between MOUSE(0) calls. The program contains the empty *delay loop*:

```
FOR D=1 TO DELAY : NEXT D
```

between calls to MOUSE(0). This takes time. Enough time is consumed so that the MOUSE(0) call doesn't always catch the button up condition between clicks.

Now look at Fig. 16-4. Here the delay is 10, whereas the delay in Fig. 16-3 is 100.

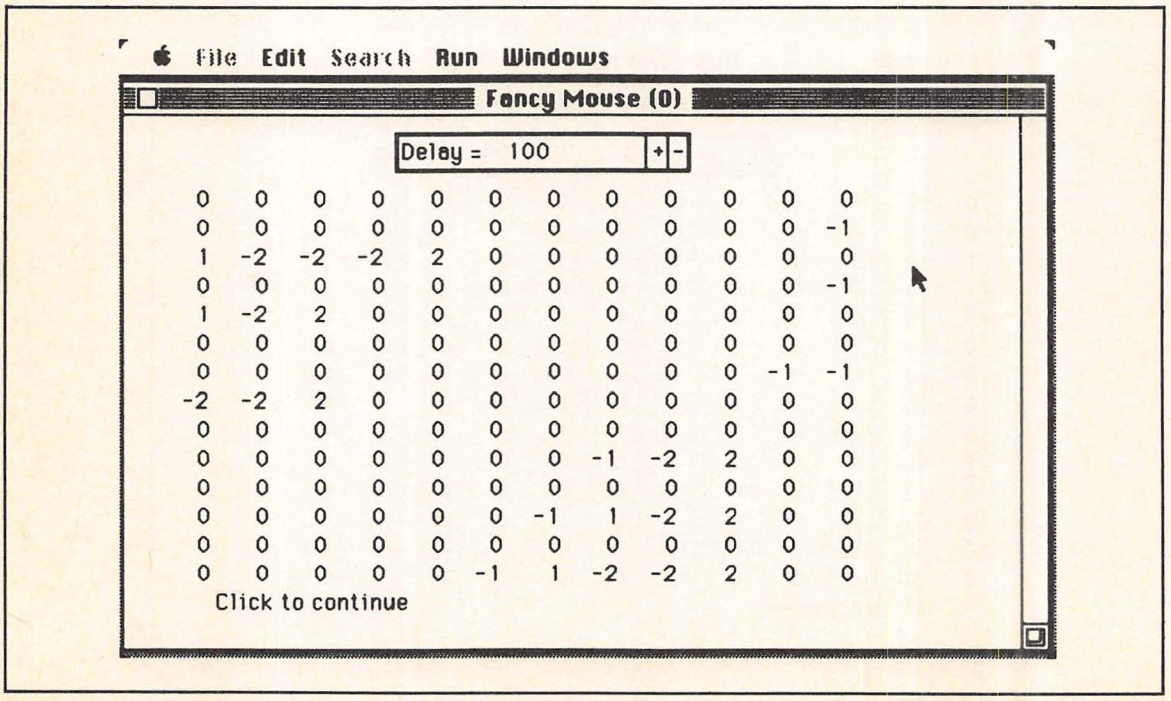


Fig. 16-3. A variety of double clicks.

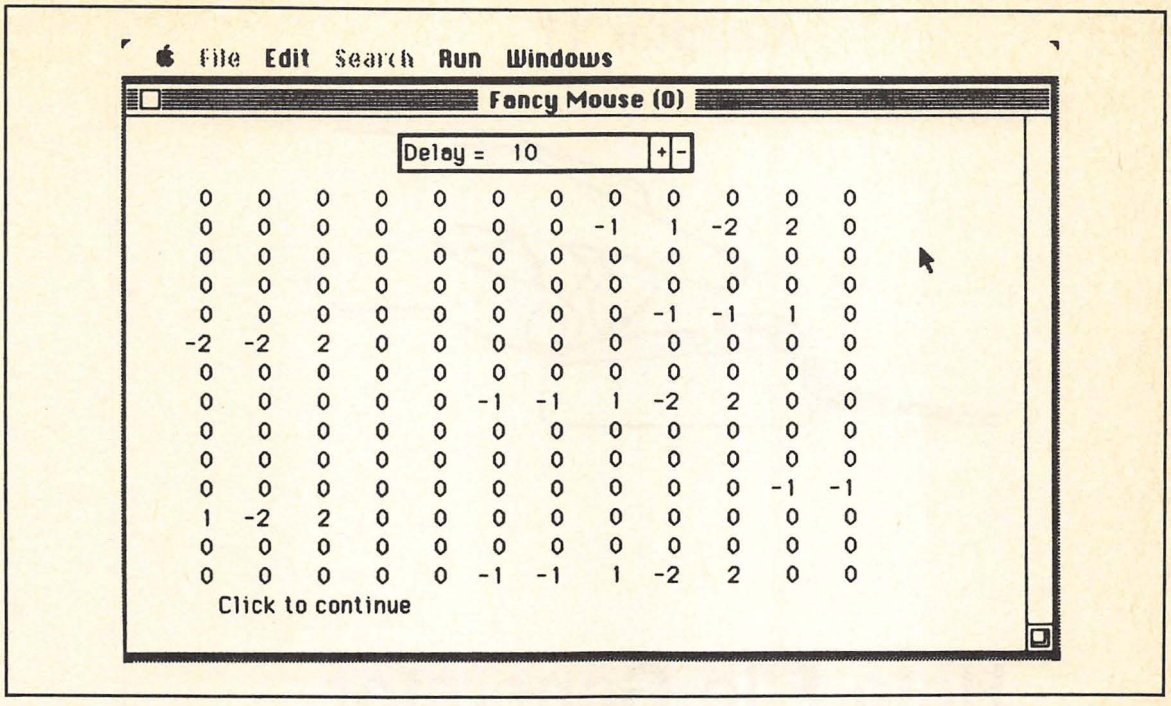
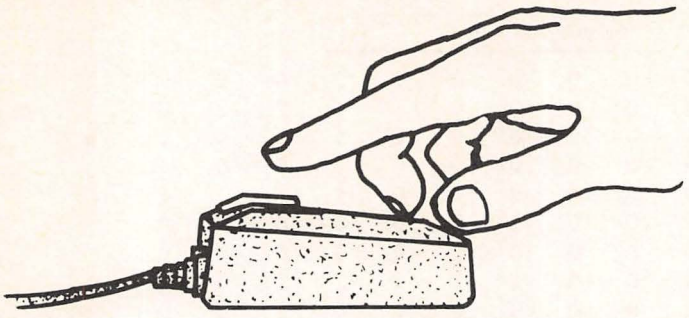


Fig. 16-4. More double clicks but less delay.

Figure 16-4 also illustrates double-clicks. Here the zero value is always present. In some of the instances, there are not one but two zeros between the 1 and the −2. The delay between MOUSE(0) calls in this case takes much less time. The calls come so quickly that it is possible for three of them to occur between the first and the second click. (Why did I say *three* instead of *two*?)

These examples demonstrate the fact that using the MOUSE(0) function is a *real time* problem. The results obtained from MOUSE(0) depend on events external to the program. The timing of those events relative to the calls to MOUSE (0) contained in the program is crucial to the outcome. A little experimentation and common sense will help us to avoid the nastier problems that can result from this uncertainty.

Chapter 17



Help File Generator

This program helps you with help. More specifically, it will create a file of help messages that can be used in another program. It allows you to compose your help messages interactively because it uses a large edit field that allows Return key presses. This means that the message you enter can be many lines long. As you are entering each help message, you make full use of the text editing capabilities of edit fields.

This program is a good candidate for enhancement and extension. It is in a primitive state as it stands, although it can be used with care. Later in the chapter there will be suggestions as to what you could do to enhance it.

THE PROGRAM LISTING

This program listing contains the symbol ✥. It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✥ ; for example, if the listing shows

```
IF XO < X THEN     ✥
    inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN inside = 0 : RETURN
```

Program Listing 17-1
Help File Generator

```
'-----------------------------------------------------------------
' Help File Generator
'
' Written by
' Richard C. Vile, Jr.
' March 3, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM hlp.txt$ (100)

INPUT "Program "; prg$
OPEN "R", #1, prg$ + ".hd", 80
FIELD #1, 80 AS hlp.tmp$
OPEN prg$ + ".hix" FOR OUTPUT AS #2

done = 0
help.num = 1
help.rn = 1
GOSUB build.help
CLOSE #1
CLOSE #2
END

build.help:
  WHILE NOT done
    CLS
    INPUT "Help topic title "; hlp.title$
    IF hlp.title$ = "quit" THEN RETURN
    EDIT FIELD 1, "", (5,40) - (400,250), 2
    BUTTON 1, 1, "Ok", (410, 40) - (450, 60)
    GOSUB get.text
    BUTTON CLOSE 1
    EDIT FIELD CLOSE 1
    IF debug THEN PRINT text$;"*****"
    hlp.len = 0
    GOSUB makelines
    PRINT #2, hlp.title$
    PRINT #2, help.rn
    PRINT #2, hlp.len
    FOR rn = 1 TO hlp.len
      LSET hlp.tmp$ = hlp.txt$ (rn)
      PUT #1
      NEXT rn
      help.rn = help.rn + hlp.len
```

```
    PRINT "Click to continue..."
    WHILE MOUSE (0) = 0 : WEND
    WHILE MOUSE (0) <> 0 : WEND
    help.num = help.num + 1
  WEND

get.text:
  WHILE -1
    d = DIALOG (0)
    IF debug THEN IF d <> 0 THEN CALL MOVETO (10,10) : PRINT "Got dialog event: ";d
    IF d = 1 THEN text$ = EDIT$ (1) : RETURN
  WEND

makelines:
  tmp$ = text$
  WHILE LEN (tmp$) > 0
    cr = INSTR (tmp$, CHR$ (13))
    IF cr >= 1 THEN nl$ = LEFT$ (tmp$, cr - 1) ELSE nl$ = tmp$
    PRINT nl$
    IF LEN (nl$) > 70 THEN PRINT "Warning: Truncating "; nl$ :  +
      PRINT "To: "; LEFT$ (nl$, 70)
    hlp.len = hlp.len + 1
    hlp.txt$ (hlp.len) = nl$
    tmp$ = RIGHT$ (tmp$, LEN (tmp$) - cr)
    IF nl$ = tmp$ THEN RETURN
  WEND
  RETURN
```

PROGRAM DETAILS

To start with, the program asks you for a program name. It doesn't want to know its own name! It wants you to give it the name of the program for which you are constructing a help file. It uses this name to build the names of the files it generates. For example, suppose you were building the help file for a program named Game5. Help File Generator would open two files named:

Game5.hd

Game 5.hix

These files will contain, respectively, the help data and the help index. The help index is what enables your program to locate each help message individually without having to read through the entire file; the help index functions just like the index in a book. The help data file contains the text of the messages you type in.

The Help Index File

For each help message, certain information is placed in the help index file (.hix file). This consists of three items:

1. The help message title
2. The record number where the message begins in the .hd file

3. The length of the message in .hd records.

Your program will read all of this information into memory when it begins.

The Help Data File

The help data file (.hd file) contains the text of all the help messages. It is a random access file. This enables any line in the file to be accessed directly; the lines preceding it do not have to be read first. The BASIC statements that set this up are:

```
OPEN "R", #1, prg$ + ".hd", 80
FIELD #1, 80 AS hlp.tmp$
```

For details, consult the MS-BASIC Interpreter Manual.

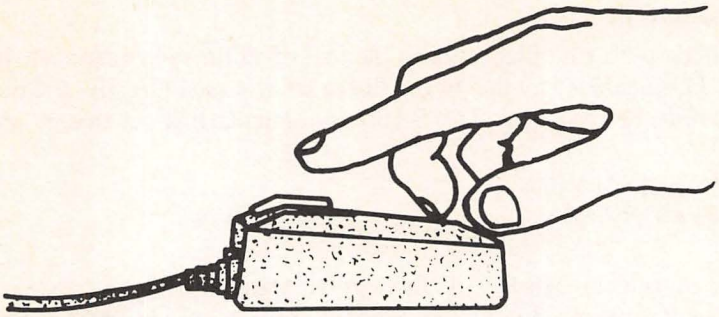
Each line of text stored in the .hd file takes up 80 bytes of space, even if the actual text is fewer than 80 characters in length. This is what allows the text to be accessed randomly. The penalty you pay for this is a few wasted "slots" in each text line in the file. But remember, putting all the help messages in a file saves an enormous amount of space for your program in memory.

HELP FOR THE HELP GENERATOR

This is a program that could easily turn into a full fledged application. Here are some suggestions for enhancements:

1. The first and most obvious is a way to review the messages that have been entered so far. This would be most easily accomplished with a menu choice.
2. If you add a menu, other "Options" come to mind. It might be nice to allow the user to rearrange the order of the messages while building the help file. This could involve simply rearranging the index, which is not written out to the .hix file until the text for all the messages has been entered.
3. Some way to retrieve an earlier message and alter its contents would be a good enhancement. You could implement this by reading in the old text, allowing the user to edit it, and then writing out the new copy at the end of the data file. The index would be changed to use the new message and ignore the old one. The cost of this approach would be some "wasted" space in the .hd file.
4. You could extend the retrieve and alter capability just mentioned to recover the "wasted" file space. This is a bit trickier and is potentially time consuming depending on what approach you take. I'll just let you muse on this one.

Chapter 18



The Help File Support

This chapter is a companion to Chapter 17. The BASIC code described here is not a complete program. It is designed to be included in other programs as support code for the help files generated by the Help File Generator. It consists of some DIM statements and three subroutines. You can keep this code in a separate MS-BASIC document. When you wish to use it in one of your programs, simply place the cursor at the end of the program and use the MERGE command to include the help support code. See the MS-BASIC Interpreter manual for details of the MERGE command. After you MERGE the code into your program, you may need to rearrange it a little. For example, you will probably want to put the DIM statements at the beginning of the program.

Program Listing 18-1
Help File Support

```
'-----------------------------------------------------------------
' Help Support Code
'
' Merge this with your program
' and use cut and paste to
' distribute the sections to
' appropriate locations
'-----------------------------------------------------------------
DIM hlp.title$ (20)
```

```
DIM hlp.rn(20), hlp.len(20)

help:
  hn = menuitem
  WINDOW CLOSE 1
  WINDOW 2, "", (50,30) - (455, 310), 2
  PRINT hlp.title$(hn)
  PRINT
  FOR rn = hlp.rn(hn) TO hlp.rn(hn) + hlp.len(hn) - 1
    GET #2, rn
    PRINT msg$
  NEXT rn
  GOSUB ok
  WINDOW CLOSE 2
  WINDOW 1
  RETURN

  helpmenu = 7
  MENU helpmenu, 0, 1, "Help"
  FOR hm = 1 TO hlps
    MENU helpmenu, hm, 1, hlp.title$(hm)
  NEXT hm

  RETURN

ok:
  BUTTON 1, 1, "Ok", (360, 240) - (390, 260)
  WHILE -1
    d = DIALOG(0)
    IF d = 1 THEN BUTTON CLOSE 1 : RETURN
  WEND

get.help.data:
prg$ = ""
OPEN prg$ + ".hix" FOR INPUT AS #1
OPEN "R", #2, prg$ + ".hd", 80
FIELD #2, 80 AS msg$

rn = 1
WHILE NOT EOF(1)
  INPUT #1, hlp.title$(rn)
  INPUT #1, hlp.rn(rn)
  INPUT #1, hlp.len(rn)
  rn = rn + 1
WEND
CLOSE #1
hlps = rn - 1
RETURN
```

THE DIM STATEMENTS

In order to access the help messages efficiently, the help index (.hix file) must be read into memory. This involves setting up three arrays to hold the information contained in the .hix file:

1. hlp.title$—The titles of the help messages
2. hlp.rn—The record numbers that indicate where each message starts
3. hlp.len—The length in records of each message

All three arrays have space for twenty items, which is the maximum number of help messages supported.

THE GET.HELP.DATA SUBROUTINE

The get.help.data subroutine reads the .hix file into the three arrays in memory. The copy of this code in the Listing contains the statement:

```
prg$ = ""
```

This must be changed for each program into which you merge the help support. Note that **prg$** contains the program name. It must match the name that precedes the .hd and .hix parts of the names of the help files. For example, in the Quickdraw Trainer program, **prg$** has the value "QDtrain." The rest of the subroutine should be easy to follow. It just reads in the data in the .hix file that was written out by the Help File Generator.

THE HELP SUBROUTINE

This subroutine does the work of displaying the help message. First, it closes window number 1 and opens window 2 for its own output. The title of the message is printed at the top of the window in bold face. Then the help message follows. A simple FOR loop reads through the lines of the help message and writes them to the output window.

Note: since the message was originally typed into an edit field, you can be reasonably sure that it will fit in the output window all at one time. Of course, this need not be the case in general. See below for suggestions.

THE OK SUBROUTINE

After the help message has been printed in the output window, the ok subroutine is invoked to detect when the user is done absorbing it. A button is created in the output window with the text "Ok." In the Listing, the button is given the number 1. If your program already has a button numbered 1, you may have a bug on your hands. If necessary, change the BUTTON statement to use a different number and avoid conflict.

A WHILE loop is used along with the DIALOG function to detect when the user presses the Ok button. This action is considered a signal to erase the button, close the help message window, and restore the program to normal operation.

THINGS TO DO

The way the help file generator is set up, there are a number of limitations as well as assumptions.

Each help message is currently limited to what will fit in the edit field. Actually you can type beyond the visible bounds of the edit field, but you won't be able to see what you typed. So practically speaking, you are limited to what will fit in the edit field. In a way this is good. It forces you to write concise and meaningful help messages. On the other hand, there may be times where there is just too much information to fit in. In such cases, it would be nice to have a way both to generate multiple "page" help messages, and a way to conveniently retrieve them. Care to give it a whirl?

Another limitation is the number of help messages. Since a menu can only have 20 entries, you are limited to 20 help message titles. If you have more than 20 topics you wish to fit in, you will have to resort to some sort of trickery. There are any number of possible schemes for this. Here are just a couple of ideas:

1. You could reserve one item in the help menu which allowed you to cycle back and forth between another set of messages.
2. You could present subtopic choices in a dialog box.

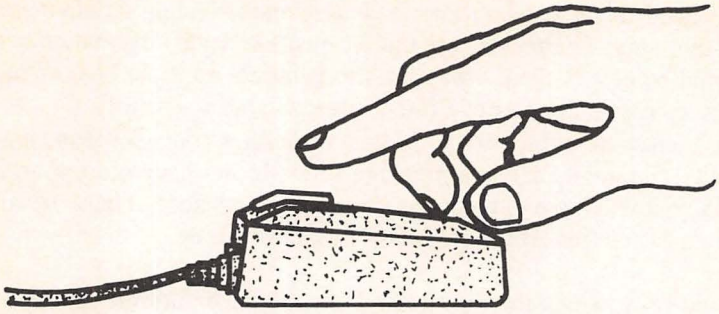
You will no doubt think of other ways to enhance the number of available messages.

The size of the message window is now always the same, regardless of the number of lines in the help message. For very short help messages, this makes the output look rather sparse; it might be considered slightly ugly by some people. If you are one of those people, here is your chance to do a little program modification. See if you can devise a scheme whereby the window varies in size according to the length of the message.

As I pointed out above, the code assumes that button number 1 is available. Unfortunately, there is no easy way to determine via a BASIC statement whether or not button number 1 exists. The function call BUTTON (1) will tell you the status of button number one if it exists. If it does not exist, then BUTTON (1) produces an *illegal function call* error. There is a trick that you can play here and it is described elsewhere in the book . . . Time for a little detective work!

The help support code assumes that window number 1 must be closed and that window number 2 is available for output. In general, of course, this won't be true. Your mission here, should you choose to accept it, is to write general code to save and restore both the active and the output windows in such a situation. Make sure that case in which active = output is considered, as well as the case in which all windows are closed to begin with.

Chapter 19



Please Take Note . . .

I have included this section for a variety of reasons. There are certain points that I wanted to make that either were not illustrated by a particular program or were too general to be tied to any specific program. I wanted to give some general advice on learning how to program interactively. I wanted to let my thoughts ramble a bit more freely and also to share some of my programming experiences with you. Most of the bugs included in Chapter Twenty-one were at one time or another infesting my programs. Maybe by telling you about them, I can help you exterminate some of the same bugs in your own programs. Finally, I want to advocate that you keep your own programmer's notebook. I will list the reasons I think this could be beneficial.

WHAT THERE IS IN THIS NOTEBOOK

The notes in this section of the book are grouped into several categories. There are the familiar topics of Macintosh programming including windows, menus, and buttons. There are also more general topics, which I have grouped in a category called "General." Each topic is preceded by a header that indicates its category and subject matter. You can read the notes in sequence, or you can simply browse. I would like to know whether you find this section helpful. See the end of the book for how to let me know your thoughts.

The bugs listed here are in no particular order and have no particular thread. Bugs are like that. They pop up randomly and in the most unexpected circumstances. That is one of the things that makes programming such an interesting and at times frustrating activity. You may not find your bug here. Or you may find it in a slightly different form

than you expected. In any case, read about the bugs and see if your debugging skill doesn't start to increase.

WHAT TO PUT IN YOUR NOTEBOOK

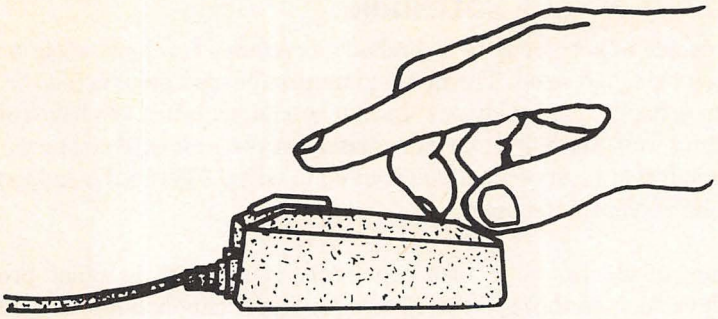
If you decide to keep your own notebook, here are a few hints about how to go about it, what to include, and so on. The first and most important point is this: be regular. Keep your notebook handy, preferably next to your computer. When you have something worth entering, don't wait. Write it down immediately. If you wait until you turn off your Macintosh, you will forget most of what you wanted to write. There is no end to the things you could include in your notebook:

- Fragments of BASIC code that you might reuse later in other programs. Short subroutines such as those in the Mouse Utilities chapter fall into this category.
- Notes about problems you encounter while writing a program—yes, bugs in your code! By writing down the problem and later the solution, you will increase the likelihood of avoiding that problem in the future. Also, this kind of diary makes interesting reading late some winter evening when you tire of programming itself.
- Points you glean while reading the MS-BASIC Interpreter manual, this book, or other books on programming. You might jot down summaries of commands that you use frequently. This last idea can be extended outside the domain of the notebook: use flash cards. I have several sets of 4×6 index cards with command summaries and miscellaneous notes that I refer to almost constantly while programming.
- Sketches of ideas you get about other programs. If you don't want to interrupt the program you are working on, write the ideas in your notebook. Believe me, if you don't, you are likely to forget. I can attest to that from personal experience.

WHAT SHOULD YOUR NOTEBOOK LOOK LIKE

Lest you interpret the name "Notebook" too literally, let me just offer some alternatives. If you don't like a book for writing in, use ordinary paper instead. Then to keep it organized, put it in a looseleaf binder. Or, if like me, you hate paper with holes punched in it, put your notes in file folders instead. I have several file drawers filled with notes from my programming and writing experiences. Be sure to review them periodically, however. It is easy to write down a good idea and then bury it in a file folder.

Chapter 20

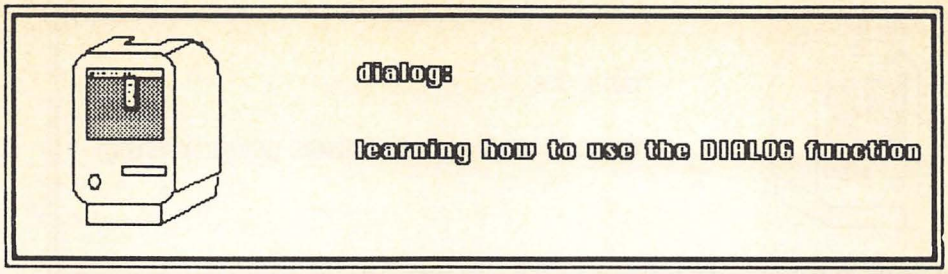


Macintosh Programming Notes

This chapter contains a potpourri of short sections. Each section deals with a specific programming topic. These are topics that I may not have dealt with in any of the example program chapters. They are all of potential interest and use to you, so do not assume that this is merely padding.

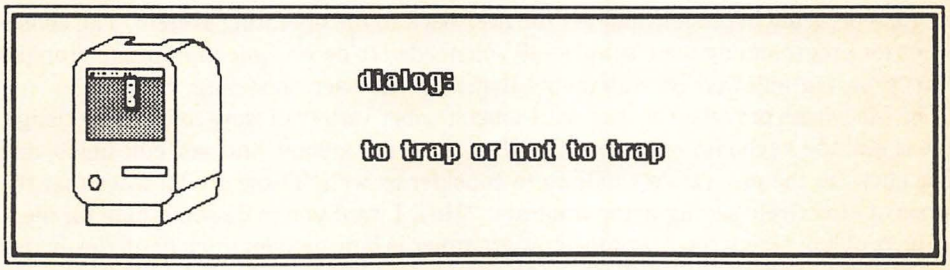
The notes are organized by categories; including menus, events, windows, and general topics. The best way to benefit by this section is to read a few sections at a time. Then try to apply what you have learned in a program. Alternatively, if you have questions about a specific area, you might consult the notes for that area. There are a number of points made here that fall into the category of "questions frequently asked."

There are too many points to be made about interactive programming for them all to fit in one book. If there are topics that you wish had been included but you find them to be absent from the book, let me know. If there is sufficient feedback and interest, there could be a follow up to this book. See the back of the book for details.



The DIALOG function provides information about the user's interaction with your program. A majority of the programs in this book make some use of it. These examples range from quite simple to reasonably complicated.

What I am about to say should come as no surprise. Your study of DIALOG should begin with the MS-BASIC Interpreter manual. The entry on the DIALOG function tells you how to interpret the return information from DIALOG (0) and how to use it in conjunction with the other DIALOG function calls.

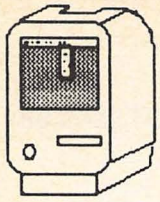


More often than any other event-trapping feature of MS-BASIC, the DIALOG function is used directly as opposed to with a trap. This is usually done when your program uses a dialog box to solicit some sort of input from the user. This happens at a specific point in the program. It is not something that can occur at any time and must be trapped. In fact, you don't want it to be trapped. You gather the input and go on. Since the input may involve using edit fields and buttons, using the DIALOG function is a natural way to handle the situation.

The ok subroutine that I have used in many of the example programs illustrates another simple situation. The program must wait until the user acknowledges that it is ok to proceed. One way to do this is to wait for the mouse button to be clicked. Another way is to wait for an Ok button to be pressed. This can be programmed quite simply. It is simple because, even though other dialog events can take place, you are not interested in them. So what you do is just wait for DIALOG (0) to equal 1, which means that a button has been pressed. Since there is no other button on the screen, this must mean that Ok has been pushed.

Filling in forms containing edit fields is usually done with direct DIALOG loops. There is generally a specific point at which the form should be filled out. In any case, once the form filling is started, it is easier to process dialog events directly than through trapping.

Dialog involving the change of windows is often trapped. This is because switching between windows is much more likely to be at the user's whim. It all depends on the design of the program, of course.



events:

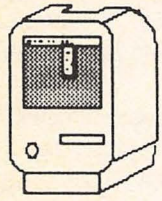
events and event-driven programming

You may have heard the Macintosh described as an *event-driven* computer. Just what does that mean and how does it apply to writing BASIC programs on the machine?

An *event* in this context is something that can happen that requires the computer's attention. Typically, an event results from something that the user does. In traditional computer dialogs, there wasn't much the user could do. Usually user interaction was limited to typing at the keyboard in response to specific prompts from a program or operating system. In cases of dire need, a special key combination like Control-C or Attention could interrupt a program in process and get the user back to the operating system. The consequences for programming were simple—all you needed to do was put appropriate prompts in your program followed by statements that read the users response. Simple!

The Macintosh provides the user with a much richer variety of ways to interrupt things. There is still the keyboard of course. But then there are menus, buttons, edit fields, and mouse clicks in the program's windows to consider as well. These are all ways that the user has of effectively saying to the program: "Hey, I want you to do something for me."

The problem facing the Macintosh programmer is how to keep track of all the events that occur and how to respond to them. There are an infinite variety of combinations of events that you might wish to handle. The programs in this book illustrate a number of mixtures of event handling.



events:

notes on trapping

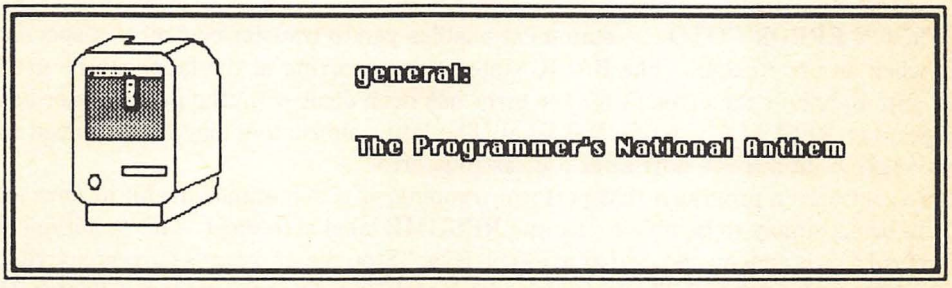
MS-BASIC provides a way to *trap* the occurrence of several different types of events. The ON statements listed here provide this capability:

ON MOUSE
ON MENU
ON DIALOG
ON TIMER
ON BREAK

I have used all but ON BREAK in one or more programs in the book.

Many programs take the form of an empty WHILE loop that simply waits until an event has occurred and relies on the trap to get it to the right place in the program. Other programs use ON MENU to catch menu events that occur while the program is busily engaged in other activities. Programming this is easy in BASIC, because the interpreter is checking for events behind the scenes. If you were using a language that had a compiler instead of an interpreter, you would be forced to interleave this checking with your other statements. The typical Macintosh application written in Lisa Pascal or C consists of a *main event loop* which constantly checks for all sorts of events. This loop may invoke other tasks if no events occur, but it must be careful not to allow any other part of the program to hog the machine and force the user to wait.

There are certain "events" that BASIC does not trap. For example, it does not react when the user types a command key (see Menu Maze for more on this). That event cannot be trapped in the same way a menu item selection is trapped. So if you want your BASIC program to handle it, you have to program an explicit test for such an occurrence. When a program explicitly tests for a condition, rather than relying on that condition being trapped, it is said to be *polling*. With an interpreted language, you have to do much less polling than with a compiled language. The interpreter does the polling for you behind the scenes.



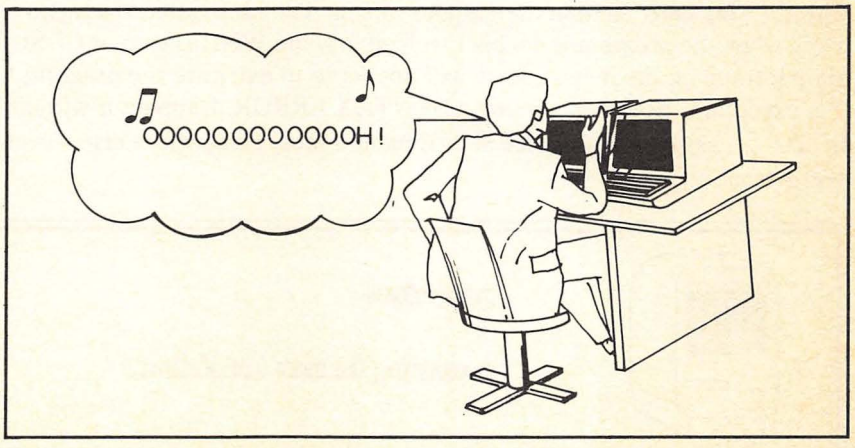
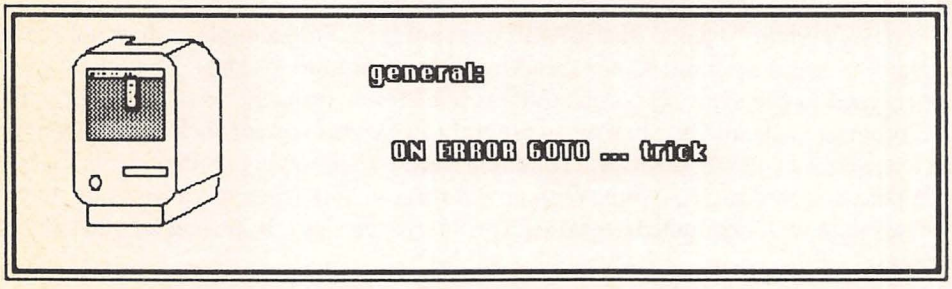


Fig. 20-1. The Programmers' National Anthem.

Programming is a unique activity. It involves the mind in ways that almost no other task does. It is fun to program and immensely satisfying when your programs work. When they don't, it can be another story.

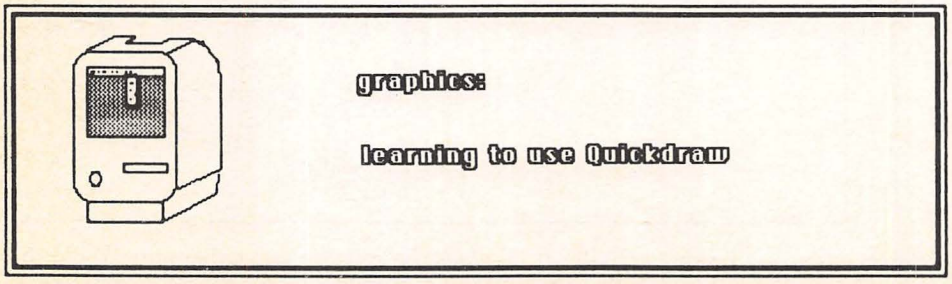
When a program has a problem, the programmer goes into a sort of suspended animation. Nothing matters but the program and its strange behavior. You will hear all sorts of muttered phrases such as: "I don't understand . . . It shouldn't do that! But I didn't make any changes that affected that. This has got to be a hardware problem. Hmmmmmmmm, why doesn't it get here?" and so on.

Eventually the problem gets solved. Some take longer than others, but they all work out one way or another. The most interesting part occurs when the programmer finally sees the light. Almost invariably this flash of insight, this sudden revelation, this removing of the blindfold causes the programmer to burst into a chorus of the Programmer's National Anthem:(Fig. 20-1) "OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOh!"



The ON ERROR GOTO . . . statement enables you to transfer control to a specific label when an error occurs. The BASIC statements occurring at the label can be used to attempt to handle the error. After the error has been dealt with, the programmer has the option to RESUME or RESUME label. The latter alternative may be employed to give the program a fresh start after a disastrous error.

In event-driven programs that perform trapping, it is sometimes useful to *force* an error to occur simply to be able to use the RESUME label statement. The prototypical example of this is a menu choice that in effect says, "Stop the program's current activity, but don't quit altogether." This occurs in the Vocabulary Quiz program in Chapter 24 with the **Stop Quiz** item in the **Options** menu. The idea is that the menu choice may be made when the program is deeply involved in some WHILE loop or GOSUB subroutine. Simply trapping the menu choice will not serve to extricate the program from the code it is executing. However, forcing a SYNTAX ERROR, trapping it with an ON ERROR GOTO . . . setup, and then RESUMEing at a label somewhere else solves the problem neatly.



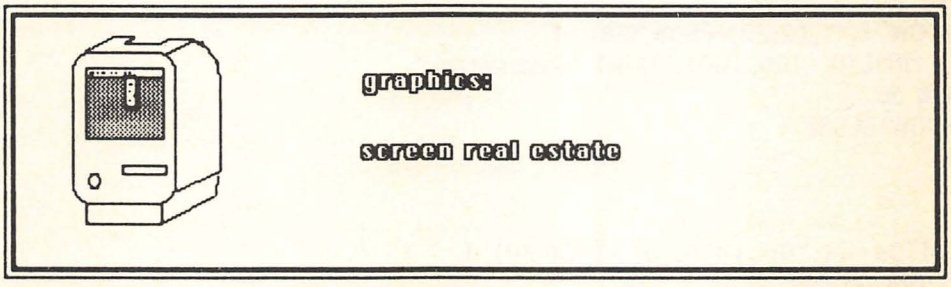
The MS-BASIC interface to Quickdraw is easy to use but difficult to master. There

are too many possibilities and too much potential. The examples I have given reflect my own interests and inclinations. I have tried to illustrate as much as I could, but there are no doubt gaps in the coverage.

An obvious place to start is the Quickdraw Trainer program. By running that program and comparing the output you get with the statements that produce it you can begin to grasp the possibilities of Quickdraw. After playing with the program as it stands, you can try implementing some of the suggested improvements. You can also think of your own ideas for improving the program and try them out.

Next you can study the other programs that use Quickdraw: Scribble and Video, Rose Patterns, Ups 'N Downs, and Mouse Kaleidoscope. Each of these programs produces graphic output and uses the Quickdraw calls in different ways. From the use of Quickdraw in these programs, you can build a beginning repertoire of techniques.

Then it is on to your own ideas and programs. Try designing simple geometric figures and writing short programs using Quickdraw to try to produce them. Then start combining such figures to see if you can draw simple pictures.



One of the annoying parts of using Macintosh graphics is figuring out the coordinates to use for all the rectangles. You have probably noticed that almost all graphic features involve a rectangle. The size and placement of these rectangles can be a matter of painstaking trial and error. This was one of my original motivations for developing the Dialog Designer application presented in Chapter 25. That program barely polishes the surface, let alone scratches it.

Here then is an open-ended challenge for you. Develop further tools and utilities to assist in allaying the rectangle coordinate blues. The following short program is another small contribution in this direction. It creates four windows in succession. Each window almost fills the desktop, leaving only a one pixel border. You can write the rectangles down on a 4×6 card and tack it in prominent view near your Macintosh. Of course, by subtracting the same value from each of the four numbers in the rectangle specification, you can create windows that leave a border of 2, 3, 5, or any number of pixels you like.

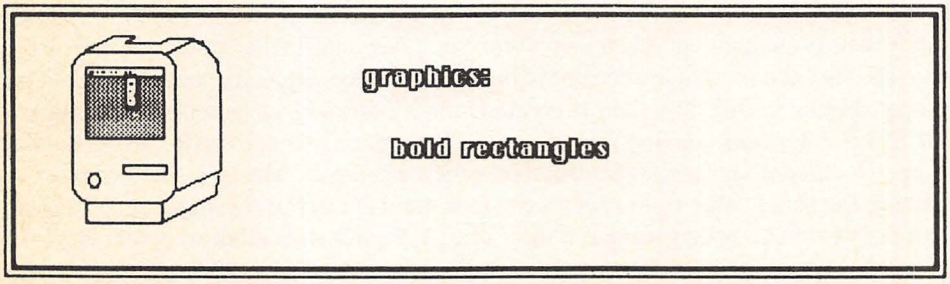
Program Listing 20-1
One-Pixel Window Borders

```
------------------------------------------------
' Windows leaving one-pixel border on desktop
```

```
'
' Written by
' Richard C. Vile, Jr.
' April 7, 1985
'-------------------------------------------------

WINDOW 1, "Startup", (2,40)-(500,250), 1
PRINT "Please close the command window before proceeding"
GOSUB ok
WINDOW 1, "Document Window", (2, 40) - (509, 339), 1
GOSUB ok
WINDOW 1, "", (9, 29) - (503, 333), 2
CALL MOVETO (100, 100) : PRINT "Dialog Box"
GOSUB ok
WINDOW 1, "", (2, 22) - (510, 340), 3
CALL MOVETO (100, 100) : PRINT "One line border"
GOSUB ok
WINDOW 1, "", (2,22)-(508,338), 4
CALL MOVETO (100, 100) : PRINT "Shadowed box"
GOSUB ok
WINDOW CLOSE 1
END

ok:
  BUTTON 1,1, "Ok", (400,10) - (450,30), 1
  d = DIALOG (0)
  WHILE d <> 1
    d = DIALOG (0)
  WEND
  RETURN
```



To draw a rectangle with an MS-BASIC program you can use either the BASIC LINE command or the Quickdraw CALL FRAMERECT. Now and then a program requires a fat, or bold, rectangle. I'll call it fat if its border is two or more pixels thick. It is easiest to accomplish this with a CALL FRAMERECT. There are at least three distinct ways to go about it. All three begin by drawing a rectangle a single pixel thick. This, of course,

involves setting up coordinates in an array: re (0), re (1), re (2), re (3). The second step in each technique is as follows:

1. Draw a rectangle nested inside the first. The resulting bold rectangle has outside dimensions equal to that of the first rectangle.

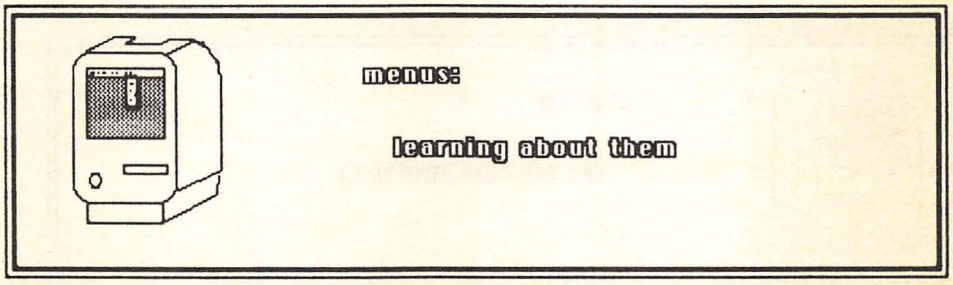
```
re (0) = re (0) + 1
re (1) = re (1) + 1
re (2) = re (2) - 1
re (3) = re (3) - 1
```

2. Draw a rectangle surrounding the first. The resulting bold rectangle has outside dimensions equal to that of the second rectangle.

```
re (0) = re (0) - 1
re (1) = re (1) - 1
re (2) = re (2) + 1
re (3) = re (3) + 1
```

3. Draw a rectangle down one pixel and to the right one pixel from the first. This doesn't quite finish the job. The top right and bottom left corners are left empty by this procedure. You must then use the PSET command to fill in these two pixels. The resulting rectangle has dimensions one pixel wider and higher than the first rectangle drawn.

```
re (0) = re (0) + 1
re (1) = re (1) + 1
re (2) = re (2) + 1
re (3) = re (3) + 1
PSET (re (0) -1, re (3))
PSET (re (1) -1, re (2))
```



Almost every program in this book uses menus in one way or another. This brings up two points:

- To write Mac programs you need to know about menus.
- You have a chance to study a lot of examples here.

Here is a suggested "course of study."

First Steps

First read and study Chapters 3 through 5 in Section 2. The programs in those chapters are short but not necessarily trivial. They already begin to present special techniques and points of interest.

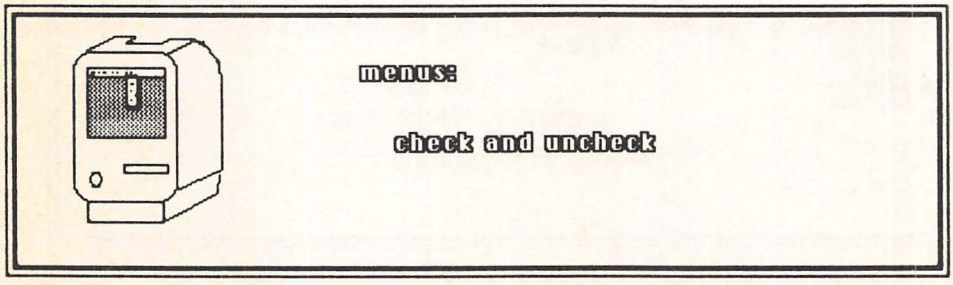
- Menu Trainer shows you how to use expressions in the menu command.
- Menu Trainer shows you how to clean the menu bar before putting up your own menus.
- Menu Trainer shows the menu features supported by MS-BASIC 2.0 in action in a program setting.
- Menu Trainer shows simply how menu features like checking and dimming are implemented with BASIC statements.
- Menu Maze shows you how to associate command keys with menu choices.
- Both Menu Trainer and Menu Maze show you how to use strings read from DATA statements to store menu item titles.
- Prototype Menu Code provides a general framework of BASIC statements that you can use in all of your programs to organize your menus.

Browse a Little

After your initial study of menus, you should skim through the remaining sample programs in Section 2 and look at the menu coding in each. Try to see how these programs follow the skeleton code presented in the Prototype Menu Code program.

Practice Makes Perfect

You can only read about menus for so long. Eventually you have to sit down and write a program of your own that uses them. See how much you can remember without referring to the sample programs. Or start with Prototype Menu Code and fill in the details for your own program.



In many programs you have one or more menu items that need to be checked and unchecked as the program progresses. There are some standard techniques for accomplishing this.

The First Method

If the menu item represents an on-off situation, it can be represented in the program by a variable. The variable can be assigned the value −1 when the menu item is checked and the value 0 when it is unchecked. To illustrate, suppose you have a game program that has the option of being timed or not (for example a question and answer game). There is an Options menu that contains the item Timed. The item is supposed to be checked when timing is being used and unchecked when it is not. Here is what you do:

1. Invent a variable: is.timed, for example.
2. Initialize it appropriately. E.g. is.timed = 0 (Timing is off)
3. Set the menu item to correspond:

```
MENU optionmenu, 3, 1, "Timing"  (if is.timed = 0)
MENU optionmenu, 3, 2, "Timing"  (if is.timed = -1)
```

4. Use the following statement to "toggle" the menu and the value:

```
IF is.timed THEN is.timed = 0 else is.timed = -1
IF is.timed THEN MENU optionmenu, 3, 2 else MENU optionmenu, 3, 1
```

The Second Method

Now suppose you have a group of items in a menu, and only one of them is ever supposed to be checked at a given time. If the user makes a selection of one of these items, it is supposed to become checked. At the same time, any previous check is supposed to be removed. It is easy to write sloppy code that winds up with all the items checked as the program progresses. This is embarrassing to you and unhelpful to the program user.

The easy way out of this particular quandary is to be a little "heavy handed." Whenever a menu choice is made of one of the items in the group, the first thing you do is to reset the entire group to be unchecked. That way you don't have to remember which one had the check mark:

```
FOR item = item1 TO item2
  MENU examplemenu, item, 1
NEXT item
```

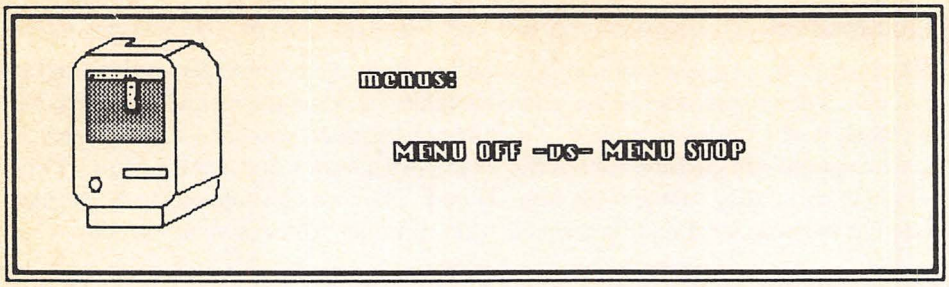
Then you check the item that was just selected:

```
MENU examplemenu, item.selected, 2
```

Voila, easy as falling off a log!

Other Techniques

Other combinations of circumstances are possible than the two just discussed. See the Quickdraw Trainer program in Chapter 12 for an example.



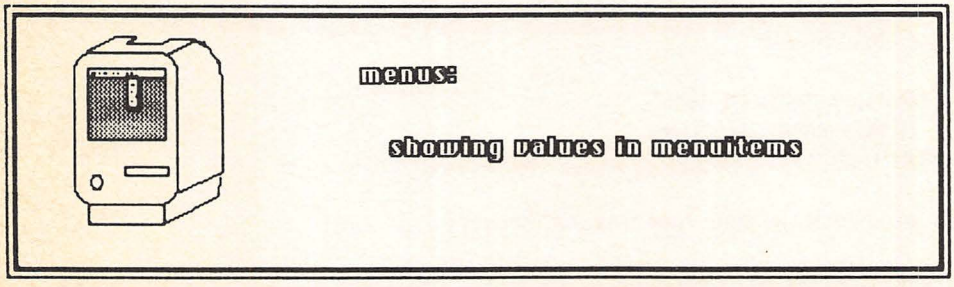
When a Basic program is interrupted because of a menu selection, there are three options for the handler subroutine:

1. Do not disable menu trapping at all
2. STOP menu trapping with a MENU STOP command
3. Totally disable menu trapping with a MENU OFF command

The first option is not recommended. It may cause the handler subroutine to be restarted before it finishes with its first item. This can cause confusion and even disaster.

The difference between (2) and (3) is that with (2) any menu choices that occur while the first choice is being processed will be remembered. Such choices are acted upon as soon as the handler turns menu processing back on. In option (3) any menu choices made while the first choice is being processed will be totally lost. This can be confusing to a user that is not aware of the difference between (2) and (3).

I recommend always using (2). If you really want the effect of (3), it is probably better to disable the menus or a set of menu choices that you wish to be ignored while the current menu choice is being processed. See the note on when and how to dim menus and menu choices.



Some menu items involve setting values of program parameters. It may be desirable to show the user the current setting by making it part of the text of the menuitem. How do you write BASIC statements to make this happen?

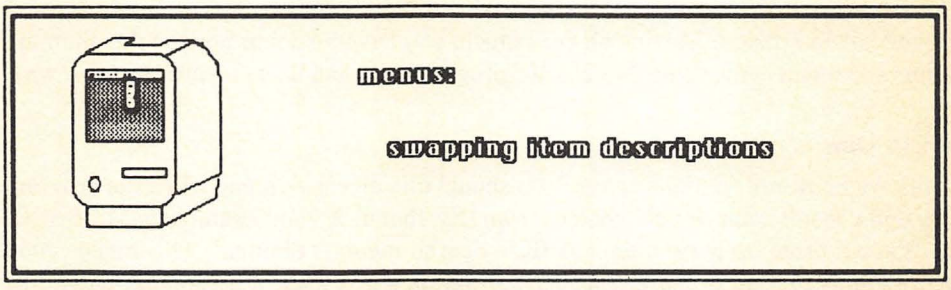
Converting Numbers to Strings

In order to print a numeric value in a menu item, all you need to do is to convert it

to text. The BASIC function STR$ does this for you. Feed it a number and it spits back a string at you. You can concatenate that string with the rest of the menu item.

```
MENU examplemenu, 1, 1, "Value of bozo: " + STR$ (bozo)
```

For consistency, you should put this statement both in the code that sets up the menus as well as the code that handles the menu item choice. Make sure that you have assigned the initial value of **bozo** (or whatever) before you set up the menus!



A menu item that represents a choice between two or more situations in the program may have its text changed as the program progresses. This is similar to checking and unchecking.

The Method

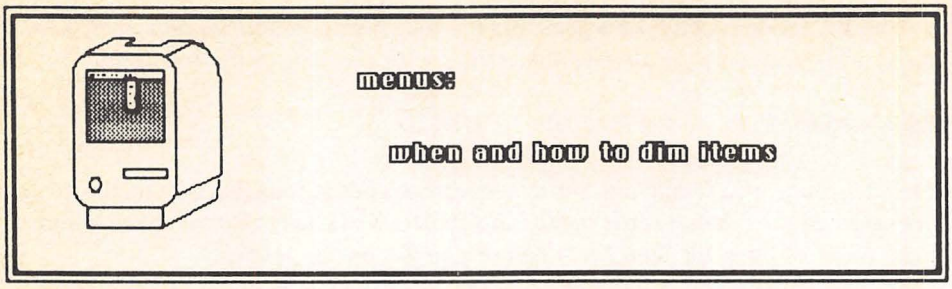
If the item has only two possible values, it can be represented in the program by a variable. The variable can be assigned –1 when the menu item has one value and 0 when it has the other value. To illustrate, suppose you have a graphics program that uses either "Fat lines" or "Skinny lines." There is an Options menu that has a choice that always contains one of these two descriptions—depending on which one the program is using. To implement this, here is what you do:

1. Invent a variable, **fat.lines**, for example.
2. Initialize it appropriately; for example, `fat.lines = -1` (Fat lines are used)
3. Set the menu item to correspond:

```
MENU optionmenu, 1, 1, "Fat Lines"
MENU optionmenu, 1, 1, "Skinny Lines"
```

4. Use the following statement to toggle the menu and the value:

```
IF fat.lines THEN fat.lines = 0 else fat.lines = -1
IF fat.lines THEN MENU optionmenu, 1, 1, "Fat Lines"
IF NOT fat.lines THEN MENU optionmenu, 1, 1, "Skinny Lines"
```



If you have used your Macintosh very much, you have no doubt noticed the dimming of menus and menu items. Your MS-BASIC programs can handle its menus the same way.

When to Dim

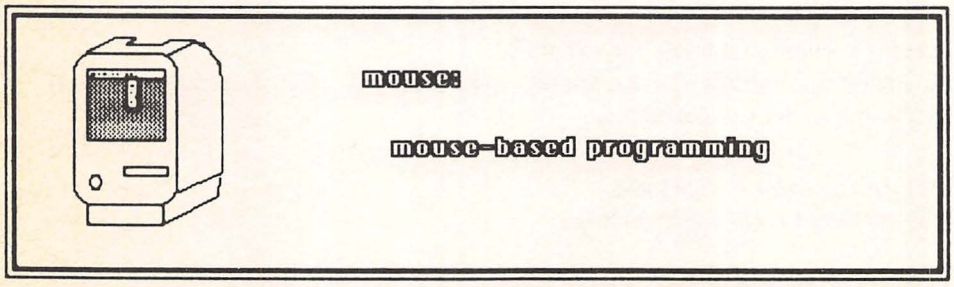
This is a matter of common sense. You should dim an entire menu when the program is busy and doesn't want to field choices from that menu. A good example is MS-BASIC itself. When a program is running, BASIC's Search menu is dimmed. This makes good sense—it is silly to want to search through a program's text while it is running. Individual items may be dimmed as well. A good example here is the Edit menu. When no selection is active, both Cut and Copy are dimmed. You can't Cut or Copy when nothing is selected! The minute you make a selection, both of these items become undimmed.

How to Dim

Dimming is easily accomplished with the MENU command:

```
MENU 1, 2, 0
```

for example, dims the second item in menu number 1.



In some ways the Mouse *is* the Macintosh. Certainly, without the mouse the Macintosh would not have the same appeal and charisma that it does. You shouldn't neglect the mouse if you want your BASIC programs to have the Macintosh look and feel.

There are many techniques that you can employ from BASIC for getting the most out of the mouse. I have tried to illustrate a variety of these in the sample programs.

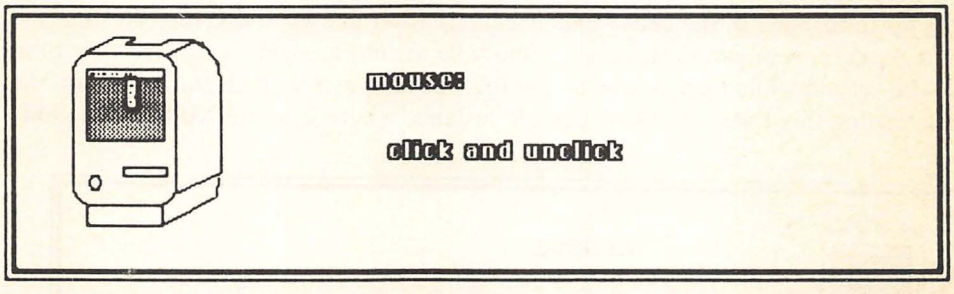
Building Mouse Technique

Your first step on the road to mouse mastery should be to absorb what the MS-BASIC Interpreter manual has to say on the subject. Consider that to be like doing finger exercises on the piano. The next step should be to read the examples in the programs of Section 2 and the relevant notes in this section. As you go, try to write some programs of your own that use the simpler techniques. Instead of copying the BASIC statements directly from the examples, try reproducing them from memory. If you encounter bugs, see if you can figure them out for yourself. The insight you gain will be worth it.

Simpler Mouse Techniques

Here is a list of a few of the simple things you can do and where you can find examples of them in the sample programs.

- Wait for a Click: Window Trainer, Pattern Maker, et. al.
- Detect a double click: Scribble, Video, Rose Patterns, et. al.
- Locate mouse in one rectangle: Mouse Utilities
- Locate mouse in multiple rectangles: Rose Patterns, Cursor Editor, Ups 'N Downs
- Trap to a subroutine on a click: Mouse Kaleidoscope
- Drag open a rectangle: Mouse Utilities
- Position a cross-hair set of axes: Mouse Utilities, Dialog Designer
- Track the cursor: Mouse Kaleidoscope



Suppose you cradle your trusty mouse in your somewhat sweaty palm and click it. You may think of that as a single action, but it is really two:

Click: Press the mouse button down until you hear the click
Unclick: Release the button, again hearing a click

Now suppose you have a program that needs to detect a click in order to take a certain action. You may need to detect the unclick as well! To take an example, look back at the Ups 'N Downs program presented earlier. In that program you will find the following lines:

```
2905 WHILE MOUSE (0) = 0 : WEND : REM ***** Wait for click *****
2906 WHILE MOUSE (0) <> 0 : WEND : REM ***** Wait for unclick *****
```

These statements are the prototypes for detecting a click followed by an unclick. You may omit the line numbers, of course.

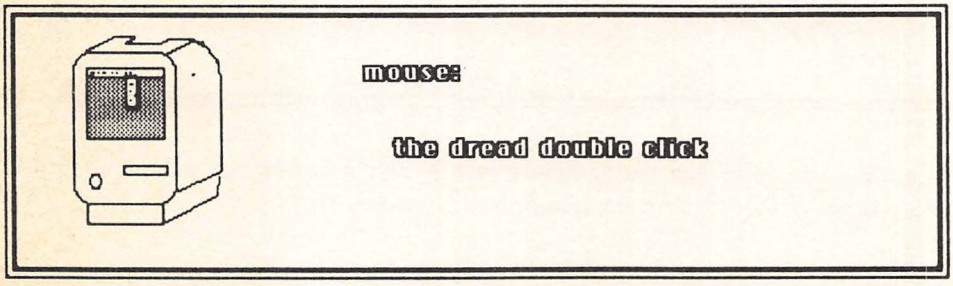
The original version of Ups 'N Downs was missing statement number 2906. I noticed when I played the game that if I clicked on a tab and held the button down, the tab would flip back and forth until I released the button. This didn't bother me, since I am quick on the button, and I could always time things so that the tab wound up where I wanted it. When my son played the game, however, he found it almost impossible to get the first tab (the one which takes the program the least time to locate) to move just once. It would always flip and flop for him, leaving it in exactly the position it started in. Needless to say, this frustrated my son and he quickly lost interest in the game.

A little thought revealed what was happening:

> Statement number 2905 was being executed
> twice for the same click!

BASIC is fast enough that by the time the mouse button is released, it has done all the processing it needs to in order to find the right tab to move, move it, update its records of the move, and come back to look for the "next" click. Well, if you aren't quick on the button, the first click turns into the second click, and the third click, and so on.

The solution of course, is to force BASIC to wait for the unclick before you let it go racing off to respond to the click. This is exactly what line 2906 accomplishes for you. If your program wants to detect a click and take action, it is almost always wise to include the second while loop as well as the first. In fact, as I sit here staring at my Mac screen writing this I can't think of a single instance where it would NOT be advisable.



Here is a quiz for you. Suppose your program executes the following statements:

```
PRINT "Click to continue, Double click to quit..."
ms = MOUSE (0)
WHILE MOUSE (0) = 0 : WEND
IF ABS (MOUSE (0)) >= 2 THEN END ELSE RETURN
```

What will happen? Think about it before you answer, and remember the Fancy MOUSE (0) program.

If you said that the program would always RETURN, then go to the head of the class! The point is that in order to perform a double click, you must first perform a single click. If your BASIC program doesn't take special action, it won't have a chance to see whether or not a second click ever took place. This situation is similar to the click-unclick problem discussed in the preceding programming note.

How to Solve the Problem

In order to field the double click, you must introduce a delay in the program and then read the mouse again:

```
WHILE MOUSE (0) = 0 : WEND
FOR de = 1 TO delay : NEXT de
ms1 = MOUSE (0)
IF ABS (ms1) >= 2 THEN END ELSE RETURN
```

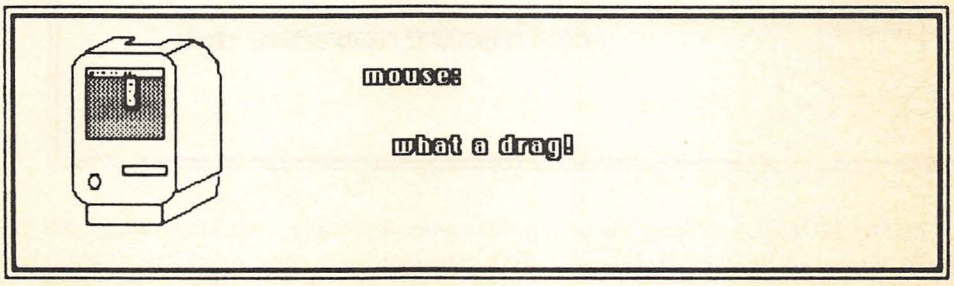
Remember the pattern of MOUSE (0) values during a double click:

0, -1, 1, 0, -2, 2, 0

The trick in the above BASIC fragment is to delay just the right amount so as to catch either the -2 or the 2. Experimentation will show the right delay value to use for yourself. Unfortunately, individuals vary widely in their reaction times. In addition, the Macintosh control panel allows you to change the time delay between clicks for a double click. So what will work at the fast setting may or may not work at the slow setting.

Challenge

Contemplate this problem for awhile and see if you can improve on the solution given here. A second technique will be presented in the chapter on solutions to the challenges.



Dragging with the mouse is next in line to clicking as the most common activity of Macintosh users. While I have not exploited the dragging idea as much as I have the clicking idea, there are some examples to be studied. The basic principles can be gleaned from them, and you can go on to invent your own techniques.

A drag operation consists of clicking the mouse button, holding it down, and moving the cursor to a new location. What happens in between depends on the situation. Here is the basic framework for dragging in BASIC:

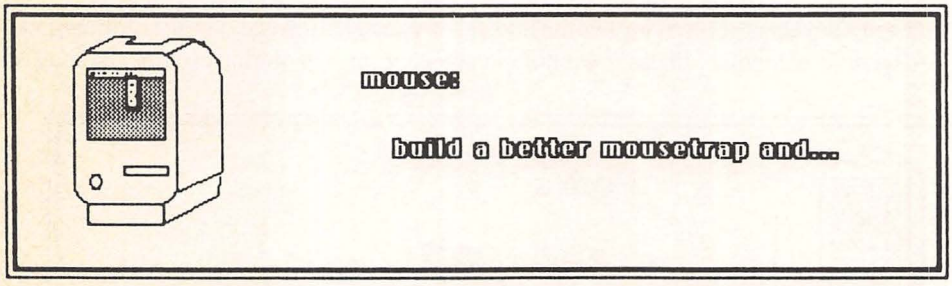
```
ms = MOUSE (0)
WHILE MOUSE (0) = 0 : WEND : ' Wait for click - start of drag
x0 = MOUSE (3) : y0 = MOUSE (4) : ' Record starting coordinates

WHILE MOUSE (0) <> 0
   x = MOUSE (5) : y = MOUSE (6) : ' Track mouse during drag
   '-----------------------------------------
   '  Do your thing with other Basic statements
   '-----------------------------------------
WEND
' Value of (x, y) gives coordinates of the point where drag ended
```

You can program anything you want to into this framework. What you can accomplish is limited only by what you can dream up to do with the values of the coordinates you receive from the MOUSE function.

There are two good examples of dragging in the book. The first is given in the Mouse Utilities chapter, Drag Open a Rectangle. The second is in the chapter on the application called the Cursor Editor. There I have implemented the idea of dragging that is used in the fat bits mode of MacPaint. Pixels in a graphics figure are inverted as you drag the mouse cursor over them on the screen.

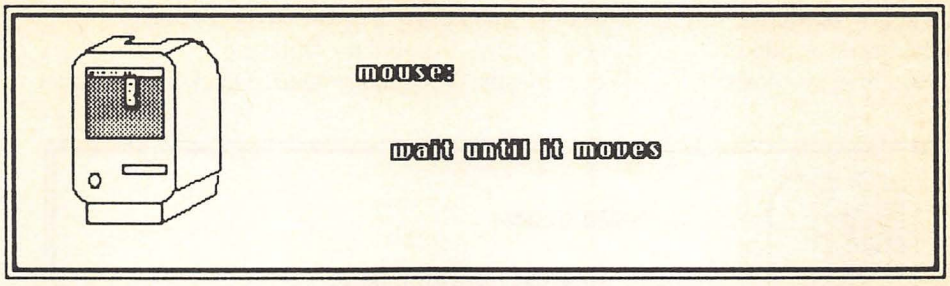
An excellent challenge for you to attempt is this. Try to write a program that allows you to drag a rectangle and its contents around the screen without destroying what it passes over. The part of the screen covered by the original rectangle should be left white as the rectangle vacates its original position.



The ON MOUSE statement is of limited value. Typically, you know when you expect the mouse to be clicked. In such cases, you test for a click explicitly, rather than relying on a trap. Still there are instances in which you might want to use ON MOUSE for simplicity.

The one program in which I have made use of the ON MOUSE statement is Mouse Kaleidoscope. There, the main activity of the program is producing graphic displays. The mouse location is used to gather coordinates, but clicking does not play a role in the graphics production. The mouse trap is used to allow the user to freeze the display as it is being

demonstrated. This is much easier to do with an ON MOUSE than with an explicit test.

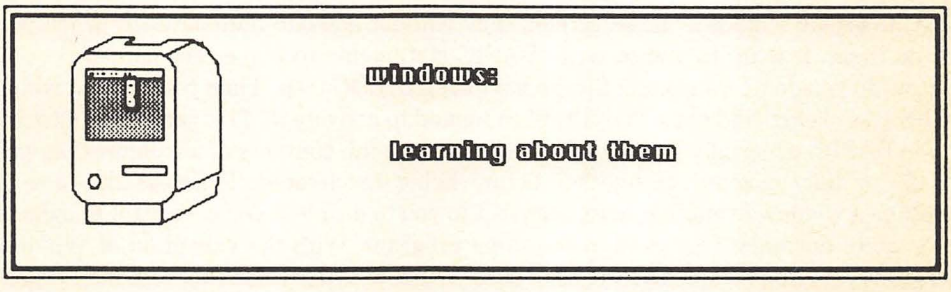


Here is an easy technique to add to your bag of tricks. The following loop can be used to detect any movement of the mouse. The MS-BASIC Interpreter uses this technique itself. When you stop a program either by choosing **Stop** from the **Run** menu or by typing command-period, a box appears in the upper right corner of the screen. It says: "Program stopped" and it stays visible on the screen until you move the mouse.

The idea is to record the coordinates of the mouse at some point and then test for either one of them changing to a different value:

```
ms = MOUSE (0)
x = MOUSE (1) : y = MOUSE (2)
WHILE (MOUSE (1) = 1) AND (MOUSE (2) = y)
   ms = MOUSE (0)
WEND
```

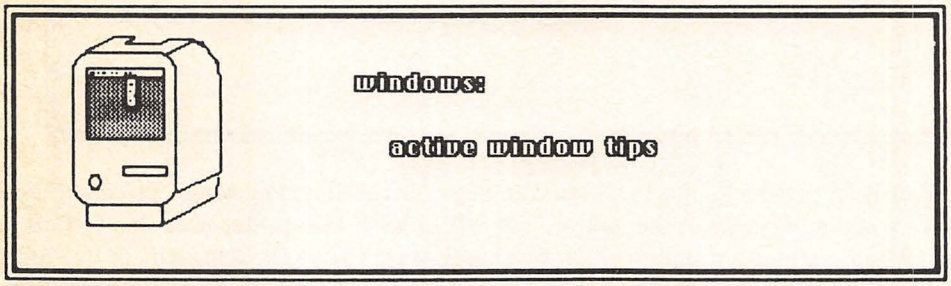
Here is a question which you can answer by a direct experiment. Are the calls to MOUSE (0) necessary in this technique?



You should begin your study of windows, as all other topics, by consulting the appropriate entries in the MS-BASIC Interpreter manual. Don't expect to absorb everything on the first reading, however. Try running the Window Trainer program and playing with it extensively. If you have a 128K Mac, you may have to change the menus to be much smaller than they are now.

Study other programs such as Pretty Flashy, Elevator, and Rose Patterns to see the diverse handling of windows. Try writing your first few programs with only one window. Here are some programs to consult to see how the various window types are used:

document windows:	Window trainer
framed windows:	Elevator, Mouse Kaleidoscope
one line borders:	Elevator, Quickdraw Trainer, Rose Patterns
shadowed window:	Pretty Flashy, Vocabulary Quiz
switching windows	Pretty Flashy, Vocabulary Quiz, Quickdraw Trainer



If you have read and studied the MS-BASIC Interpreter manual on the subject of windows, you have no doubt wrestled with the concept of active versus output window. Here are a few observations of my own.

The Differences from Other Mac Software

Most Macintosh applications adhere to the following convention regarding windows (at least document windows):

If you click on part of a nonactive window, it will automatically become active.

The Macintosh operating system does not guarantee this for you, however. Each application's implementors are responsible for seeing to it that it happens. When you use MS-BASIC to create windows on the screen, they will not activate automatically when you click on them. It is up to you to write BASIC statements to make that happen.

How do you do it? You detect the click with a DIALOG trap. Then you find out which window was clicked and use a WINDOW command to activate it. This can be quite tricky to do in BASIC, especially if you wish to reinstate the full contents of a window that was partially or fully covered by another before being reactivated. I discuss the idea of refreshing a window in another note. I leave it to you to decide if you ever want to include two or more document windows in the same program. With the exception of Window Trainer, no program included in this book does. Window Trainer does not respond to clicking automatically either. It has menu choices to allow you to try any combination of active and output windows.

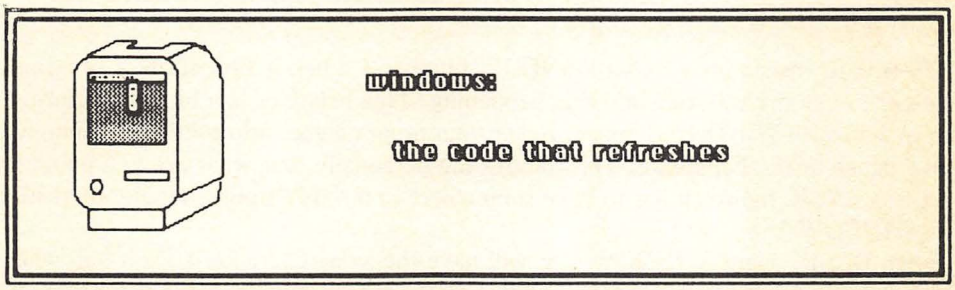
User Control over BASIC Windows

The only window that can be affected without the use of supporting BASIC code is an active document window. The user can resize, drag, or close such a window. If another document window exists, but it is not the active window, any attempt to resize, drag, or close it will fail, unless the BASIC program is specifically designed to respond to such

actions. There is a slight bug here. Some may call it a feature instead. See the entry titled "Wolf Windows in Sheep's Clothing" in Chapter Twenty-one.

Program Control over BASIC Windows

For a number of examples of the manipulations you can perform with the active and output windows and with multiple windows in your programs, see the chapters concerning the Pretty Flashy and Window Trainer programs.

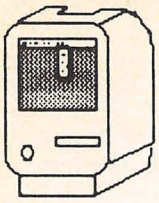


The DIALOG function will tell you when one of your windows has been altered to the extent that some of its data may have been erased. For example, an inactive window is partially covered by the user dragging the active window on top of it. Then the active window is closed or perhaps dragged off again. When such events occur, DIALOG (0) returns a value of 5, and DIALOG (5) returns the number of the window that needs to be refreshed.

If your program contains a subroutine that responds to DIALOG events, this situation can be handled under certain circumstances. The subroutine could be reached either by a trap set with the DIALOG ON and ON DIALOG GOSUB combination, or it could be invoked every so often by the program. There is an important difference between these two approaches:

1. The trap approach will catch one event at a time immediately as it occurs. It is like special delivery from the post office. The mail is delivered as soon as possible. The code in the subroutine need only check for one value in DIALOG (0).
2. The code used in the call-every-so-often approach must be able to field many events at once. This is like your daily trip to the mail box. You never know if there will be no mail at all or a whole stack of letters. The code in the subroutine must loop until the value in DIALOG (0) is found to be 0.

Certain conditions make it easier to handle refreshing windows in your BASIC programs. Typically, you should be able to refresh windows whose contents are not too complicated to reproduce and which don't vary much. For example, you might dedicate a window to the display of status information. Every now and then you call a subroutine to regenerate the contents of the window with the latest status values. You can call the subroutine an extra time then if you discover that the window needs refreshing. For an example of this type of handling, see the Pretty Flashy program.

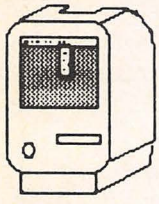


windows:

the startup window

MS-BASIC insists on opening WINDOW number 1 when it first starts a program. I have not found any way to avoid this happening. This behavior is a bit disappointing. If you wish to open WINDOW 1 yourself, then your program goes into a little schizophrenic window dance at the beginning. This annoys me personally, but what are you going to do? I guess BASIC figures it has to have somewhere to put its output, so it automatically opens WINDOW 1.

When BASIC opens WINDOW 1, it will have the same characteristics it had when it was last opened. This includes size, location on the screen, type of window (document, dialog box, and so on), and others. For this reason, your program can never really guarantee anything about the window. For that reason, I believe it is a good idea to force WINDOW 1 to be what you want it to be by using an explicit WINDOW 1 command at the beginning of every program. Now the embarrassing question is, have I practiced what I am preaching? I will leave it to you to find the programs that don't include an explicit WINDOW 1 command. I will probably hear you "Ah Hah!" all the way here in Dexter.



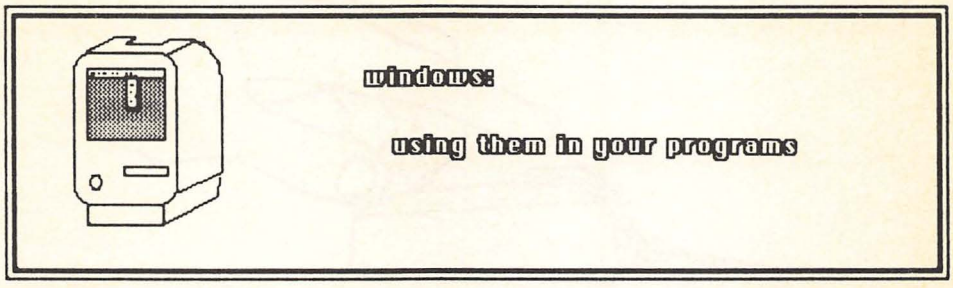
windows:

rectangle gotcha

When you create a window with the WINDOW statement, you specify a rectangle for the size and location of the window on the screen. That rectangle does not specify the entire window, however. It includes only the part of the window in which text and graphics may appear. It excludes the border, if any, of the window. In a document window, the rectangle excludes the title bar, the scroll bar, and the size box. If you don't take this into account, then parts of your windows may end up off the screen. The following short example illustrates this.

```
'-----------------------------------
' Hidden Title Bar for Document Window
'-----------------------------------
```

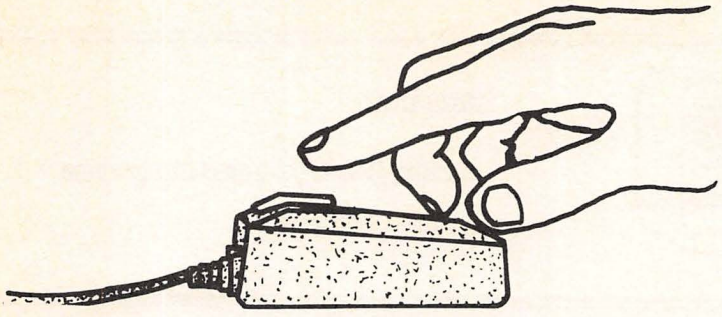
```
WINDOW 1, "Hidden From View", (5,30) - (160,60)
WINDOW 2, "Just Visible", (169,39) - (310, 60)
WINDOW 3, "One Pixel Separation", (315, 40) - (490, 60)
```



Here are a few tips on getting the most from your windows:

- Try to minimize the number of windows you use. I am not trying to say you should always stick with a single window, but only use multiple windows in a program if there is a good reason. Examples of good uses of multiple windows include help message information, scoring in a game, prompts for information, and dialog boxes.
- When you have dialog that is more complicated than detecting an Ok button push, use a dialog box window to conduct it in.
- Use nondocument windows for programs with lots of graphics. Examples in this book are Rose Patterns and Mouse Kaleidoscope. The reason for this is that you don't want the user to be able to resize the window. That can lead to ugly results at best.
- For graphics, use windows that fill the screen. I have left a border in the graphics programs in the book so that the figures would look better on the pages, but on the screen, a full white desktop is best. You don't have to worry about drawing outside the window either—Quickdraw sees to it that no harm is done.

Chapter 21



Bugs

Next you will find a collection of bugs I have encountered or narrowly avoided in my MS-BASIC experience. See how many of these seem familiar. You can read this chapter straight through. It is more likely that you will want to browse, however. Bugs are like caviar—a little goes a long way!

FAILING TO CLEAR THE DIALOG QUEUE

The DIALOG function potentially may contain more than a single value at any given time. When you write calls to DIALOG (0) into your program, BASIC returns these values to you one at a time. You have to be careful with this, because it will burn you if you are not.

A typical example of this might occur with buttons. Suppose you have several buttons that the user might press in a given context. Only one of these at a time makes any sense. Suppose further that your program is set up to trap the button pushes and a subroutine exists to handle the button pushes. The first thing that the subroutine does is to turn off further dialog trapping:

```
DIALOG OFF
```

This does not prevent events from being recorded in the DIALOG (0) queue. It only prevents the program from immediately trapping them. Now suppose the user accidentally presses a button while the first button push is being processed. Then the next time a further button push is trapped, the second push will be detected instead. This may or may not lead to trouble depending on which buttons get pushed the second and third times. But it can cause problems.

The solution is to *clear* the dialog queue before leaving the dialog trapping subroutine:

```
WHILE DIALOG (0) <> 0
WEND
```

This loop removes any unwanted or accidental events from the queue before the next trap occurs. Then when the trap does occur, the subroutine will find the event that caused the trap first in the queue instead of buried behind the other events.

FORGETTING THAT REFERENCE TO DIALOG (0) IS DESTRUCTIVE

This is the other side of the DIALOG coin. It is easy to write statements like the following:

```
WHILE DIALOG (0) <> 0
  IF DIALOG (0) = 1 THEN GOSUB handlebuttons
WEND
```

This most emphatically will not work! Look at the loop again. There are two calls to DIALOG (0) per time through the loop. Only the value of the second call is actually examined by the program. This means that every other event will be skipped! What is even worse, if this loop is in a subroutine that gets invoked from some higher level trapping routine, it may manage to miss all DIALOG (0) = 1 events! Think about it. DIALOG (0) returns 0 when there are no events left to report about. Suppose a program contains the above loop, intended to trap button pushes. The user of the program indeed pushes a button. The program traps the event and wends its way to the subroutine containing the WHILE loop. The first DIALOG (0) call in the WHILE statement returns 1 (a button was pushed), but doesn't do anything with it. The next call to DIALOG (0) in the IF statement gets a 0 back and does nothing. Next the WHILE statement invokes DIALOG (0) a third time, getting another 0 back. This all happens at BASIC speed—instantaneous compared to the time it takes the user to push the button again. So by the time the user tries the button push again, the program has dropped out of the WHILE loop and resumed listening for the next button push. When it comes, the same scenario just outlined will occur again . . . And again . . . And again!

The right way to process DIALOG (0) in a loop is as follows:

```
d = DIALOG (0)
WHILE d <> 0
  IF d = 1 THEN GOSUB handlebuttons
  IF d = 5 THEN GOSUB refreshwindows
  ' etc.
  d = DIALOG (0)
WEND
```

USING EDIT$ (N) WHEN EDIT FIELD N HAS BEEN CLOSED

This practice will cause an illegal function call error. The value of EDIT$ (n) is not defined when EDIT FIELD n does not exist. It is fairly easy to fall into this trap when

writing programs that process forms. The use of the field contents via the EDIT$ function is in a different subroutine than the subroutine that tidies up the screen after the form is filled in. You just have to make sure that the EDIT$ (n) call is made before the EDIT FIELD CLOSE n statement is executed.

A MENU SUDDENLY DROPS DEAD

It is easy to forget the MENU ON statement at the end of a menu handling subroutine. If this statement is omitted, you can still pull down the menus you have created, and you can still make choices, but nothing will happen. The program will not hear about the menu event unless the MENU ON statement is issued.

AN ACTIVE WINDOW GLITCH

The concept of an active window has been thoroughly discussed earlier. There is either a bug or a "feature" in the MS-BASIC handling of the active window. Suppose you have an application that uses two or more document windows. When you display those windows simultaneously, you will notice that only one of them at a time appears as the active window. As long as all of your windows appear on the screen, the window that appears to be active really will be active. The bug occurs if you use the mouse to close the active window. One of the remaining windows will suddenly become highlighted, thus appearing to become active. It does not become active as far as BASIC is concerned, however. If your program attempts to do output to the active window, that output will be lost. It will get dumped into the proverbial bit bucket.

THE MOUSE TRAP TRAP

It is not always safe to use dialog trapping. A good example is a situation in which you are using the mouse to detect on screen activity. For example, suppose you have a little program that allows the user to draw freehand sketches by dragging with the mouse. You set up an Ok button that is to be pressed when the user is finished drawing. If you try to trap the button press, you may be in trouble. The problem is that clicking the mouse for the drawing will also be detected as a dialog event, and the program will trap to the dialog handling subroutine. If your handler assumes, as mine often does, that the event must have been a button push, then no drawing at all will take place. You will have fallen into the mouse trap trap. Your event handler will have trapped the mouse when you didn't intend it to!

WINDOW CHARACTERISTICS SUDDENLY CHANGE

If you have started writing programs that use more than one window, you may have encountered this next bug. Once a window has been created, you may close it and reopen it without using the full syntax of the WINDOW command. This is more than a shorthand convenience! Each window has characteristics attached to it that determine the nature of the graphics and text that appear in that window. Suppose you open a window with the statement:

```
WINDOW 1, "Graphics", (10,10)-(450,300)
```

The resulting window will inherit the characteristics owned by the last incarnation of window number 1. On the other hand, if you open a window with the statement:

```
WINDOW, 1, "Graphics", (10,10)-(450,300), 1
```

the resulting window will have the *default* characteristics assigned to a newly created window. This difference can be difficult to spot. Especially if you have a lot of window switching and the logic of your program is complex.

Thus if you have been using PENS and calls to routines such as PENMODE, PENSIZE, and PENPAT, and not getting the results you expected, check your window statements and see if this might be the problem.

BE CAREFUL WHO YOU CALL WITH WHAT

The CALL statement gives you the ability to reach Macintosh ROM routines. The values that you pass to these routines are not first checked by BASIC to see if they are "sane" or not. You must be careful to make sure that you don't pass values to routines that could cause problems. I encountered an example of this while playing with a program to print text in various sizes. If you CALL TEXTSIZE (1) and then print something, you will get a rude shock. The alert box with the bomb in it pops up on your screen. The box contains the message ID = 4 down in the corner. If you look this up, you find out that a zero-divide error has occurred. The TEXTSIZE routine is supposed to set the current text size to 1 point. Evidently this causes some scaling routine somewhere to use 0 as a denominator in a division. Oh well, live and learn.

WANTED: BIG MENU TO MEET SKINNY MAC. OBJECT: BIG FAT BOMB!

If you create menus that take up a lot of screen space, you can run into difficulties. This is particularly true on a 128K machine. You know you are in trouble when you get the bomb alert box when you try to pull down such a menu (Figs. 21-1 and 21-2).

Here is my conjecture as to what is happening. You have no doubt noticed that the parts of the screen covered up by menus, no matter how large the area, are always restored when the menu pops back up into the menu bar. The way this is accomplished involves some magical things called *regions*. Each window on the Macintosh screen is composed of various regions, both visible and invisible. One of these is known in "Inside Macintosh" parlance as the *update* region. This is the part of the window that must be refreshed when the window comes to the top of the heap again. Actually this concept is not limited to windows. The entire desktop is an example of a *graphics viewport* that is associated with an update region. The parts of the window or graphics port that are covered up when a menu is pulled down become part of the update region. In order to restore this region, it is necessary to remember what it contained before it was covered up. One way to do this is simply to copy the whole thing to a separate area in memory. Then later when the menu disappears, it is copied back to the screen. Since there is no visible evidence of this copying process, I also conjecture that it must be interleaved with the menu pullback—but that is another story. When a large part of the desktop is put into the update region because a fat menu covers it, a lot of memory is required to store it. On a 128K Macintosh, this memory just isn't there and you get the bomb alert with an ID that

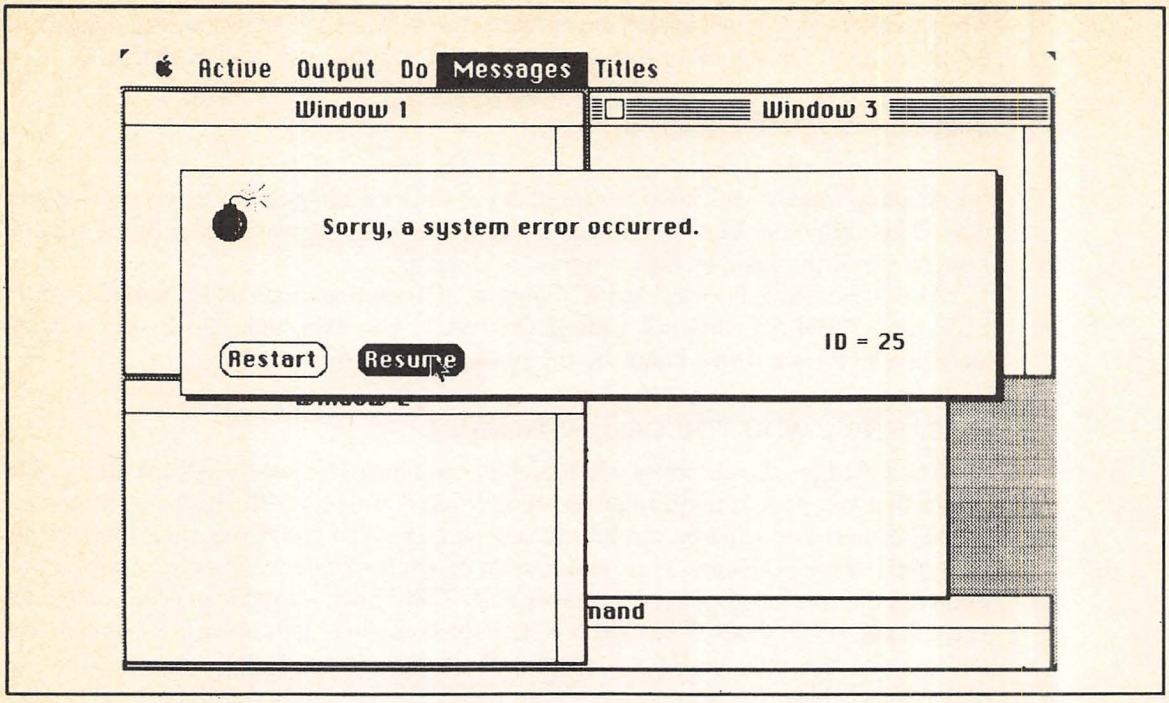


Fig. 21-1. A Bomb that appeared when I tried to display a huge menu on a Mac with a 316K RAM disk in operation.

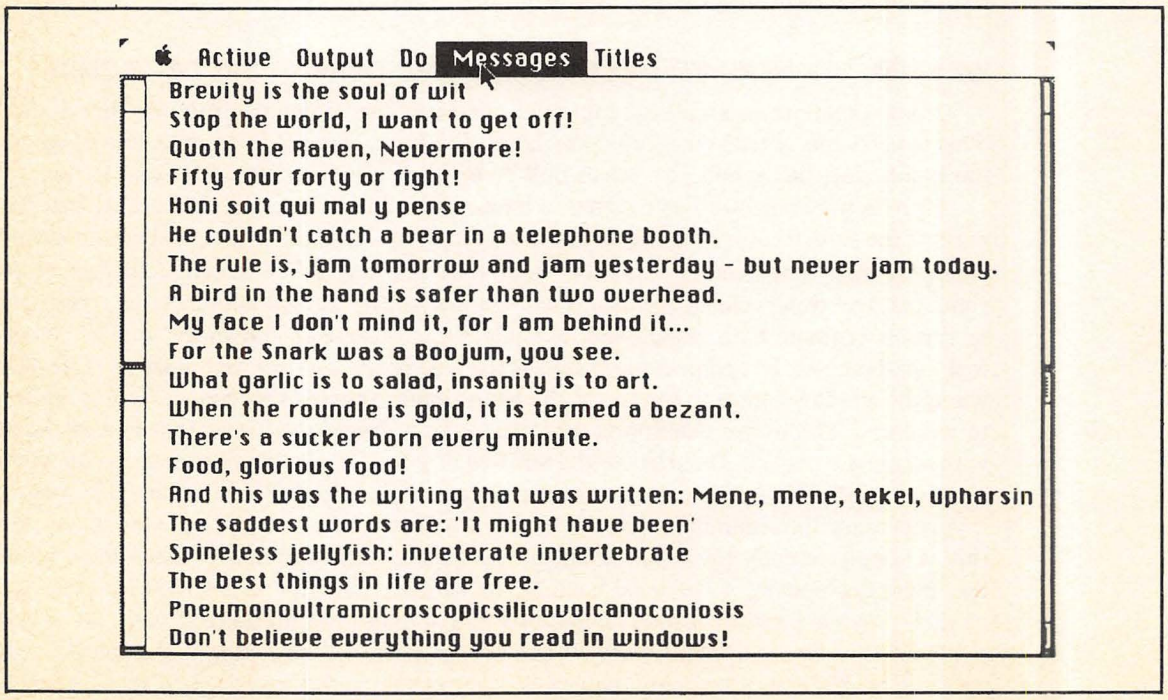


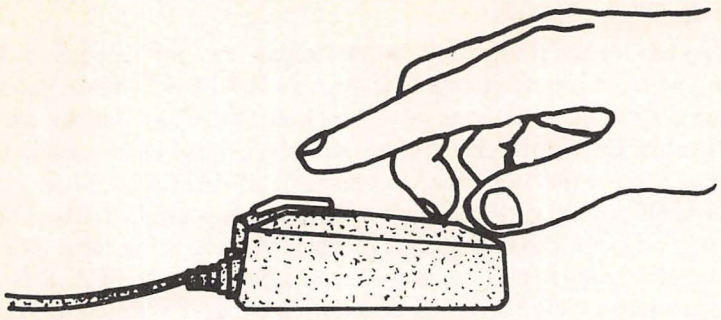
Fig. 21-2. The huge menu displayed using a 512K Mac with no RAM disk in operation.

indicates that you ran out of memory. The moral of the story is—don't make fat menus! If you have a 128K Macintosh, you will just have to live with skinny menus!

STRANGE INTERACTIONS

The DIALOG (0) has reared its head more than once in this chapter. Now consider DIALOG (n), where n is not equal to 0. If you call DIALOG (1) twice in a row, you may get the same answer back. It depends on whether or not any buttons were pushed between the first and the second calls. In the normal course of events, some will have been pushed. This is because the usual reason for examining DIALOG (1) is that you have already looked at DIALOG (0) and gotten a result of 1. Now it is possible to trick yourself in some situations. A good example is a situation in which you are using some other kind of trap as well as a dialog trap. For example, you might have an ON TIMER statement. This could occur in a game where a certain action by the player is timed. In such a situation, don't rely on the value of DIALOG (1) to reflect reality. Always use DIALOG (0) first! If DIALOG (0) is not equal to 1, no buttons have been pushed.

Chapter 22



Cursor Editor

I wrote Cursor Editor very soon after I acquired Microsoft BASIC. I used Version 1.1 to write it, and there are obvious artifacts of that fact in the program as it is presented here.

Cursor Editor is a miniature fat bits in BASIC. It shows you some of the techniques required by MacPaint in its fat bits mode. Of course, it also falls far short of MacPaint. You can learn about bit maps and their representation in programs by studying Cursor Editor. You can also produce neat little pictures to include in your programs via PUT commands. You will learn in more detail how GET and PUT arrays are structured by reading and studying this chapter and the program.

Cursor Editor enables you to build simple pictorial images directly in BASIC. You don't need to use MacPaint and transfer via CLIP:. With that method, you must read the pictures into your program from files, which can be time consuming. With Cursor Editor, you can turn the output into DATA statements that your program can use directly to load up a GET/PUT style array. By extending the program a bit, you can build libraries of pictures that you can access over and over again as you write more and more programs.

Cursor Editor gives you a chance to further your "mouse technique." The program is required to find out which square in a 16 × 16 grid of squares the mouse is clicked in. It also must track the mouse as it is dragged over subsequent squares in the grid. This provides a non-trivial example of BASIC processing during a drag operation. You may be surprised at how easy it is.

Figure 22-1 presents a structure diagram of the subroutines of Cursor Editor. You can use it as a quick introduction to the organization of the program. Notice the division of the program into two parts: those routines parallel to the box labelled Cursor Maker Menu and all those below that box.

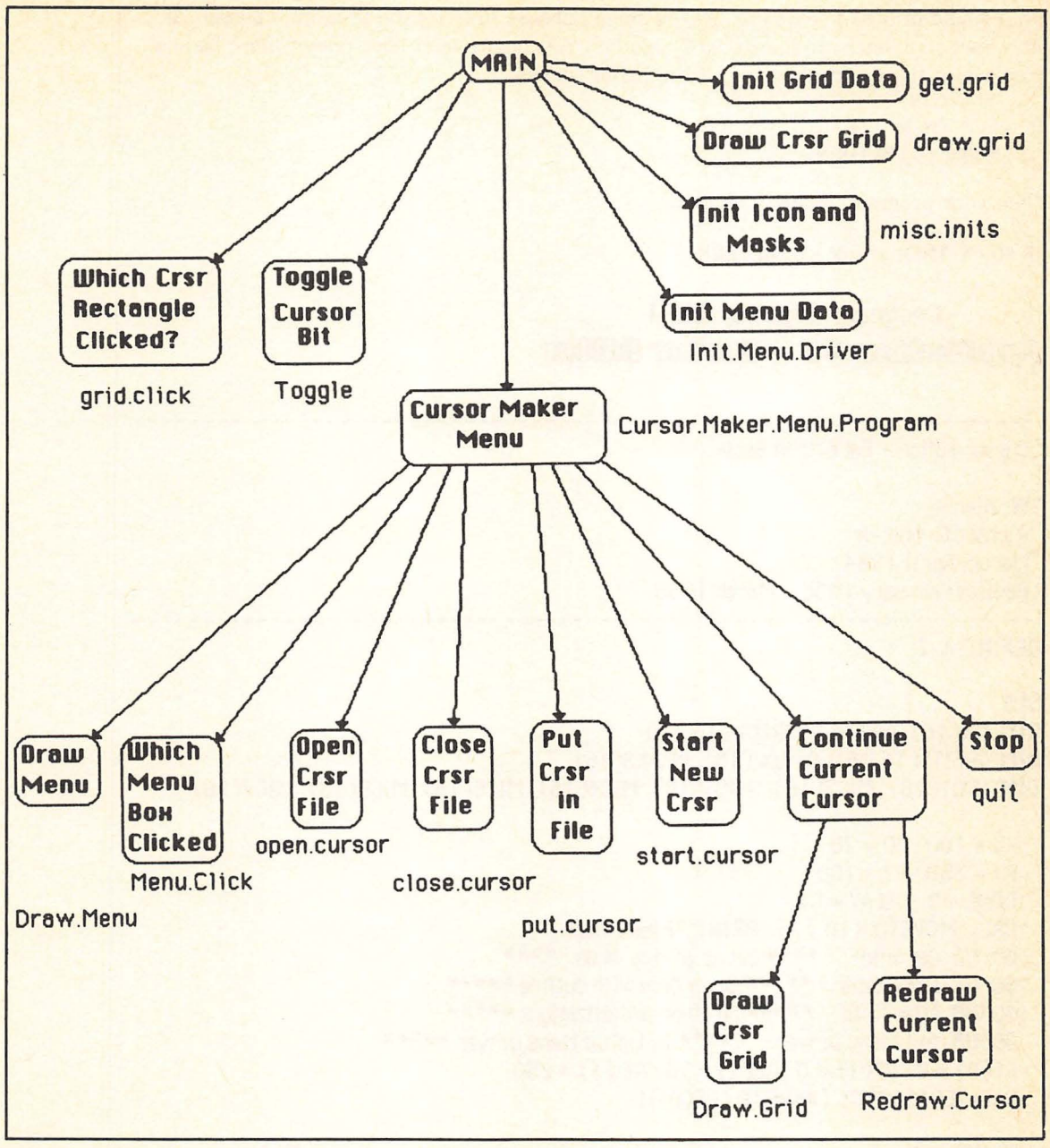


Fig. 22-1. A structure diagram for Cursor Editor.

THE PROGRAM LISTING

This program listing contains the symbol ♣. It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit

on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✥ ; for example, if the listing shows

```
IF X0 < X THEN    ✥
    inside = 0 : RETURN
```

then your program should look like:

```
IF X0 < X THEN inside = 0 : RETURN
```

Program Listing 22-1
Application #1 - Cursor Editor

```
'-----------------------------------------------------------------
' Cursor Editor - Fat Bits in Basic
'
' Written by
' Richard C. Vile, Jr.
' March-April 1984
' Revised February 1985, March 1985
'-----------------------------------------------------------------
DEFINT A-Z

CLS
DIM MX(16), MY(16), MRECT(15,15)
DIM CRBITS(16,16), Cursor(18), MASKS(16)
DIM MIC1(18), MIC2(18), MIC3(18), MIC4(18), MIC5(18), MIC6(18), MIC7(18)

  X0 = 100 : Y0 = 20
  PX = 350 : PY = 100
  DONE = 0 : DELAY = 50
  CALL MOVETO (10,10) : PRINT "Please wait...";
  GOSUB Get.Grid : ' ***** Set up grid co-ords *****
  GOSUB Draw.Grid : ' ***** Draw Grid with outline *****
  GOSUB Misc.Inits : ' ***** Do other initializations *****
  GOSUB Init.Menu.Driver : ' ***** Initialize Menu Driver *****
  RE(0) = 0 : RE(1) = 0 : RE(2) = 16 : RE(3) = 250
  CALL ERASERECT (VARPTR(RE(0)))

  WHILE -1
    WHILE -1
      MS = MOUSE(0) : WHILE MOUSE(0) = 0 : WEND
      LAST = -99 : LROW = -1 : LCOL = -1 : FIRST = -1
      WHILE MOUSE(0) <> 0 : ' ***** Allow dragging *****
        WHICH = 1
        GOSUB grid.click
```

```
        IF WHICH = 0 THEN 190 : ' ***** Click outside grid *****
        IF WHICH = -2 THEN 130 : ' ***** Avoid "blinkers" *****
        PUT (PX, PY), Cursor : ' ***** Erase previous preview *****
        GOSUB toggle : ' ***** Toggle bit in cursor *****
        PUT (PX, PY), Cursor : ' ***** Redisplay preview *****
130   WEND
      FOR de = 1 TO delay : NEXT de
    WEND
190 GOSUB Cursor.Maker.Program.Menu : ' ***** Menu *****
  WEND

grid.click:
'*********************************
'* Determine which cursor bit toggled *
'*********************************
  X = MOUSE (5) - XO : Y = MOUSE (6) - YO
  COL = X \ 10
  ROW = Y \ 10
  IF COL >= 16 OR ROW >= 16 THEN WHICH = 0 : RETURN
  IF (COL=LCOL) AND (ROW=LROW) THEN WHICH = -2 : RETURN
  LROW = ROW : LCOL = COL
  RETURN

 toggle:
 ' *********************************
 ' * Set appropriate bit in icbits array   *
 ' * and adjust cursor array accordingly  *
 ' *********************************
  IF FIRST THEN nowbit = Cursor (ROW + 2) AND MASKS (COL)
  IF FIRST THEN Cursor(ROW + 2) = Cursor(ROW + 2) XOR MASKS(COL)
  IF NOT FIRST THEN GOSUB adjust
  RE(0) = YO + ROW*10 + 2 : RE(1) = XO + COL*10 + 2
  RE(2) = RE(0) + 7 : RE(3) = RE(1) + 7
  IF FIRST THEN CALL INVERTRECT (VARPTR (RE(0)))
  IF NOT FIRST THEN IF parity THEN CALL PAINTRECT (VARPTR (RE (0)))  +
    ELSE CALL ERASERECT (VARPTR (RE (0)))
  IF first THEN first = 0 : IF nowbit = 0 THEN parity = -1 ELSE parity = 0
  RETURN

adjust:
  IF parity THEN Cursor (ROW + 2) = Cursor (ROW + 2) OR MASKS (COL)
  IF NOT parity THEN Cursor (ROW + 2) = Cursor (ROW + 2) AND (NOT MASKS (COL))
  RETURN

Draw.Grid:
'*************************
'* Draw grid and outline *
'*************************
```

```
  FOR I= 0 TO 16 : LINE (X0, Y0 + 10*I) - (X0 + 160, Y0 + 10*I) : NEXT I
  FOR I = 0 TO 16 : LINE (X0 + 10*I , Y0) - (X0 + 10*I, Y0 + 160) : NEXT I
  LINE (X0-1,Y0-1) - (X0-1,Y0+161)
  LINE (X0+161, Y0-1) - (X0+161, Y0+161)
  LINE (X0-1, Y0-1) - (X0 + 161, Y0-1)
  LINE (X0-1, Y0+161) - (X0 + 161, Y0 + 161)
  RE(0) = PY - 2 : RE(1) = PX - 2
  RE(2) = PY + 18 : RE(3) = PX + 18
  CALL FRAMERECT (VARPTR (RE(0)))
  RE(0) = RE(0) - 1 : RE (1) = RE(1) - 1
  RE(2) = RE(2) + 1 : RE(3) = RE(3) + 1
  CALL FRAMERECT (VARPTR (RE(0)))
  RETURN

Misc.Inits:
'**********************
'* Miscellaneous Inits   *
'**********************
  FOR I=0 TO 15
    FOR J=0 TO 15
      CRBITS(I,J) = 0
    NEXT J
  NEXT I
  Cursor(0) = 16 : Cursor (1) = 16
  FOR I = 2 TO 17: Cursor(I) = 0 : NEXT I
  MT = 75 : ML = 100 : MIL = ML + 250
  MASKS(0) = &H8000: MASKS(1) = &H4000
  MASKS(2) = &H2000: MASKS(3) = &H1000
  MASKS(4) = &H800: MASKS(5) = &H400
  MASKS(6) = &H200: MASKS(7) = &H100
  MASKS(8) = &H80: MASKS(9) = &H40
  MASKS(10) = &H20: MASKS(11) = &H10
  MASKS(12) = &H8: MASKS(13) = &H4
  MASKS(14) = &H2: MASKS(15) = &H1
  RETURN

Init.Menu.Driver:
'**************************
'* Set up Menu Rectangles, etc. *
'**************************
  FOR I=0 TO 17: READ MIC1(I) : NEXT I
  DATA 16, 16, 0, 30720, -31744, -31752, -30724, -28673, -25087
  DATA -17407, -2046, -4092, -8184, -16368, -32, 0, 0, 0

  FOR I = 0 TO 17: READ MIC2(I) : NEXT I
  DATA 16, 16, -1, -31711, -31711, -31711, -31711, -1, -31711
  DATA -31711, -31711, -31711, -1
```

```
  DATA -31711,-31711,-31711,-31711, -1

  FOR I=0 TO 17: READ MIC3(I) : NEXT I
  DATA 16, 16, 0, 0, 0, 31744, -32256, -31748, -1022, -32767
  DATA -32767, -32767, -32767, -32767, -32767, -1, 0, 0

  FOR I = 0 TO 17 : READ MIC4(I) : NEXT I
  DATA 16, 16, 8184, 12300, 24582, -16381, -32767, -4369
  DATA -15189, -6993, -23383, -6935, -32767, -32767
  DATA -16381, 24582, 12300, 8184

  FOR I = 0 TO 17: READ MIC5(I) : NEXT I
  DATA 16, 16, 0, 4080, 2064, 2640, 2064, 2064, 2640, 3024
  DATA 2064, 4080, 1056, 1056, 4080, 2064, 4080, 0

  FOR I = 0 TO 17 : READ MIC6(I) : NEXT I
  DATA 16, 16, 0, 0, 0, 0, 32, 8240, 16376, 16380
  DATA 16376, 8240, 32, 0, 0, 0, 0, 0

  ' ***** Continue current cursor
  FOR I = 0 TO 17:  READ MIC7(I) : NEXT I
  DATA 16, 16, -1, -31711, -31327, -31327
  DATA -31711, -1, -31711, -31327
  DATA -31327, -31711, -1, -31711
  DATA -19411, -19411, -31711, -1

  MX(0) = MIL + 1 : MX(1) = MIL + 49 : MX(2) = 500
  FOR I= 0 TO 7 : MY(I) = MT + 20*I : NEXT I : MY(8) = 300
  FOR I= 0 TO 7
    MRECT (I,0) = 0 : MRECT (I,2) = 0 :  MRECT (I,1) = I
  NEXT I
  MRECT (7,1) = 0
  RETURN

Redraw.Cursor:
'*******************
'* Redraw cursor on grid *
' *******************
  FOR I=0 TO 15
    FOR J = 0 TO 15
      IF (Cursor (I+2) AND MASKS(J)) = 0 THEN 5450
      RE (0) = Y0 + I*10 + 2 : RE (1) = X0 + J*10 + 2
      RE (2) = RE (0) + 7 : RE(3) = RE(1) + 7
      CALL PAINTRECT (VARPTR (RE(0)))
5450 NEXT J
  NEXT I
  RETURN
```

```
Cursor.Maker.Program.Menu:
  Done = 0

  WHILE NOT Done

    CLS
    GOSUB Draw.Menu
    GOSUB Menu.Click
    IF WHICH <= 0 THEN GOSUB continue.cursor : RETURN
    ON WHICH GOSUB open.cursor, close.cursor, put.cursor, start.cursor, continue.cursor,
quit

  WEND
  RETURN

Draw.Menu:
  PUT (MIL + 16, MT), MIC1
  PUT (MIL + 16, MT + 20), MIC3
  PUT (MIL, MT + 40), MIC5
  PUT (MIL + 16, MT + 40), MIC6
  PUT (MIL + 32, MT + 40), MIC1
  PUT (MIL + 16, MT + 60), MIC2
  PUT (MIL + 16, MT + 80), MIC7
  PUT (MIL + 16, MT + 100), MIC4
  CALL MOVETO (ML, MT + 16) : PRINT "Open cursor file"
  CALL MOVETO (ML, MT + 36) : PRINT "Close cursor file"
  CALL MOVETO (ML, MT + 56) : PRINT "Put cursor in file"
  CALL MOVETO (ML, MT + 76) : PRINT "Start new cursor"
  CALL MOVETO (ML, MT + 96) : PRINT "Continue with this cursor"
  CALL MOVETO (ML, MT + 116) : PRINT "Stop"
  FOR I = 1 TO 5
    LINE (ML-5, MT + 20*I - 2) - (ML + 300, MT + 20*I -2)
  NEXT I
  RE (0) = MT - 1 : RE(1) = ML - 6
  RE(2) = RE(0) + 123 : RE(3) = RE(1) + 307
  CALL FRAMERECT (VARPTR (RE(0)))
  RE (0) = RE(0) - 1 : RE(1) = RE(1) - 1
  RE(2) = RE(2) + 1 : RE(3) = RE(3) + 1
  CALL FRAMERECT (VARPTR (RE(0)))
  LINE (MIL, MT -1) - (MIL, MT + 121)
  LINE (MIL -1, MT - 1) - (MIL -1, MT + 121)
  MS=MOUSE (0) : WHILE MOUSE (0) = 0 : WEND

  RETURN

Menu.Click:
'****************************************
```

```
'* Determine Menu Rectangle clicked *
'****************************************
  MS = MOUSE (0) : ' ***** Clear mouse *****
  WHILE MOUSE (0) = 0 : WEND : ' ***** Wait for click *****
  X = MOUSE (3) : Y = MOUSE (4)
  XM = 0
  WHILE X > MX(XM) : XM = XM + 1 : WEND
  YM = 0
  WHILE Y > MY(YM) : YM = YM + 1 : WEND
  WHICH = MRECT (YM, XM)
  RETURN

open.cursor:
'********************
'* Open Cursor File    *
'********************
  CLS
  INPUT "Cursor file name "; CRFILE$
  OPEN CRFILE$ FOR OUTPUT AS #1
  RETURN

close.cursor:
'********************
'* Close Cursor File *
'********************
  CLOSE #1
  RETURN

put.cursor:
'********************
'* Put cursor in file *
'********************
  CLS
  INPUT "Name for cursor "; CRNAME$
  PRINT #1, CRNAME$
  FOR I=0 TO 17
    PRINT #1, Cursor(I)
  NEXT I
  CLS
  FOR I = 2 TO 14 STEP 4
    PRINT "Cursor "; I-2; "->"; I + 1; ":";
    PRINT Cursor(I); Cursor(I+1); Cursor(I+2); Cursor(I+3)
  NEXT I
  MS= MOUSE (0)
  PRINT "Click to continue"
  WHILE MOUSE (0) = 0 : WEND
  WHILE MOUSE (0) <> 0 : WEND
```

```
  RETURN

start.cursor:
  CLS
  GOSUB Draw.Grid
  GOSUB Misc.Inits
  Done = -1
  RETURN

continue.cursor:
  CLS
  GOSUB Draw.Grid
  PUT (PX, PY), Cursor
  GOSUB Redraw.Cursor : ' ***** Redraw blobs on grid *****
  Done = -1
  RETURN

quit:
'***********
'* STOP   *
'***********
  END
```

PROGRAM DESIGN CONSIDERATIONS

Since the program was designed and implemented with MS-BASIC 1.1, most of the interactive programming features are conspicuous by their absence. The program, however, has its own menu in the form of a screen display from which the user can make choices by clicking the mouse. Figure 22-2 shows the program menu screen, with its pictorial item descriptions. You click on the pictures to make the corresponding menu choice. These pictures, or mini-icons, or cursors, if you will, were produced by an earlier version of the program itself. The menu is not a pull down menu. BASIC 1.1 did not provide the MENU command. This menu is more like a sit there and stare at you menu. Still, it has nice pictures!

The program originally had line numbers for every line. I modified it to remove most of the line numbers and to substitute names for numbers as the subroutine labels. There are a few line numbers left lying around in the program. It is an interesting and fairly easy exercise to remove them if you so desire.

PROGRAM CAPABILITIES AND USE

Cursor Editor allows you to create Macintosh graphics that are up to 16 × 16 pixels square. Figure 22-3 shows the initial screen, and Fig. 22-4 shows a typical session with Cursor Editor in full swing. The mouse is used to turn pixels on and off. This is done by a combination of single clicks and dragging. A single click on a grid square will invert the pixel in that square. You can also click on a grid square and then drag over other grid squares. Whatever the color of the pixel in the first grid square was after the initial click, all the squares dragged over will turn the same color. This enables you to turn whole

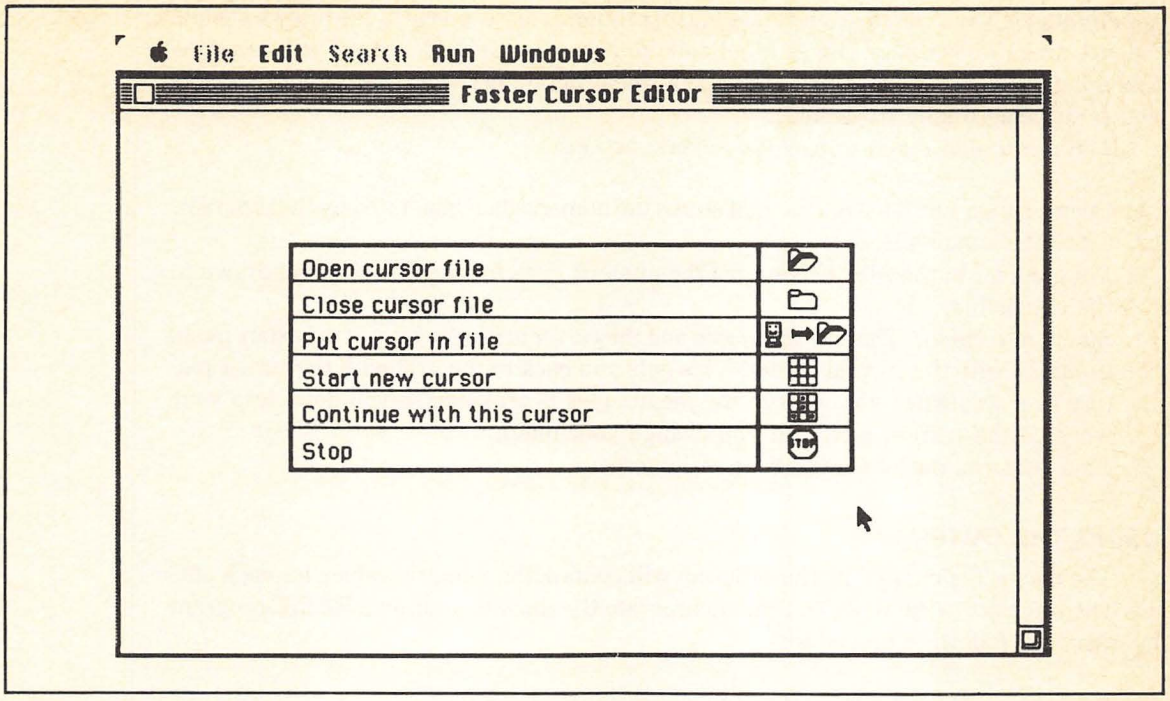


Fig. 22-2. The Cursor Editor menu. The user clicks on the pictures.

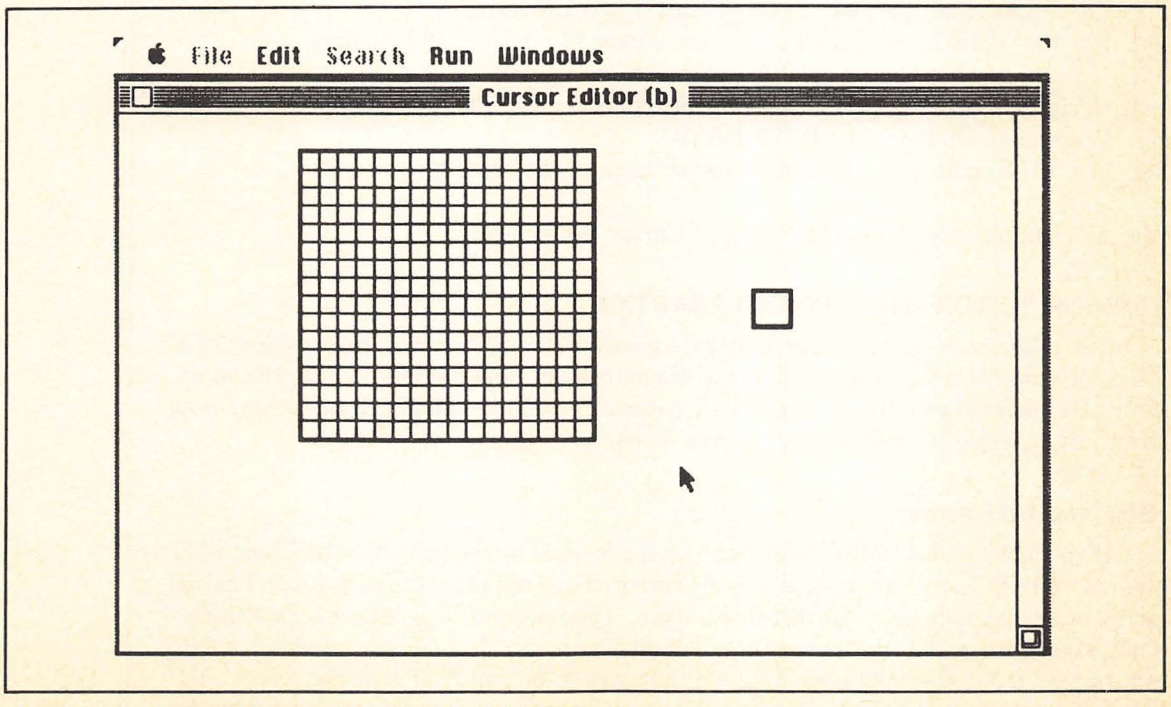


Fig. 22-3. The Cursor Editor drawing grid and preview rectangle.

areas white or black with a single click-drag operation. The response of the program while you are dragging is not as fast as MacPaint, so if you drag too fast, there will be pixels that are missed. If, however, you are patient, the program performs well enough to keep you from being totally frustrated.

The menu allows you to perform several actions:

1. Open a cursor file. This is a file that stores the numeric data generated by the program.
2. Close the cursor file.
3. Put a cursor in the file. This writes the numeric data for the cursor being drawn to the cursor file.
4. Start a new cursor. This clears the grid and the cursor array so that you may start fresh.
5. Continue with the current cursor. This gets you back to the grid with the cursor picture as it was when you went to the menu. This is provided so you don't lose your work on the current picture if you change your mind.
6. Stop gets you out of Cursor Editor altogether.

USING THE OUTPUT

The cursor file created by the program will contain the numeric values for each cursor that you elect to put in it. You can incorporate the file into your own BASIC program by using the following procedure:

1. Run Cursor Editor and create your cursors and cursor file.
2. Exit the program back to BASIC.
3. Chose **New** from the **File** menu to start a new program.
4. Use the MERGE command to put the cursor file in the LIST window.
5. Use the editor to insert DATA statements.
6. Declare an array or arrays for your program to store the data in.
7. Write commands to READ the DATA.
8. Use PUT commands to display the pictures on the screen.

For an example, just study the listing of Cursor Editor itself.

CURSOR EDITOR: THEORY AND PRACTICE

The program is called Cursor Editor because a Macintosh cursor lives inside a 16 × 16 pixel square. On the other hand, in the Macintosh developer's Bible, *Inside Macintosh*, an icon is defined as a 32 × 32 pixel bit pattern. I decided not to call the program an Icon Editor, since it really doesn't create Icons.

GET and PUT arrays

If you have read the MS-BASIC Interpreter manual on the subject of GET and PUT arrays, you will know that to store a single cursor image requires 36 bytes. An INTEGER array of 18 elements provides that much room. The program thus declares: **DIM Cursor (17)**. The question that the manual does not answer is this. How do the numbers in **Cursor** correspond to the bits in the picture it produces when you PUT it on the screen? The answer is simple but perhaps tedious and requires a knowledge of binary arithmetic for

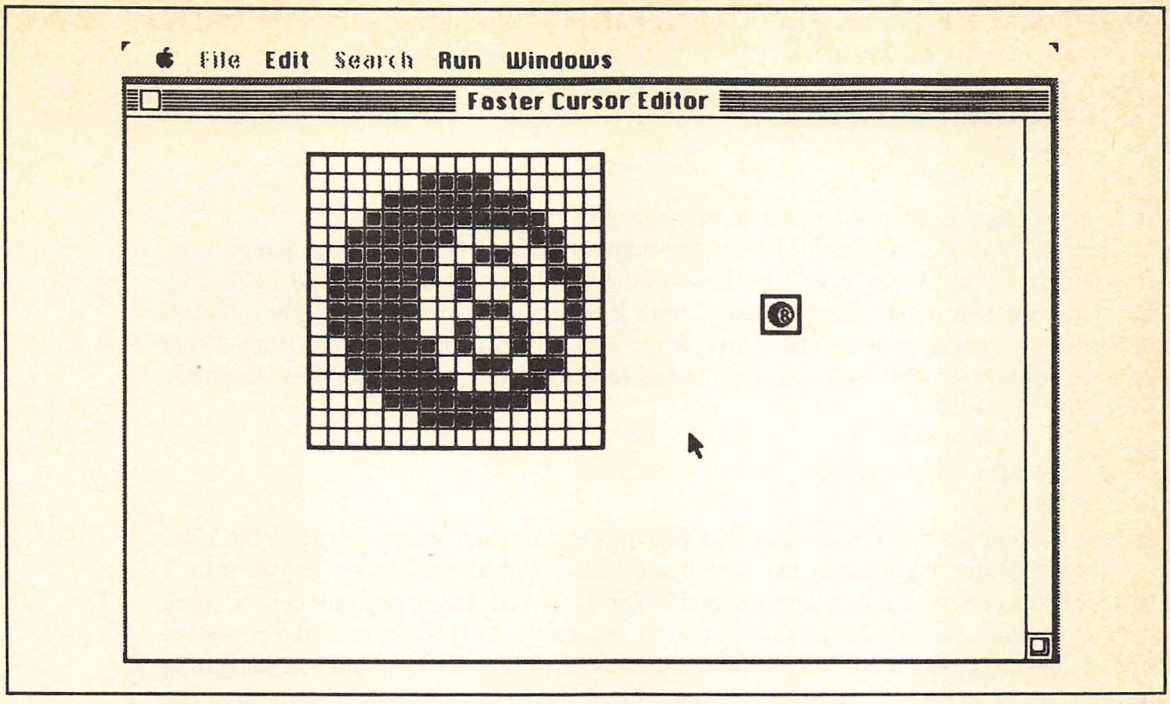


Fig. 22-4. The Cursor Editor in action—a typical cursor.

full understanding. At the risk of boring those of you who already know binary and losing those of you who do not, let me try to give a crash course.

Binary Representation

Cursors, Icons, Fat Bits pictures, and so on are all examples of what are often referred to as *bit images*. A *bit* is the representation of a binary digit. Binary digits appear in binary numerals. If you wish to fully understand bit images, you must understand binary numerals.

Binary numerals aren't different numbers. They are merely the same old numbers we know and love in a different set of clothes. This involves the use of *powers of two*:

$2^0 = 1$ $\qquad$ $2^4 = 16$

$2^1 = 2$ $\qquad$ $2^5 = 32$

$2^2 = 4$ $\qquad$ $2^6 = 64$

$2^3 = 8$ $\qquad$ $2^7 = 128$

and so on. Any number can be expressed by adding together powers of two. Moreover, there is precisely one way to do this for any given number. People say that the representation is *unique*. This is not true for any old method of representing numbers. Just to give

one celebrated example, consider the number 1729. This number may be expressed in two different ways as the sum of cubes (third powers) of other numbers:

$$1729 = 1728 + 1 = 12^3 + 1^3$$
$$1729 = 1000 + 729 = 10^3 + 9^3$$

It turns out that 1729 is the smallest number with this property.

Back to binary . . . In Cursor Editor, the numbers being manipulated are integers between 0 and 65535. Actually, BASIC thinks of them as being between −32768 and 32767, but that is another story. The way that Cursor Editor manipulates integers, the question of positive and negative never arises. In order to express any number between 0 and 65535 as a sum of powers of 2, you potentially need to use powers of two from the 0th power to the 15th power:

$2^0 = 1$ up to $2^{15} = 32768$.

In fact, the number 65535 is the sum of all the powers of two from the 0th up to the 15th.

Now consider any number between 0 and 65535. Put all the powers of two from 1 to 32768 in a lineup with 0 at the right and 32768 at the left. Given any number, N, think of it as walking past all the powers of two in sequence. As it walks past the reception line of powers of two, it will leave those that it needs to express itself in binary standing in their places in line. It will dismiss those that it doesn't need. Now imagine that the lineup that results is symbolized by putting a 1 in all the places where powers of two remain and putting a 0 in all the places where powers of two were dismissed. The resulting "string" of 0s and 1s is called the *binary numeral* for N. By tradition, any leading 0s are omitted from this string of 0s and 1s.

Take a simple example. Let N = 28. Line up all the powers of two. N starts walking down the line. Two to the zeroth power is first in line. N walks up and says: "Hello, how are you. How are the wife and all the little bits?" and dismisses two to the zeroth power, since 1 is not needed in order to express 28. Next in line is two to the first power. N is equal courteous to this power, but it is also not needed so it wanders off to play binary video games. Then N comes to two to the second power. N definitely needs 4, so two to the second power sticks around. This continues down the line until all the powers have been visited. Remaining in line are 2^2, 2^3, 2^4. All the others have gone their merry way. The binary numeral for 28 is: 11100. If you followed that, you are well on your way to understanding binary numerals. Pick a few numbers at random yourself and see if you can express them in binary.

Bits, and Bit Images

The 1s and 0s in a binary numeral are referred to as *bits*. Pixels in a Cursor Editor style picture can be made to correspond to binary numerals. Think of the white pixels as 0s and the black pixels as 1s. Assembling them in some orderly fashion could produce binary numerals. Heres how it works in the Cursor Editor program.

Figure 22-5 pictures the relationship between a single BASIC integer value and a row of the cursor grid. The pixels in the row correspond to bits in the integer with pixels at the right of the row corresponding to the smaller powers of two and pixels at the left of

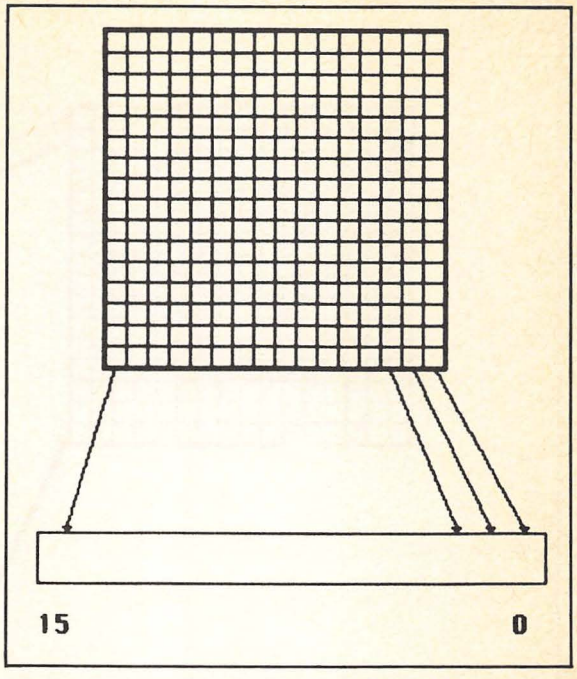


Fig. 22-5. The correspondence between the bits in a grid row and an entry in the Cursor array.

the row corresponding to the higher powers. Figure 22-6 pictures the relationship between the Cursor array and the entire cursor grid. Each row of the grid is represented by one integer in the Cursor array.

Keeping the Cursor up to Date

The integers in the Cursor array must form an accurate image of the picture being drawn in the grid on the screen. To do this it is necessary to change individual bits in the numbers stored in Cursor. The question then becomes "which bits in which numbers?" This question is answered below. The program determines the row and column number of each grid square as it is clicked. Then it decides which bit to change as follows.

The row number clicked tells which integer in Cursor to change. Since the first two entries in Cursor represent the height and width of the image being formed, the first entry that represents a grid row is Cursor (2). Now Cursor (2) represents row number 0 of the grid, Cursor (3) represents row number 1 of the grid, and so on. In general, then, Cursor (ROW +2) represents row number ROW of the grid.

Which bit to change is determined by the column number calculated and stored in the variable COL. The numbering of columns is backwards from the numbering of the bits. The 0th column in the grid corresponds to bit number 15, or the 15th power of 2, in the integer. The program must be careful to flip this around before changing individual bits in the integers stored in Cursor. So just how are the individual bits altered?

Cursor Editor uses what are known as *bit-manipulation* operations. These operations use the powers of two themselves written in a notation called *hexadecimal*. Each power of two is represented by a separate hexadecimal constant. The program stores these constants in an array called MASKS. The name MASKS is frequently used by programmers

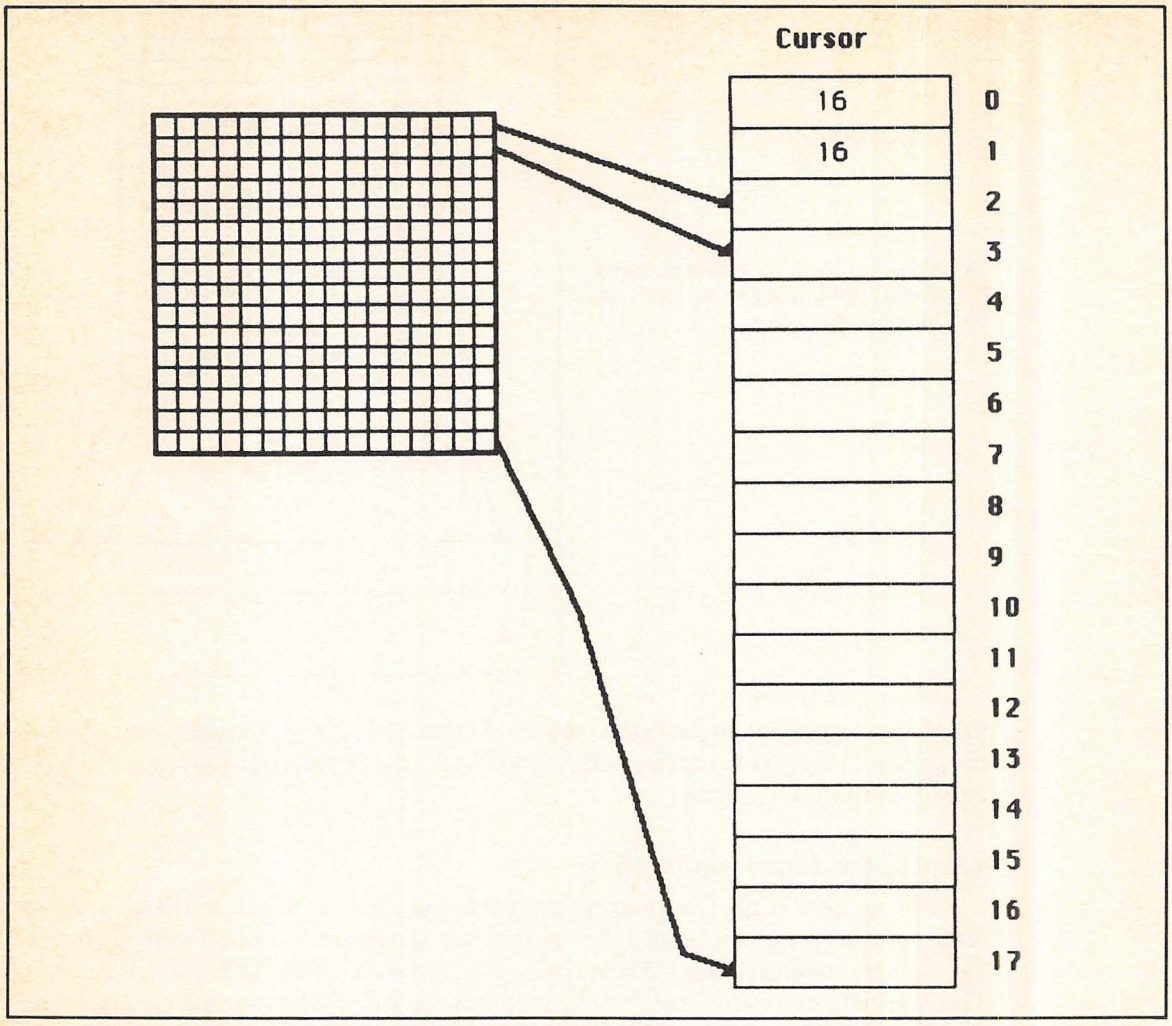


Fig. 22-6. The correspondence between the elements of the Cursor array and the rows of the grid display.

to refer to the use of the powers of two in this fashion. The values are stored so that the backwards numbering of the columns compared to the numbering of the bits is taken into account.

In order to change a bit from 1 to 0 or vice versa the XOR operator is used. In order to force a bit to be a 1 the OR operation is used. In order to force a bit to be a 0, a combination of operations is used:

```
Cursor (ROW + 2) = Cursor (ROW + 2) AND (NOT MASKS (COL))
```

That's "all" there is to it! The program accomplishes all this in the subroutines named toggle and adjust. Later in the chapter, I will discuss how the program decides which operation to use.

MORE MOUSE TECHNIQUES

Cursor Editor uses the MOUSE function extensively. This is a good chance for you to extend your mouse programming repertoire. Some of the operations performed are the same as before, but at least one technique is new. Also, the drag operation involved is more ambitious than any included in earlier programs.

Which Rectangle?--Revisited

The cursor grid shown in Figure 22-3 contains 256 different rectangles. When the user clicks or clicks and drags, the program must constantly calculate which rectangle the mouse is in. Knowing which rectangle is important for two purposes:

- To know which screen rectangles to change
- To know which Cursor array bits to change

Earlier, in the Rose Patterns program, I discussed one technique for distinguishing between several rectangles. That technique is also used in Cursor Editor in the handling of the program menu. For the detection of grid squares, I have a new one for you.

Each square in the grid is 10 pixels on a side. Think about any point inside the grid. Think of the upper left corner of the grid as being the point with coordinates (0,0). Then, think of the total grid as 16 rows high and 16 columns wide. Assume that the rows and columns are numbered beginning with 0. If you take the coordinates of any point (X, Y) inside the grid, you can get the row and column number of that point's grid square by dividing each coordinate by 10 and ignoring the remainder. More exactly:

Grid row number = Y divided by 10 (throw away the remainder)
Grid column number = X divided by 10 (throw away the remainder)

For example, the point (47,123) is in column 4 and row 12.

MS-BASIC provides an operator that performs the operation just described. It is the \ operator, so that 47\10 = 4 and 123\10 = 12. The Cursor Editor program uses this calculation to figure out which grid square the mouse pointer is in. In order to convert to the "local coordinate system" of the grid, the screen or global coordinates of the upper left corner of the grid: (X0, Y0) are subtracted from the screen coordinates of the mouse pointer:

```
X = MOUSE(5) - X0 : Y = MOUSE(6) - Y0
```

The row and column numbers are then calculated with the '\' operator:

```
COL = X \ 10 : ROW = Y \ 10
```

Since the ROW and COL values are used directly in updating the Cursor array, a rectangle map such as the one used in Rose Patterns is unnecessary. This is a much more efficient technique and the speed of the Cursor Editor is greatly enhanced by it.

A Dragging Operation

The main loop of Cursor Editor waits for each mouse click and then performs the following actions:

- It detects the ROW and COL of the square clicked as just described.
- It checks for a click outside the grid (ROW or COL > = 16)
- If a click happens outside the grid, it invokes the program menu.
- If a grid square is clicked, it adjusts the pixel inside accordingly.

These actions occur first for the square clicked on. The pixel in that square is *inverted.* The action doesn't stop there, however. There is an inner loop that repeats these actions as long as the mouse button is held down. There is a difference in the way the pixels of the grid are adjusted after the click but during the drag. This depends on what happened to the first pixel. If it turned black, then all subsequent pixels turn black—or stay black if they already were black to begin with. Likewise, if the first pixel turns white, then all pixels dragged over turn white. How this is accomplished is discussed next.

Bits Drag All the Same Color

The drag operation introduces two new problems:

How to remember what happened to the first pixel.
How to tell the difference between the first and other pixels.

This is the problem of deciding which bit manipulation operation to use when altering the values in Cursor. The program examines the first bit altered during a drag operation. It accomplishes this with the AND operation: **nowbit = Cursor (ROW + 2) AND MASKS (COL)**. It then sets a variable called **parity** to represent logically whether the resulting pixel changed to black or to white. If the value stored in **parity** is – 1 (True), the remaining pixels in the drag all turn black. If the value stored in parity is 0 (False), all pixels dragged over turn white.

CHALLENGES

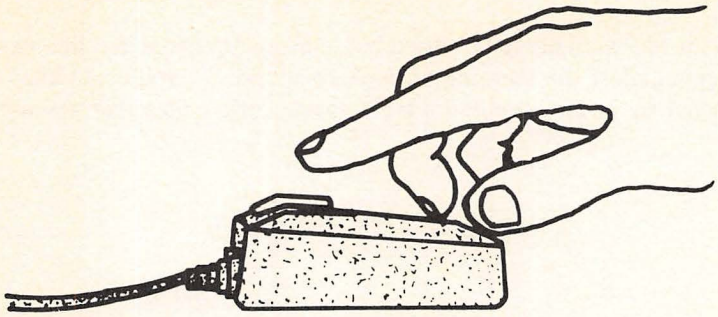
Cursor Editor is usable as it stands, but there are a number of enhancements you might undertake. Most are aimed at making the program more useful in the process of developing your own applications.

- Change the window to a single line border window.
- Enlarge the size of the grid to 32x32 to allow full Macintosh icons.
- Have the program generate the BASIC DATA statements directly. Right now you have to edit the output to insert the DATA statements. If you don't need the name of each cursor or icon in your application, you may remove it from the output.
- Allow the program to recall a cursor that has already been put into the file for further editing. This could involve turning the cursor file into a random access file. Or you could store all the cursors in memory until Stop was chosen from the program menu.

Then have the quit subroutine write out the data for all the cursors created to the cursor file and close it.

- Add a feature to allow the program to read cursor files created during a prior program run.
- Allow cursor files not only to be edited but to be manipulated in other ways. For example, you could allow the merging of two cursor files, or you could allow a cursor file to be created by including cursors from various other files that already exist.

Chapter 23



Mouse Kaleidoscope

I have always been fascinated by kaleidoscopes. As a kid, they were among my favorite toys. The principle of the kaleidoscope is simple. Take a pattern of any sort and reflect it about one or more axes to produce a strongly repetitive and symmetrical pattern. This principle is followed in Mouse Kaleidoscope to produce symmetric graphical patterns on the Macintosh screen. They aren't in color. But the program user has control over their generation.

While the program was written purely for entertainment, it illustrates a number of points about writing BASIC applications.

THE PROGRAM LISTING

This program listing contains the symbol ✥ . It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✥ ; for example, if the listing shows

```
IF XO < X THEN     ✥
   inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN  inside = 0 : RETURN
```

Program Listing 23-1
Application #2 - Mouse Kaleidoscope

```
'-----------------------------------------------------------------
' Mouse Kaleidoscope
'
' Written by
' Richard C. Vile, Jr.
' January 22, 1985
'-----------------------------------------------------------------

DEFINT a-z
max.spoor = 1024
mac512k = 0
DIM spoorx (max.spoor), spoory (max.spoor), re(3)
DIM pat (3)
IF mac512k THEN DIM icon.buf (7500), icon.bufc (7500)
DIM patterns (35)
DIM the.pattern (3), masks (7), bytes (8)
IF mac512k THEN OPEN "Robby" FOR INPUT AS 1 :  +
   icon$ = INPUT$ (LOF (1), 1) :CLOSE #1
spoor.length = 32
delay = 7
repeat.it = -1
pat.clr = -1
lf = 5 : tp = 40 : rg = 300 : bt = 335
square = 1
round = 2
oval = 3
icon = 4
shape = oval
GOSUB get.patterns
GOSUB get.mask.data
GOSUB makemenus

MENU ON
ON MENU GOSUB menus
GOSUB makewindow
CALL PENMODE (want.mode)
CALL PENPAT (VARPTR (patterns (0)))
GOSUB restart
GOSUB coords
MOUSE ON
ON MOUSE GOSUB check.mouse
GOSUB click
CLS
```

```
WHILE -1
  GOSUB track
  GOSUB coords
WEND

menus:
  MENU OFF
  menunumber = MENU (0)
  menuitem   = MENU (1)
  ON menunumber GOSUB shapes, values, set.pattern, control, drawing
  MENU ON
  MENU
  RETURN

shapes:
  MENU 1, shape, 1
  shape = menuitem
  MENU 1, shape, 2
  CLS
  RETURN

values:
  ON menuitem GOSUB leng, spoor.delay, size
  RETURN

leng:
  title$ = STR$ (spoor.length)
  GOSUB box
  spoor.length = VAL (EDIT$ (1))
  IF delay >= spoor.length THEN leng
  IF leng >= 1024 THEN leng
  GOSUB clr
  RETURN

spoor.delay:
  title$ = STR$ (delay)
  GOSUB box
  delay = VAL (EDIT$ (1))
  IF delay >= spoor.length THEN spoor.delay
  IF delay >= 1024 THEN spoor.delay
  GOSUB clr
  RETURN

size:
  title$ = STR$ (side)
  GOSUB box
  side = VAL (EDIT$ (1))
```

```
  hs  = side / 2
  qs  = side / 3
  GOSUB clr
  RETURN

box:
  CLS
  EDIT FIELD 1, title$, (10,10) - (100, 24)
  BUTTON 1, 1, "Ok", (105, 10) - (130, 26)
  BUTTON 2, 1, "Up", (135, 10) - (160, 26)
  BUTTON 3, 1, "Down", (165, 10) - (210, 26)
  satisfied = 0
  WHILE NOT satisfied
    d = DIALOG (0)
    btn = 0
    IF d = 1 THEN btn = DIALOG (1)
    IF (d=6) OR (btn = 1) THEN satisfied = -1
    IF (btn = 2) THEN EDIT FIELD 1,  +
      STR$ (VAL (EDIT$ (1)) + 2), (10,10) - (100, 24)
    IF (btn = 3) THEN EDIT FIELD 1,  +
      STR$ (VAL (EDIT$ (1)) - 2), (10,10) - (100, 24)
  WEND
  RETURN

clr:
  EDIT FIELD CLOSE 1
  BUTTON CLOSE 1
  BUTTON CLOSE 2
  BUTTON CLOSE 3
  RETURN

set.pattern:
  IF menuitem = 9 THEN GOSUB build.new.pattern : CLS : RETURN
  IF menuitem = 10 THEN RESTORE : GOSUB get.patterns : RETURN
  MENU 3, current.pattern, 1
  CALL PENPAT (VARPTR (patterns((menuitem-1)*4)))
  MENU 3, menuitem, 2
  current.pattern = menuitem
  CLS
  RETURN

coords:
  IF wants.clear THEN GOSUB restart
  IF spot = spoor.length-1 THEN full = -1
  spot = ((spot+1) MOD spoor.length)
  IF full AND repeat.it THEN RETURN
  ms = MOUSE (0)
```

```
  spoorx(spot) = MOUSE (1) : spoory(spot) = MOUSE (2)
  RETURN

check.mouse:
  IF ABS (MOUSE (0)) >= 2 THEN wants.clear = -1
  IF ABS (MOUSE (0)) = 1 THEN WHILE MOUSE (0) <> 0 : WEND
  RETURN

track:
  now = spot : blot = 0
  GOSUB plot
  IF (spot < delay) AND NOT full THEN RETURN
  rubout = spot - delay
  IF (rubout < 0) THEN rubout = rubout + spoor.length
  now = rubout : blot = -1
  GOSUB plot
  RETURN

plot:
  x = spoorx (now) : y = spoory (now)
  ON shape GOSUB plot.rect, plot.oval, plot.round, plot.icon
  RETURN

plot.rect:
  re(0) = y - hs : re(1) = x - hs : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))
  re(0) = wide - x - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))
  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))
  re(0) = high - y - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))
  re(0) = x - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
```

```
    ELSE CALL PAINTRECT (VARPTR (re(0)))

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASERECT (VARPTR (re(0)))  +
    ELSE CALL PAINTRECT (VARPTR (re(0)))
  RETURN

plot.round:
  re(0) = y - hs : re(1) = x - hs : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)
  re(0) = wide - x - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)
  re(0) = high - y - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)
  re(0) = x - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEROUNDRECT (VARPTR (re(0)),qs,qs)  +
    ELSE CALL PAINTROUNDRECT (VARPTR (re(0)),qs,qs)
  RETURN

plot.oval:
  re(0) = y - hs : re(1) = x - hs : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))  +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))  +
```

```
    ELSE CALL PAINTOVAL (VARPTR (re(0)))
  re(0) = wide - x - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))   +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))   +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))
  re(0) = high - y - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))   +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))

  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))   +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))
  re(0) = x - hs : re(2) = re(0) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))   +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))
  SWAP re(0), re(1) : re(2) = re(0) + side : re(3) = re(1) + side
  IF blot THEN CALL ERASEOVAL (VARPTR (re(0)))   +
    ELSE CALL PAINTOVAL (VARPTR (re(0)))
  RETURN

plot.icon:
  IF NOT mac512k THEN RETURN
  IF blot THEN PUT (x,y)-(x+side,y+side), icon.bufc, AND    +
    ELSE PUT (x,y)-(x+side,y+side), icon.buf
  IF blot THEN PUT (y,x)-(y+side, x+side), icon.bufc, AND    +
    ELSE PUT (y,x)-(y+side, x+side), icon.buf
  IF blot THEN PUT (wide-x, y)-(wide-x+side, y+side), icon.bufc, AND    +
    ELSE PUT (wide-x, y)-(wide-x+side, y+side), icon.buf
  IF blot THEN PUT (y, wide-x)-(y+side, wide-x+side),icon.bufc, AND    +
    ELSE PUT (y, wide-x)-(y+side, wide-x+side),icon.buf
  IF blot THEN PUT (x, high-y)-(x+side, high-y+side), icon.bufc, AND    +
    ELSE PUT (x, high-y)-(x+side, high-y+side), icon.buf
  IF blot THEN PUT (high-y, x)-(high-y+side, x+side), icon.bufc, AND    +
    ELSE PUT (high-y, x)-(high-y+side, x+side), icon.buf
  IF blot THEN PUT (wide-x, high-y)-(wide-x+side, high-y+side), icon.bufc, AND    +
    ELSE PUT (wide-x, high-y)-(wide-x+side, high-y+side), icon.buf
  IF blot THEN PUT (high-y, wide-x)-(high-y+side, wide-x+side), icon.bufc, AND    +
    ELSE PUT (high-y, wide-x)-(high-y+side, wide-x+side), icon.buf
  RETURN

build.new.pattern:
  CLS
```

```
  cancel = 0
  GOSUB make.grid
  CALL MOVETO (16, 265)
  PRINT "Draw a pattern in the grid"
  BUTTON 75, 1, "Ok", (16,175)-(41,195)
  BUTTON 76, 1, "Cancel", (16, 200) - (81, 220)
  BUTTON 77, 1, "Preview", (16, 225) - (100, 245)
  GOSUB grid.dialog
  IF cancel THEN GOSUB clean.grid : RETURN
  GOSUB record.pattern
  GOSUB clean.grid
  GOSUB pattern.dialog
  RETURN

record.pattern:
  FOR i = 1 TO 8 : bytes (i) = 0 : NEXT i
  FOR row = 1 TO 8
    FOR col = 1 TO 8
      btn = col*8 + row
      IF BUTTON (btn) = 2 THEN GOSUB bit.on
    NEXT col
  NEXT row
  FOR j = 1 TO 8
    k = INT ((j-1)/2)
    IF (j MOD 2) = 0 THEN POKE VARPTR (the.pattern (k))+1, bytes (j)  +
      ELSE POKE VARPTR (the.pattern (k)), bytes (j)
  NEXT j
  RETURN

show.pattern:
  re (0) = 0 : re (1) = 175 : re (2) = 350 : re (3) = 500
  CALL PENPAT (VARPTR (the.pattern (0)))
  CALL PAINTRECT (VARPTR (re (0)))
  RETURN

bit.on:
  bytes (row) = bytes (row) + masks (col-1)
  RETURN

grid.dialog:
  GOSUB clear.dialog
  done = 0
  WHILE NOT done
    d = DIALOG (0)
    IF d = 1 THEN GOSUB check.buttons
    IF d = 6 THEN done = -1
  WEND
```

```
  RETURN

pattern.dialog:
  GOSUB clear.dialog
  CLS
  FOR btn = 2 TO 8
    BUTTON btn, 1, "Pattern"+STR$(btn), (10, btn*15) - (90, (btn+1)*15), 3
  NEXT btn
  PRINT "Which pattern do you wish to replace?"
  done = 0
  WHILE NOT done
    d = DIALOG(0)
    IF d = 1 THEN which = DIALOG(1)-1 : done = -1
    IF d = 6 THEN which = 1 : done = -1
  WEND
'  PRINT "Picked ";which
  FOR i = 0 TO 3
    patterns(which*4 + i) = the.pattern(i)
'    PRINT which*4 + i;
  NEXT i

  MENU 3, current.pattern, 1
  current.pattern = which+1
  MENU 3, current.pattern, 2

  FOR btn = 2 TO 8
    BUTTON CLOSE btn
  NEXT btn
  CALL PENPAT (VARPTR (the.pattern(0)))
  RETURN

check.buttons:
  btn = DIALOG(1)
  IF btn = 75 THEN done = -1: RETURN
  IF btn = 76 THEN done = -1 : cancel = -1: RETURN
  IF btn = 77 THEN GOSUB preview : RETURN
  IF BUTTON(btn) = 2 THEN BUTTON btn, 1 ELSE BUTTON btn, 2
  RETURN

preview:
  GOSUB record.pattern
  GOSUB click
  GOSUB show.pattern
  GOSUB click
  CLS
  RETURN
```

```
make.grid:
  FOR col = 1 TO 8
    FOR row = 1 TO 8
      BUTTON row*8 + col, 1, "",   +
        (row*16,col*16)-(((row+1)*16)-2, ((col+1)*16)-2), 3
    NEXT row
  NEXT col
  RETURN

clean.grid:
  FOR btn = 9 TO 77
    BUTTON CLOSE btn
  NEXT btn
  RETURN

clear.dialog:
  WHILE DIALOG (0) <> 0 : WEND : ' Clear dialog queue of stray events
  RETURN

control:
  ON menuitem GOSUB pattern.clear, repeat.kaleidoscope, quit
  RETURN

pattern.clear:
  IF pat.clr THEN pat.clr = 0 ELSE pat.clr = -1
  IF pat.clr THEN MENU 4, 1, 2 ELSE MENU 4, 1, 1
  RETURN

repeat.kaleidoscope:
  IF repeat.it THEN repeat.it = 0 ELSE repeat.it = -1
  IF repeat.it THEN MENU 4, 2, 2 ELSE MENU 4, 2, 1
  RETURN

quit:
  WINDOW CLOSE 1
  MENU RESET
  CHAIN "Master Menu"
  END

drawing:
  ON menuitem GOSUB cpy, invert, not.cpy, not.invert
  CALL PENMODE (want.mode)
  RETURN

cpy:
  want.mode = 8
```

```
  MENU 5,1,2
  IF want.draw <> 1 THEN MENU 5,want.draw,1
  want.draw = 1
  RETURN

invert:
  want.mode = 10
  MENU 5,2,2
  IF want.draw <> 2 THEN MENU 5,want.draw,1
  want.draw = 2
  RETURN

not.cpy:
  want.mode = 12
  MENU 5,3,2
  IF want.draw <> 3 THEN MENU 5,want.draw,1
  want.draw = 3
  RETURN

not.invert:
  want.mode = 14
  MENU 5,4,2
  IF want.draw <> 4 THEN MENU 5,want.draw,1
  want.draw = 4
  RETURN

makemenus:
  FOR mn=1 TO 5
    MENU mn,0,0,""
  NEXT mn

  MENU 1,0,1,"Shape"
  MENU 1,1,1 + ABS(shape = 1),"Square"
  MENU 1,2,1 + ABS(shape = 2),"Round"
  MENU 1,3,1 + ABS(shape = 3),"Oval"
  IF mac512k THEN MENU 1,4,1 + ABS(shape = 4),"Picture"

  MENU 2,0,1,"Values"
  MENU 2,1,1,"Length"
  MENU 2,2,1,"Delay"
  MENU 2,3,1,"Size"

  MENU 3,0,1,"Patterns"
  MENU 3,1,2,"Black"
  MENU 3,2,1,"Pattern2"
  MENU 3,3,1,"Pattern3"
```

```
    MENU 3,4,1,"Pattern4"
    MENU 3,5,1,"Pattern5"

    MENU 3,6,1,"Pattern6"
    MENU 3,7,1,"Pattern7"
    MENU 3,8,1,"Pattern8"
    MENU 3,9,1,"Make a new one..."
    MENU 3,10,1,"Restore Originals"

    MENU 4,0,1,"Control"
    MENU 4,1,2,"Pattern Clear"
    MENU 4,2,2,"Repeat"
    MENU 4,3,1,"Quit"

    MENU 5,0,1,"Drawing"
    MENU 5,1,1,"Copy"
    MENU 5,2,2,"Invert"
    MENU 5,3,1,"Negative Copy"
    MENU 5,4,1,"Negative Invert"
    want.draw = 2
    want.mode = 10
    RETURN

makewindow:
    side = 64 : hs = side / 2 : qs = side / 3
    wide = rg - lf : high = bt - tp
    midx = wide / 2 : midy = high / 2
    WINDOW 1,"Kaleidoscope",(lf,tp)-(rg + 15,bt), 1
    IF mac512k THEN PICTURE (0,0)-(wide-12, high), icon$
    IF mac512k THEN GET (0,0) - (wide-12, high), icon.buf
    RETURN

restart:
    spot = -1
    show = 0
    full = 0
    wants.clear = 0
    CLS
    RETURN

click:
    PRINT "click to continue"
    ms = MOUSE (0)
    WHILE MOUSE (0) = 0 : WEND
    WHILE MOUSE (0) <> 0 : WEND
    RETURN
```

```
get.mask.data:
  FOR i = 0 TO 7
    READ masks(i)
  NEXT i
  RETURN

get.patterns:
  FOR i = 0 TO 31
    READ patterns(i)
  NEXT i
  current.pattern = 1
  RETURN

DATA &hFFFF, &hFFFF, &hFFFF, &hFFFF
DATA &hFFFF, &hF00F, &hF00F, &hFFFF
DATA &hE7C3, &hA518, &h18A5, &hC3E7
DATA &h2050, &h7373, &h3737, &h0502
DATA &hFF99, &h99E7, &hE799, &h99FF
DATA &h7789, &hBFBF, &h779B, &hF8F8
DATA &hBFC0, &hBFBF, &hB0B0, &h0B0B
DATA &hCCCC, &h3333, &hCCCC, &h3333

DATA &H80, &H40, &H20, &H10
DATA &H8, &H4, &H2, &H1
```

PROGRAM CAPABILITIES AND OPERATION

Mouse Kaleidoscope produces dynamic graphic displays using the principle of a kaleidoscope. Each display is started with a single shape such as a square or a circle. Multiple reflections about various axes produce a total of eight copies of each instance of the shape. The location of the first copy is determined by the mouse location. As the mouse cursor is moved, the shapes follow it. After a certain number of these eight-fold shape patterns are produced, the program begins to erase them. The erasing is done in the same order that the original drawing was done. This means that eventually everything that was displayed will be erased. This keeps the display from getting visually cluttered.

You can control the kaleidoscope in various ways. For example, you can set the delay. The delay determines how many shapes will be displayed before any are erased.

KALEIDOSCOPE PRINCIPLES

The kaleidoscopes you had as a child usually had six-fold symmetry. That is, a single pattern covering a sixty degree arc of a circle was replicated to produce a total of six equal copies in a circular pattern surrounding a common center. The replications were achieved with the use of mirrors cemented inside the end of a long tube and aligned at a sixty degree angle to one another.

The patterns produced by the Mouse Kaleidoscope have eight-fold symmetry. It is easier to produce those on a rectangular CRT screen than it would be to produce patterns with six-fold symmetry. Given the coordinates of the mouse, systematic transformations

are carried out to produce a pattern of eight points. A geometric figure is drawn surrounding each of these eight points to produce one stage of each kaleidoscopic display.

The transformations can be described as follows. The window of the program is assumed to have four axes of symmetry. The axes all pass through the center of the window. Two of the axes are coincident with the x and y axes of a rectangular coordinate system. The other two are at forty-five degree angles to the original two. A given point may be thought of as lying in one of four quadrants determined by the vertical and horizontal axes. It is first reflected about the diagonal axis in its quadrant to produce a second point. The pair of resulting points is then reflected about the vertical and horizontal axes to produce four more points. Finally, the pair is reflected about the diagonal axis not in their quadrant to produce a final pair of points, making eight in all.

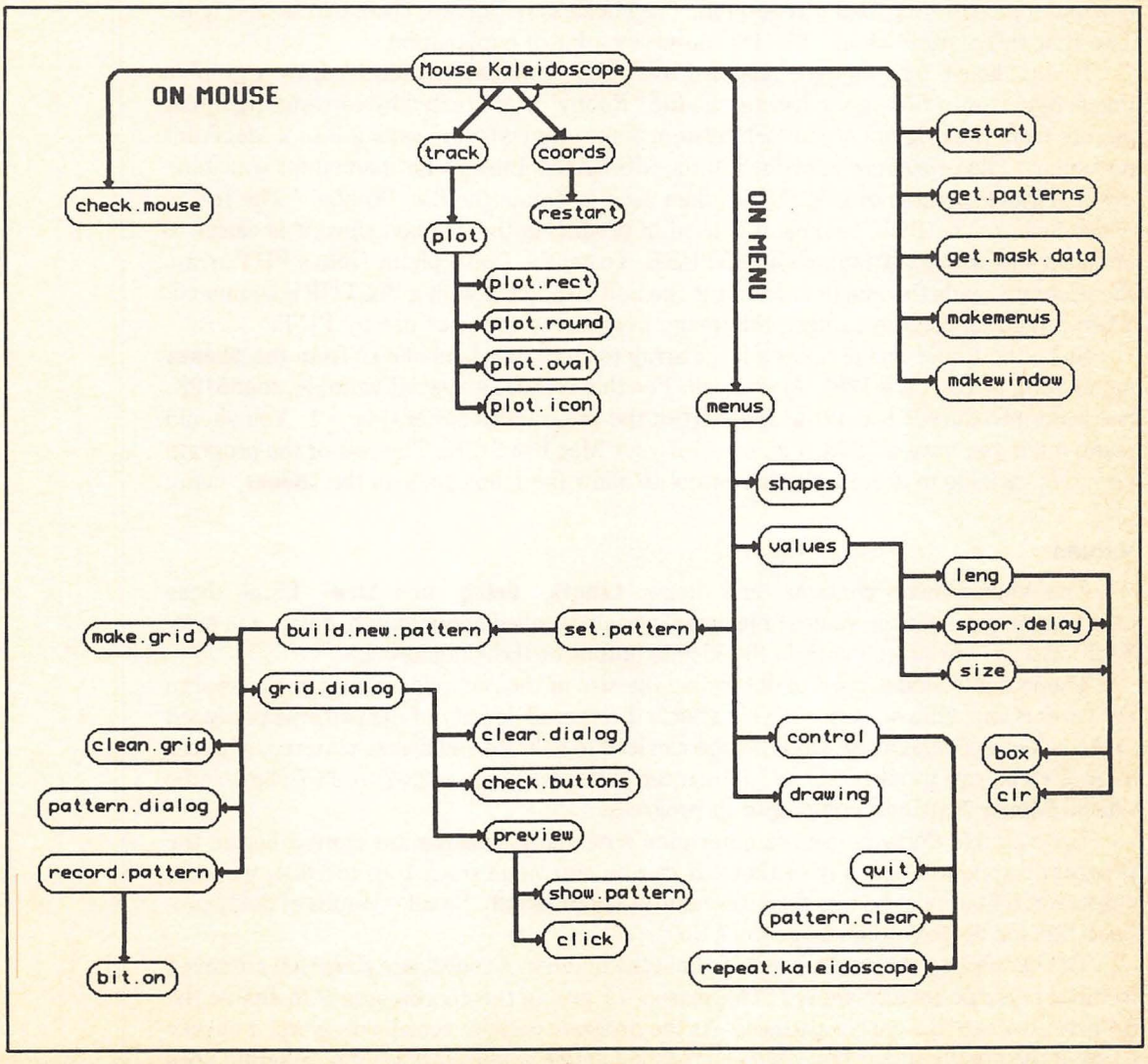


Fig. 23-1. Structure diagram for Mouse Kaleidoscope.

MENUS

Mouse Kaleidoscope is another example of a program that uses a large number of menus. It has a number of menus that provide the user with ways to vary the character of the kaleidoscopes produced. Figure 23-1 presents a structure diagram of the subroutines used in Mouse Kaleidoscope. Use it as a reference while reading the descriptions of the program's behavior and implementation in the remainder of the chapter. The menus are **Shapes**, **Values**, **Patterns**, **Control**, and **Drawing**. I will discuss the possible choices in each.

Shapes

The **Shapes** menu allows you to choose the basic shape that is replicated while the kaleidoscope pattern is being generated. The choices are Square, Oval, Circle, and Icon. The first three are obvious. The last deserves a bit of explanation.

If you choose Icon as the shape, a PICTURE command is used to draw a graphic image read from a file. In the listing, the file "Robby" was created by copying the Robot picture from the original Macintosh System disk scrapbook and saving it in a MacPaint document. The procedure explained in the MS-BASIC Interpreter manual for transferring MacPaint pictures to BASIC was then used to create the file "Robby." The rest is in the listing. The PUT command is used in producing the display, since it is easier to erase images with PUT than with PICTURE. To get the Robot picture into a PUT array, the program reads the image in from the file and displays it with a PICTURE command. Then a GET is used to capture the image in an array for later use by PUT.

Since the Robot image takes a large array to store, the Icon choice from the **Shapes** menu will not run on a 128K Macintosh. For this reason, a special variable, mac512K, has been introduced. It is set at the start of the program to either 0 or −1. You should make it 0 if you have a 128K Mac or −1 if your Mac has 512K. The rest of the program uses this variable to decide whether or not to allow the Icon choice in the **Shapes** menu.

Values

The Values menu contains three items: **Length**, **Delay**, and **Size**. Using these choices, you can alter the values of program variables called spoor.length, delay, and side. Each of these variables controls the visible output of the kaleidoscope.

The variable side is used to determine the size of the rectangle parameter passed to the various Quickdraw routines. This affects the overall density of the patterns produced in the kaleidoscope. A small value of side can lead to delicate and sparse patterns, whereas a large value can produce bolder and denser patterns. Figures 23-2 to 23-7 show some typical Mouse Kaleidoscope output in progress.

The variable delay is used to determine how many patterns are plotted before the program begins to erase any of them. It can be any value from 1 up to 1024, with the constraint that it must be less than the value of spoor.length. Smaller values of delay produce thinner displays than large ones do.

The variable spoor.length is used to limit the number of coordinate pairs that are saved in the arrays spoorx and spoory. The reason for saving the coordinates is to enable the program to erase the earlier patterns. As the program gathers coordinate pairs, it checks to see if the number it has collected has reached spoor.length. If it has, the program does

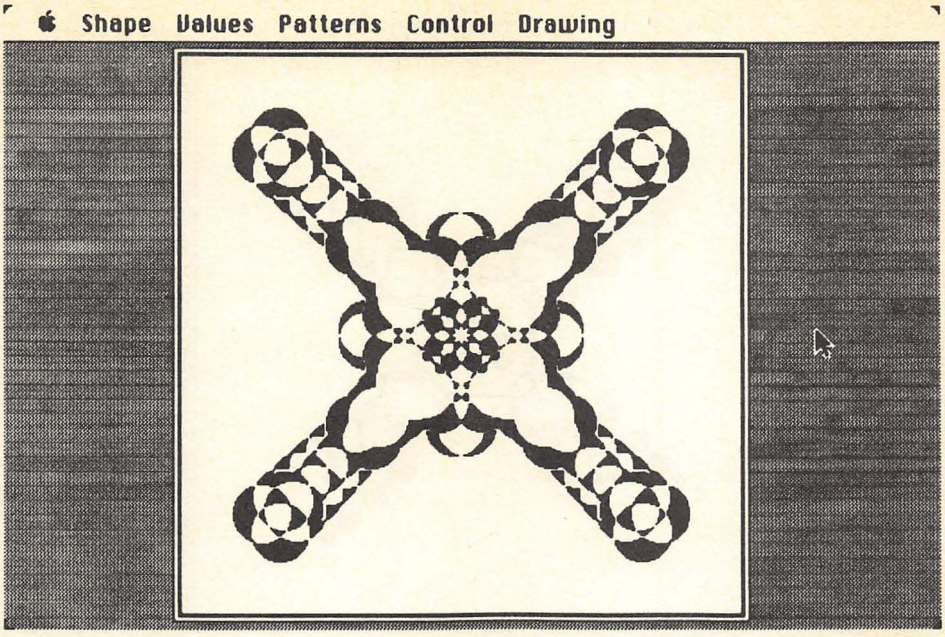


Fig. 23-2. Typical Mouse Kaleidoscope output.

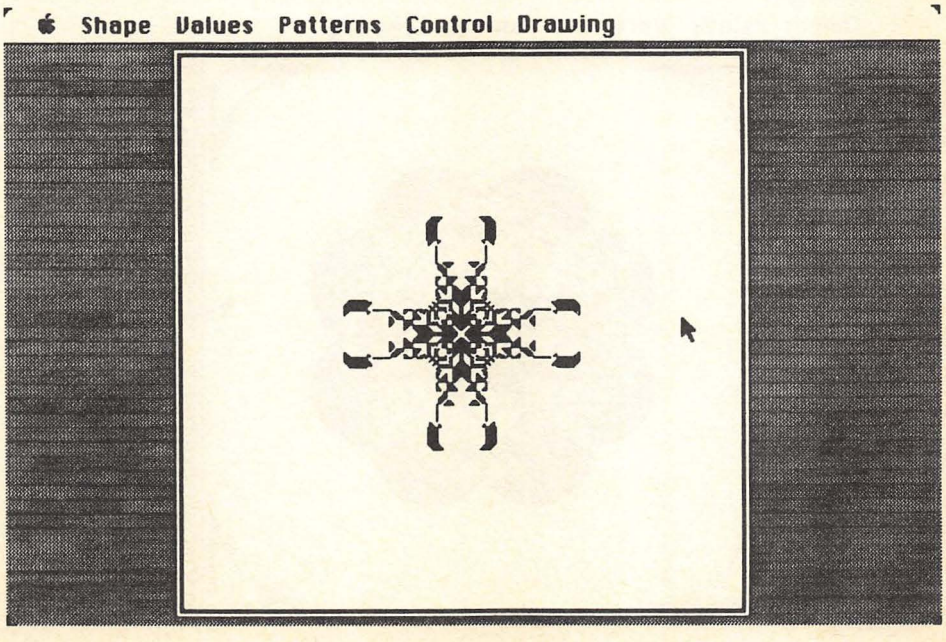


Fig. 23-3. Typical Mouse Kaleidoscope output.

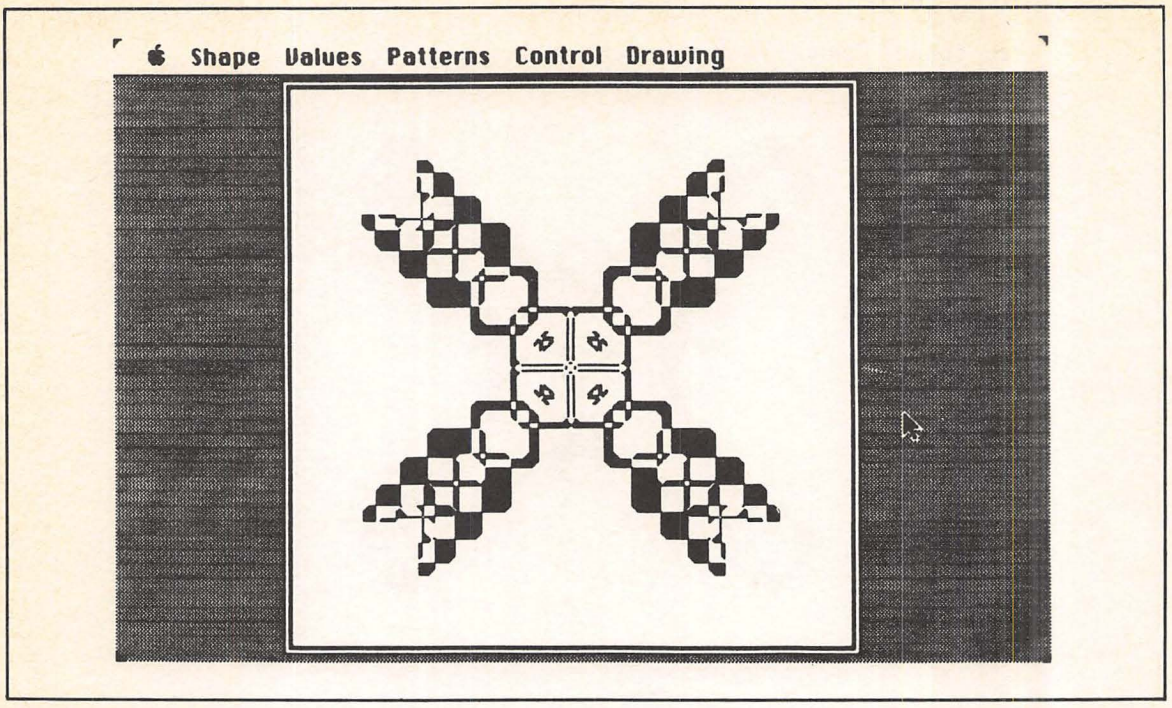


Fig. 23-4. Typical Mouse Kaleidoscope output.

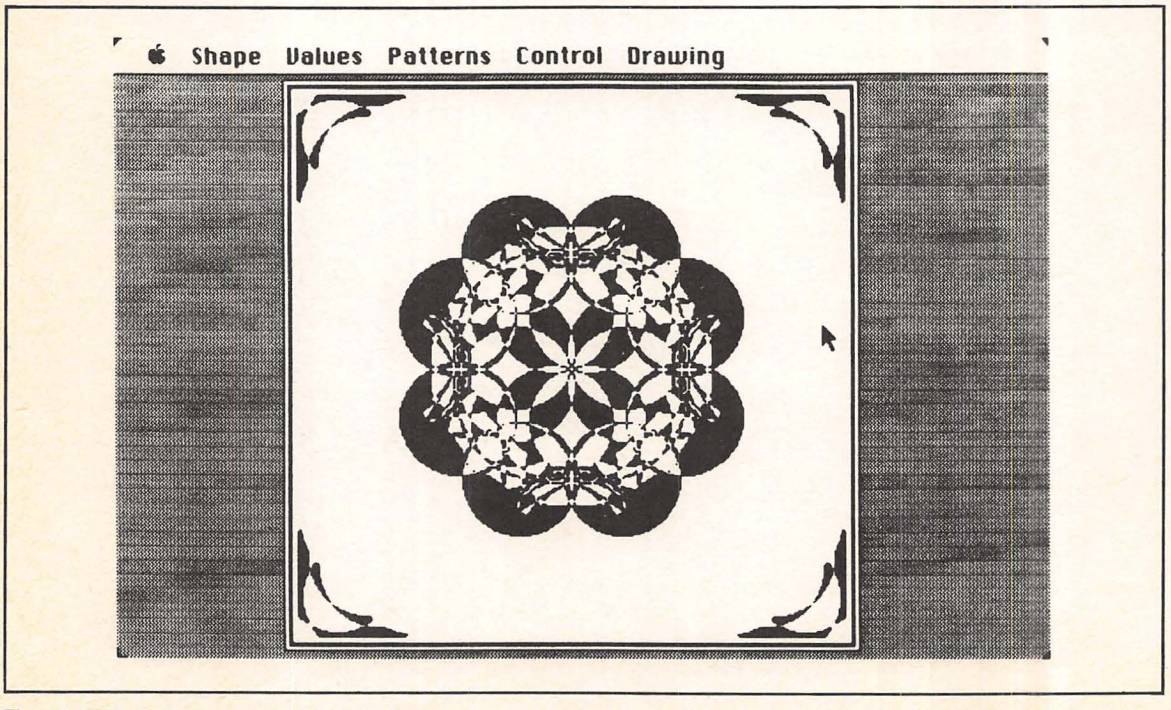


Fig. 23-5. Typical Mouse Kaleidoscope output.

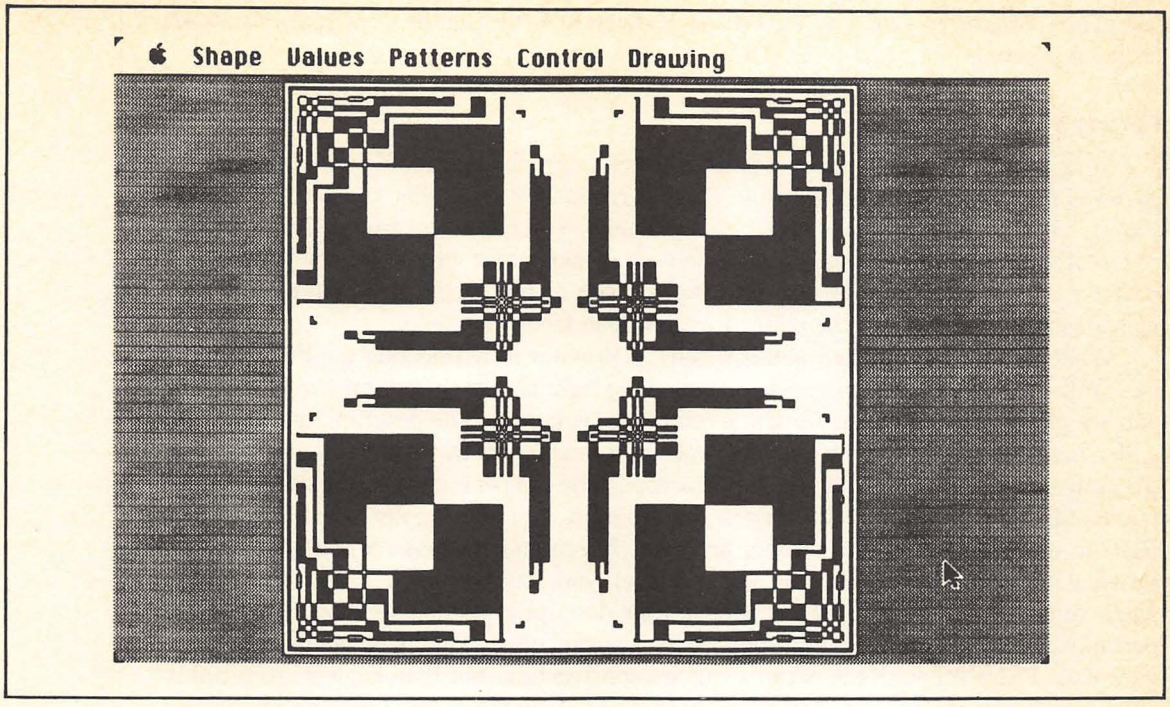


Fig. 23-6. Typical Mouse Kaleidoscope output.

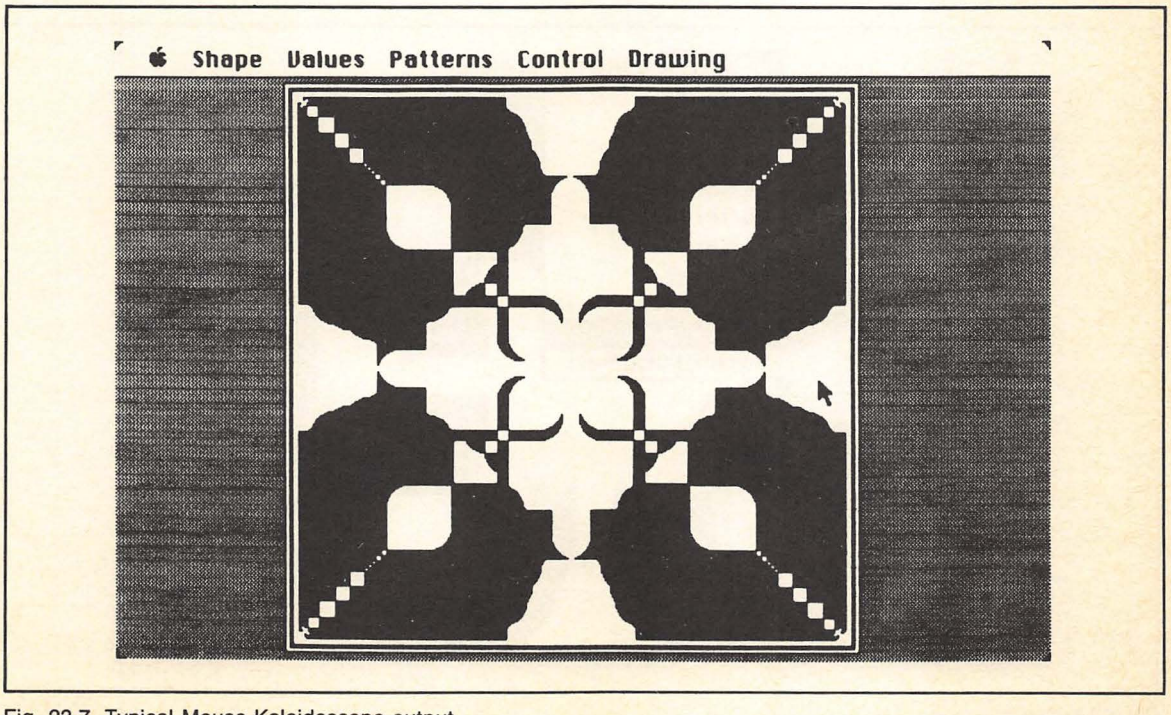


Fig. 23-7. Typical Mouse Kaleidoscope output.

one of two things depending on the setting of the item Repeat in the Control menu. Read on for full details.

Patterns

Mouse Kaleidoscope will use a pattern to fill its graphic shapes if you tell it to. The patterns it uses may be selected from those it reads in or ones that you build yourself. Figure 23-8 shows the choices in the Pattern menu. You can select from black or one of seven other nonblack patterns. The menu item **Make a new one...** allows you to construct your own pattern and use it to replace one of the predefined patterns. Mouse Kaleidoscope incorporates the pattern maker code from Chapter 9.

While you are editing your pattern, you can preview it by selecting the Preview button. This causes the pattern to be displayed in the right part of the output window. When you are satisfied with your pattern, you select the Ok button. The program then asks you which pattern you wish the new one to replace. It finishes off by making the new pattern the active pattern and continues its kaleidoscope. The Cancel button is supposed to allow you to change your mind and return to the program with nothing disturbed. There is a BUG in the current implementation, however. If you start to make a new pattern and preview it at least once, when you select Cancel, you don't get back to where you were when you started the new pattern. Mouse Kaleidoscope continues its display using the partially constructed pattern you supposedly just cancelled. This happens because Preview calls PENPAT with the pattern you are constructing, but then neglects to recall it later. See if you can fix this. Figures 23-9 to 23-11 show the pattern making process in action.

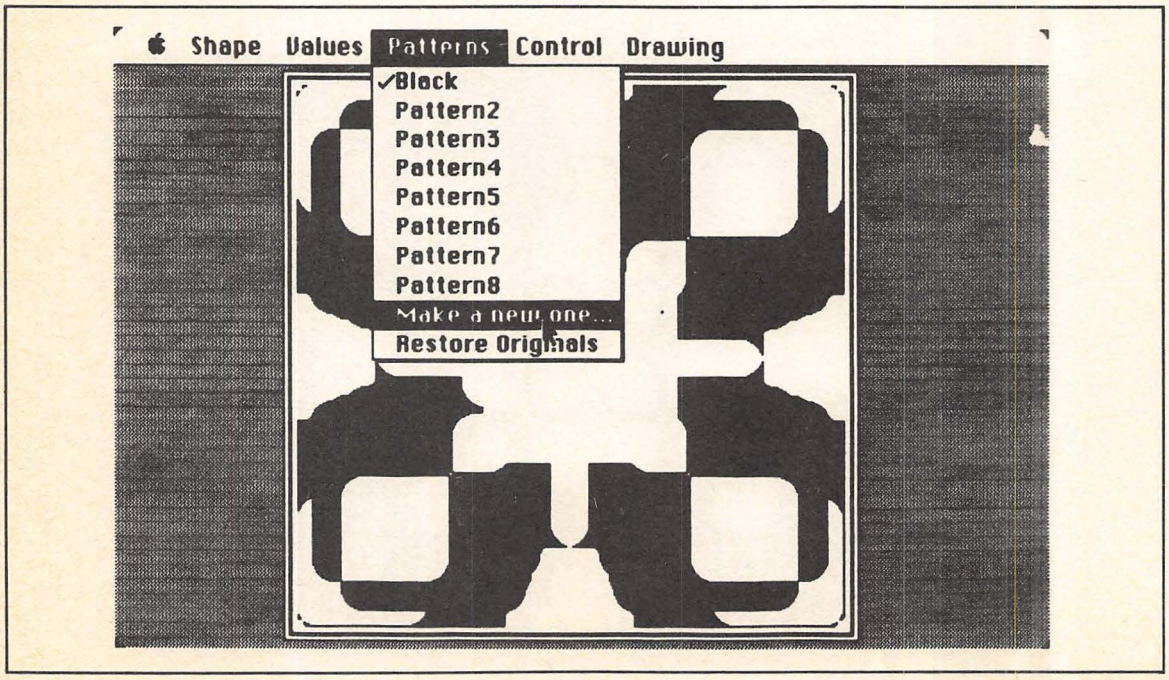


Fig. 23-8. The Patterns menu.

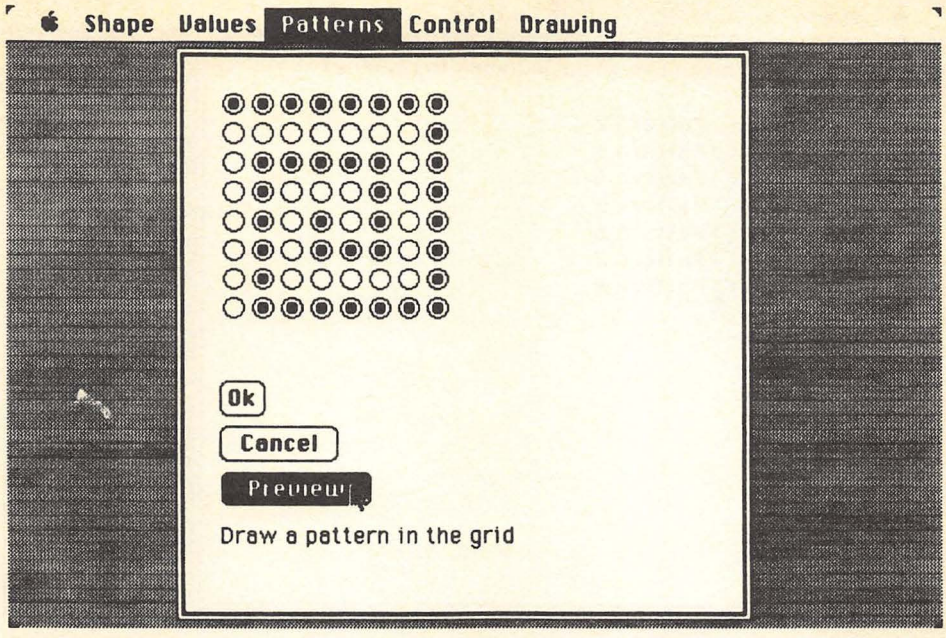


Fig. 23-9. The display when the user has chosen to preview a constructed pattern.

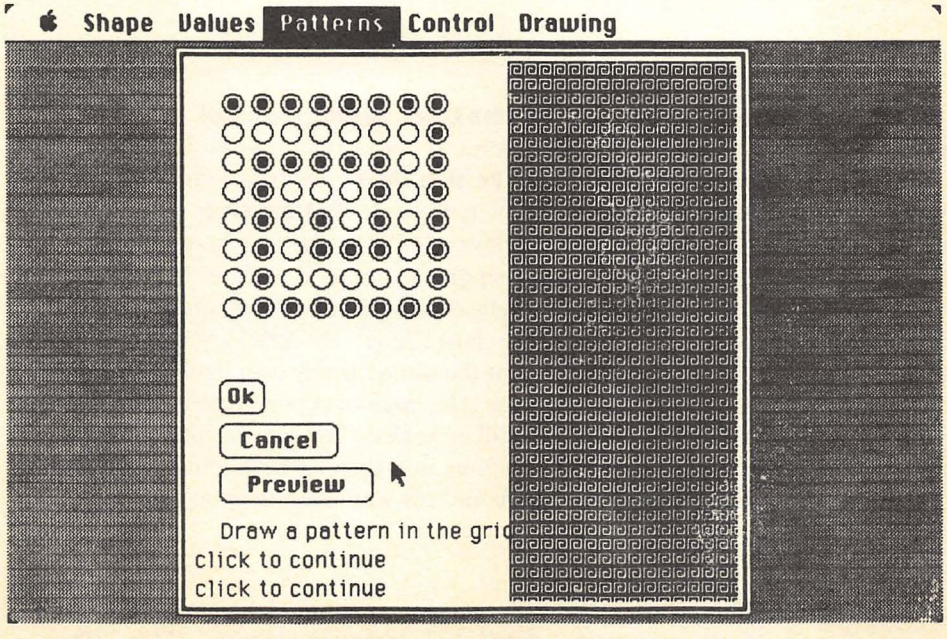


Fig. 23-10. The pattern being previewed fills the right half of the output window.

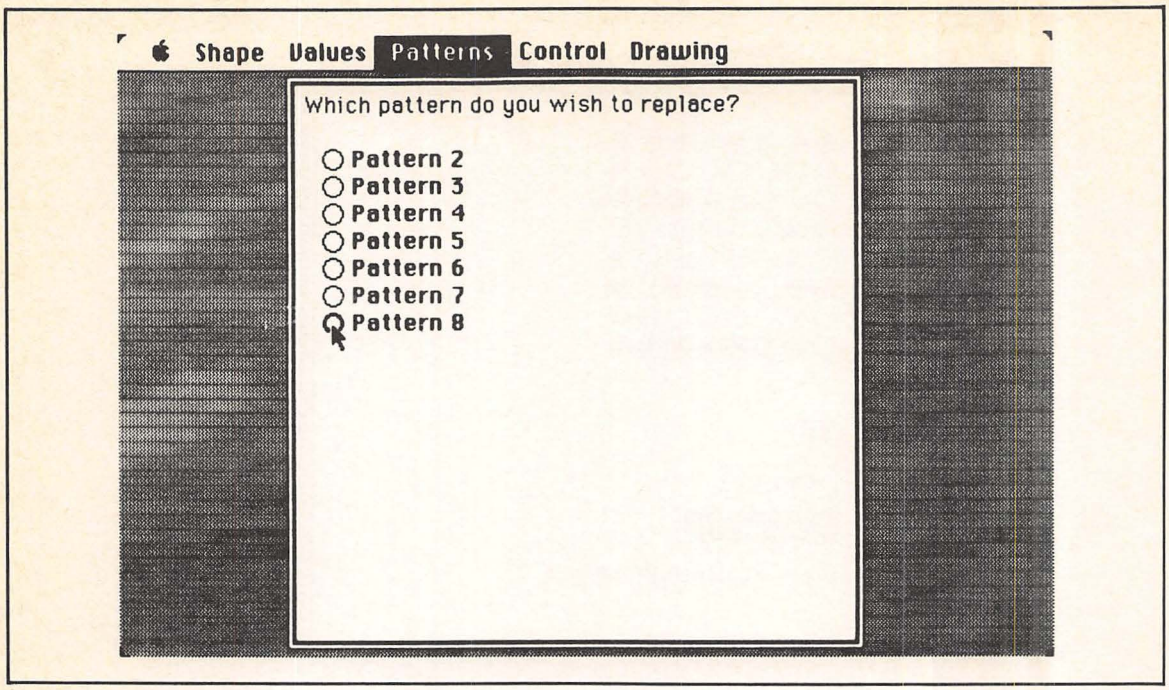


Fig. 23-11. The pattern replacement dialog.

Further discussion of the implementation techniques, especially the dialog, are given below.

Control

The **Control** menu has three choices: **Pattern Clear**, **Repeat**, and **Quit** .

The **Quit** choice allows you to stop the program. It closes the program window and resets the menu bar to the MS-BASIC menus. **Pattern Clear** controls whether or not the screen is cleared when the user selects a new pattern. If **Pattern Clear** is off, the display is not cleared and the new pattern overlays what was already there.

The Repeat choice provides the program's single most useful feature. If Repeat is checked, the program will repeat the entire kaleidoscope over and over again. It uses the variable **spoor.length** discussed earlier to decide how "long" the kaleidoscope will be. This might at first seem to be boring. Why look at the same display over and over again? Well, the only thing that is frozen during the repeats is the pattern of points used to generate the kaleidoscope patterns. Everything else can still be varied. This allows you to "build" a path on the screen for the kaleidoscope production, and then sit back and change the size, delay, and patterns. The variety you can obtain this way may surprise you. Try it!

Drawing

The selections in this menu control the way that the graphical elements are drawn. The choices are **Copy**, **Invert**, **Negative Copy**, and **Negative Invert** . They more or less speak for themselves. One point to note, for the black pattern, neither

Negative Copy nor **Negative Invert** should be chosen, since both have the effect of gradually erasing the entire display.

PROGRAM DIALOGS

The Pattern menu is implemented using dialog concepts. The changing of the program variables **spoor.length, delay,** and **size** is also accomplished with a dialog. Let me spend a bit of time discussing the details of how these dialogs work.

Create a New Pattern

The bits of the newly created pattern are stored in the array of radio buttons shown in Fig. 23-9. The program must monitor the user's choice of these buttons, as well as the three push buttons: Ok, Cancel, and Preview.

This is carried out in the subroutine set.pattern and its subordinates. Most of the "action" occurs in the subroutines grid.dialog and check.buttons. First grid.dialog clears the dialog queue by invoking clear.dialog. It then performs a loop that detects button pushes:

```
done = 0
WHILE NOT done
  d = DIALOG (0)
  IF d =1 THEN GOSUB check.buttons
  IF d = 6 THEN done = -1
WEND
```

Only two possible values of DIALOG (0) are of any interest to the subroutine: d = 1, which represents a button push, and d = 6, which means that the user pressed the Return key on the keyboard. When a button push is detected, the subroutine check.buttons is invoked to sort out which button it was. The radio buttons are ignored during the dialog. Do you understand why? The push buttons are checked individually and appropriate action taken. If either Ok or Cancel is pushed, the loop control variable, **done**, is set to −1 so that the dialog loop will be stopped when check.buttons returns. If Preview is pushed, the preview subroutine is invoked to show the user what the pattern will look like.

Replace Existing Pattern with Created Pattern

After the user has constructed a new pattern, it is used to replace one of the seven predefined nonblack patterns. The user is presented with the column of buttons shown in Fig. 23-10. Mouse Kaleidoscope must detect which one is pressed. This is accomplished with a dialog loop in the subroutine pattern.dialog. The details are quite similar to the grid.dialog loop.

Change the Value of a Program Variable

When the user makes a choice from the **Values** menu, a dialog box in which the change is made is presented. The subroutine box handles the dialog for changing any of the values. It contains an edit field that can directly accept the value as typed in by the user. It also has three buttons labelled Up, Down, and Ok. Ok has its usual meaning. The Up and Down buttons cause the value to be increased or decreased by 2.

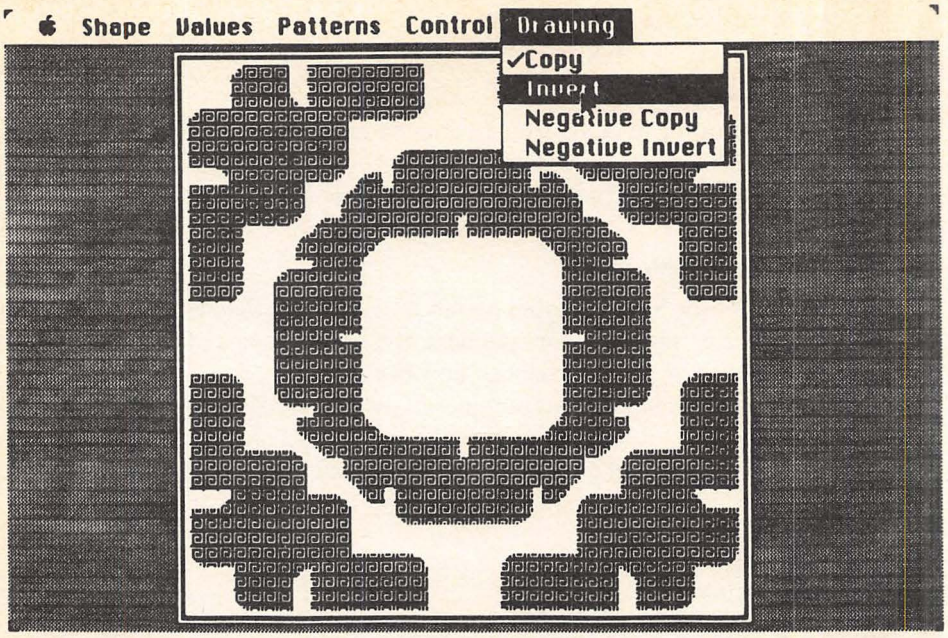


Fig. 23-12. The display when the user is about to change drawing mode from Copy to Invert.

The dialog loop for this box is a little more complicated than the pattern dialogs. The handling of d = 6 is similar to before. In the case of d = 1, however, a little more care is taken. When d = 1, the number of the button pushed is detected by using the DIALOG(1) function:

```
btn = DIALOG (1)
```

When btn = 2 or btn = 3, the value in the edit field must be changed. The default value attached to an edit field can be set by the EDIT FIELD statement. Likewise, the value stored in the field at any given time can be retrieved by the EDIT$ function.

There is one slight problem that must be solved. The "value" returned by the EDIT$ function is in the form of a string. It must be converted back to the number in order to be changed. Then it must be converted into a string once more in order to assign it as the new value of the EDIT FIELD default. The BASIC functions VAL and STR$ come to the rescue:

VAL (EDIT$ (1)) provides the numeric value of the string EDIT$(1)
STR$ (VAL (EDIT$ (1)) – 2) and
STR$ (VAL (EDIT$ (1)) + 2) provide the new values in string form.

These expressions are used in the two statements beginning with:

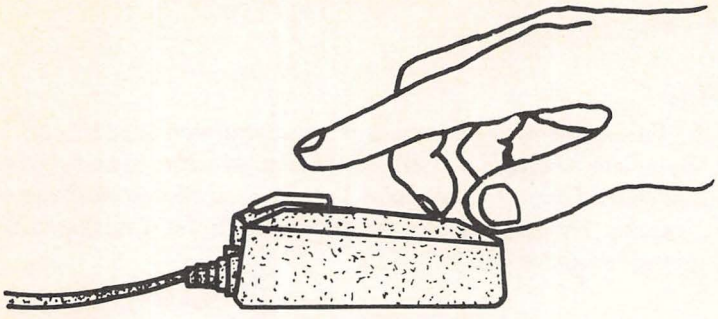
in order to modify the value in the edit field when the user pushes either the Up or the Down button.

CHALLENGES

Erasing the buttons in pattern making is slow—can you speed it up?

Can you implement a feature that allows the display paths to be saved and restored? How about a way to read in paths and replay the kaleidoscope automatically. And as long as you have gone that far, how about a way to generate the displays without any need to move the mouse around at all?

Chapter 24



Vocabulary Quiz

Vocabulary Quiz is but one component of an imagined application called Vocabulary Study. The entire application would be controlled from a master menu program that would CHAIN to the individual modules. Programs like Vocabulary Quiz would make up the modules. Each module would return to the main menu using a CHAIN statement, instead of concluding with an END statement.

Vocabulary Quiz is the least developed of the four applications I have included. Many of its options are either partially or totally unimplemented. There are many questions as to the overall feel of the program as well. The design of the quiz, the flow of the program, and the interactive features are all possible subjects of revision. In addition to design questions, it is quite possible that there are a few bugs left as well.

Figure 24-1 shows the structure diagram for Vocabulary Quiz. The program presents a series of optionally timed vocabulary drills. The quizzes are multiple choice. You are presented with a word and five possible definitions. You choose what you think is the closest definition by clicking on a check box. The words are divided into categories such as noun, verb, and adjective. In each category, the program runs through all its entries before scoring the quiz.

THE PROGRAM LISTING

This program listing contains the symbol ♣. It is used to indicate BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be

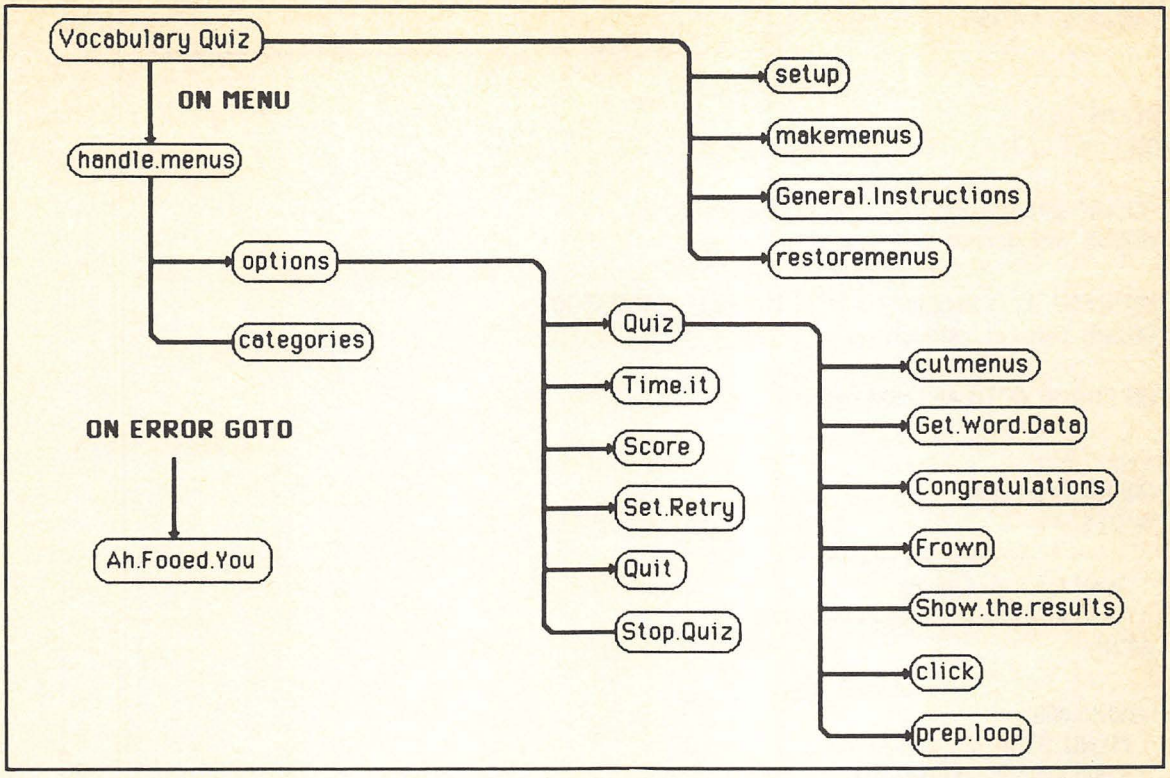


Fig. 24-1. A structure diagram for Vocabulary Quiz.

sure that such lines are joined back together. Always leave at least one or more blanks in place of the ✦; for example, if the listing shows

```
IF XO < X THEN     ✦
    inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN  inside = 0 : RETURN
```

Program Listing 24-1
Vocabulary Quiz

```
'-----------------------------------------------------------------
' Vocabulary Quiz
'
' Timed multiple choice quiz with user
' selectable delay times
'
' Written by
' Richard C. Vile, Jr.
```

```
' January 5, 1985
'-------------------------------------------------------------------

DEFINT a-z
DIM cat$(3)

 GOSUB setup
 GOSUB makemenus

 WINDOW 1, "Vocabulary Quiz", (10, 45) - (460, 300)
 GOSUB General.Instructions

 ON ERROR GOTO Ah.Fooed.You

master.loop:
 GOSUB prep.loop
 WHILE -1
   '----------------------
   ' Wait for menu choice
   '----------------------
 WEND

 handlemenus:
    MENU STOP
    menunumber = MENU(0)
    menuitem   = MENU(1)

    ON menunumber GOSUB options, categories
    MENU ON
    MENU
    RETURN


options:
   ON menuitem GOSUB Quiz, Time.it, Score, Set.Retry, Quit, Stop.Quiz
   RETURN

Quiz:
   WINDOW CLOSE 1
   GOSUB cutmenus
   MENU optionmenu, 6, 1, "Stop Quiz"
   MENU ON
   Done    = 0
   correct = 0
   nw      = 0

   WHILE NOT Done
```

```
GOSUB Get.Word.Data
IF word$ = "xyzzy" THEN Im.Outta.Here
WINDOW 2, "", (10,25) - (200, 225), 4
CALL TEXTFACE (1)
CALL MOVETO (15,15) : PRINT word$;
FOR i = 1 TO 5
  BUTTON i, 1, definition$(i), (15, 25+15*i)- (200, 40 + 15*i), 2
NEXT i
BUTTON 6, 1, "Ok", (130, 160) - (165, 175), 1
Times.Up = 0
ON TIMER (Quiz.Seconds) GOSUB Out.Of.Time
IF Quiz.Seconds > 0 THEN TIMER ON ELSE TIMER OFF

Picked = 0 : OldCh = 1 : Choice = 1
WHILE NOT Picked

  WHILE (DIALOG (0) <> 1) AND (NOT Times.Up)
  WEND : ' Wait for a button or out of time
  Buttonpushed = DIALOG (1)
  IF Buttonpushed = 6 THEN Picked = -1 ELSE OldCh = Choice : Choice = Buttonpushed
  IF Picked THEN PickAgain
  BUTTON OldCh, 1, definition$(OldCh),  +
    (15,25+15*OldCh)-(200,40+15*OldCh), 2
  BUTTON Choice, 2, definition$(Choice),  +
    (15,25+15*Choice)-(200,40+15*Choice), 2
PickAgain:
WEND
IF Choice = Answer THEN GOSUB Congratulations ELSE GOSUB Frown
WEND

Im.Outta.Here:
  GOSUB show.the.results
  GOSUB click
  GOSUB prep.loop
RETURN

Time.it:
  RETURN

Score:
  RETURN

Set.Retry:
  RETURN

Quit:
  ON ERROR GOTO 0
  END
```

```
Stop.Quiz:
   '-----------------------------------
   ' Force a SYNTAX ERROR in order
   ' to get to ON ERROR trap.  That, in turn
   ' will route us back to the outer
   ' loop of the program.
   '-----------------------------------
   PRINT /
   RETURN

categories:
   MENU categorymenu, current.cat, 1
   which.cat$ = cat$ (menuitem)
   current.cat = menuitem
   MENU categorymenu, current.cat, 2
   RESTORE
   READ next.cat$
   WHILE next.cat$ <> which.cat$
     WHILE word$ <> "xyzzy"
       GOSUB Get.Word.Data
     WEND
     READ next.cat$
   WEND
   RETURN

Out.Of.Time:
   Times.Up = -1
   RETURN

Congratulations:
   correct = correct + 1
   WINDOW 3, "", (220, 170) - (460, 310), 2
   PRINT "Right you are!"
   RETURN

Frown:
   WINDOW 3, "", (220, 170) - (460, 310), 2
   PRINT "Sorry, wrong answer."
   RETURN
show.the.results:
   WINDOW 3
   PRINT "You got "; correct; " out of a possible "; nw
   RETURN

General.Instructions:
   PRINT "This program will test your vocabulary skills."
   PRINT "You will be presented several words whose definitions"
```

```
    PRINT "you are to determine. Each word will be displayed in a"
    PRINT "box, along with 5 possible definitions. Check the definition"
    PRINT "that you think is closest to the meaning of the word. You"
    PRINT "will be allowed 5 seconds per word to start with. If you"
    PRINT "wish more time or less time, use the Timing... option to change"
    PRINT "the value. When you are ready to begin, select";
    CALL TEXTFACE (1)
    PRINT " Start Quiz"
    CALL TEXTFACE (0)
    PRINT "from the Options menu. Good Luck!"
    RETURN

Ah.Fooed.You:
    WINDOW CLOSE 2
    WINDOW CLOSE 3
    RESUME Master.Loop

makemenus:
    FOR mn = 1 TO 3   : 'DEBUG ONLY - Change to 5 later
      MENU mn, 0, 0, ""
    NEXT mn

    optionmenu = 1
    MENU optionmenu, 0, 1, "Options"
    categorymenu = 2
    MENU categorymenu, 0, 1, "Categories"
    RETURN

restoremenus:
    MENU optionmenu, 1, 1, "Start Quiz"
    MENU optionmenu, 2, 1, "Timing..."
    MENU optionmenu, 3, 1, "Score..."
    MENU optionmenu, 4, 1, "Retries..."
    MENU optionmenu, 5, 1, "Quit"

    MENU categorymenu, 1, 1, "Adjectives"
    MENU categorymenu, 2, 1, "Nouns"
    MENU categorymenu, 3, 1, "Verbs"
    RETURN

cutmenus:
    FOR it = 1 TO 5
      MENU optionmenu, it, 0
    NEXT it
    FOR it = 1 TO 3
      MENU categorymenu, it, 0
    NEXT it
    RETURN
```

```
setup:
   cat$ ( 1 ) = "...Adjectives"
   cat$ ( 2 ) = "...Nouns"
   cat$ ( 3 ) = "...Verbs"
   Quiz.Seconds = 5
   RETURN
prep.loop:
   TIMER OFF
   GOSUB restoremenus
   menuitem = 1
   GOSUB categories
   WINDOW CLOSE 2
   WINDOW CLOSE 3
   ON MENU GOSUB handlemenus
   MENU ON
   MENU
   RETURN

click:
   PRINT "Click to continue..."
   ms = MOUSE ( 0 )
   WHILE MOUSE ( 0 ) = 0 : WEND
   WHILE MOUSE ( 0 ) <> 0 : WEND
   RETURN

Get.Word.Data:
   nw = nw + 1
   READ word$
   IF word$ = "xyzzy" THEN nw = nw - 1
   FOR i=1 TO 5
     READ definition$ ( i )
   NEXT i
   READ answer
   RETURN
   DATA "...Adjectives"
   DATA ostensive, silent, obvious, apparent, harmful, joyous, 3
   DATA stertorous, offensive, rude, aggressive, smelly, snorting, 5
   DATA nacreous, smelly, black, slimy, ugly, pearly, 5
   DATA columbine, dovelike, hesitant, voluble, royal, saffron, 1
   DATA tendentious, skinny, branched, ropelike, biased, parted, 4
   DATA epicene, centered, neuter, effete, ageless, ancient, 2
   DATA xyzzy, a,a,a,a,a,1
   DATA "...Nouns"
   DATA nosh, lunch, snack, treat, bread, jam, 2
   DATA pollex, hive, voter, ambrosia, thumb, bee, 4
   DATA ostler, stableboy, duck, projection, bonespur, forge, 1
   DATA muscovado, sweet wine, spasm, cane sugar, hideout, weed, 3
```

```
DATA muzhik, cacaphony, peasant, hound, mycelium, jelly, 2
DATA farandole, carbuncle, dust, ornament, journey, dance, 5
DATA xyzzy, a,a,a,a,a,1
DATA "...Verbs"
DATA inveigle, lure, protest, rant, upstage, oil, 1
DATA fleer, chew, scoff, shout, escape, infest, 2
DATA xyzzy, a,a,a,a,a,1
DATA "...End"
```

HOW'S YOUR VOCABULARY?

Vocabulary Quiz consists of multiple choice questions that you answer by clicking the appropriate check box. The program begins by displaying general instructions and then waits for the user to select **Start Quiz** from the **Options** menu. Figure 24-2 shows a quiz about to begin. If you don't select a category before beginning, the program starts out with **Adjectives**.

As long as time has not run out, the quiz taker is allowed to change answers indefinitely. The user signals satisfaction by either clicking the Ok button or pressing the Return key. The program then judges the answer—it is either correct or incorrect. The quiz taker is notified of the outcome in a separate output window. When all the questions in a particular category have been responded to, a summary score is displayed in the same window. These activities are illustrated in Figs. 24-3 to 24-6.

PROGRAMMING TECHNIQUES USED IN VOCABULARY QUIZ

Vocabulary Quiz contains a number of interesting programming tricks of the trade. These are general enough that you may find use for them in your own programs.

Using Expressions in BUTTON Commands

The quiz window is constructed through the use of BUTTON statements that indicate the answers. There are five alternatives for each vocabulary word in the quiz. Each of these is displayed with its own check box. Putting them on the screen could have been done using five separate BUTTON statements. I choose to use a loop instead, with the loop variable i assisting in the specification of the BUTTON command arguments:

```
FOR i = 1 TO 5
   BUTTON i, 1, definition$ (i), (15, 25+15*i) - (200, 40+15*i), 2
NEXT i
```

Note especially how i is used to specify the rectangles containing each check box. This technique is not limited to check boxes.

The Quiz Dialog Loop

The dialog loop in Vocabulary Quiz is similar to others presented earlier. There are more combinations of circumstances to handle in it, however. The version presented in Listing 24-1 is still full of problems. I will outline them here and leave it as a challenge for you to solve as many as you can. A full discussion of the solutions can be found in Chapter 27.

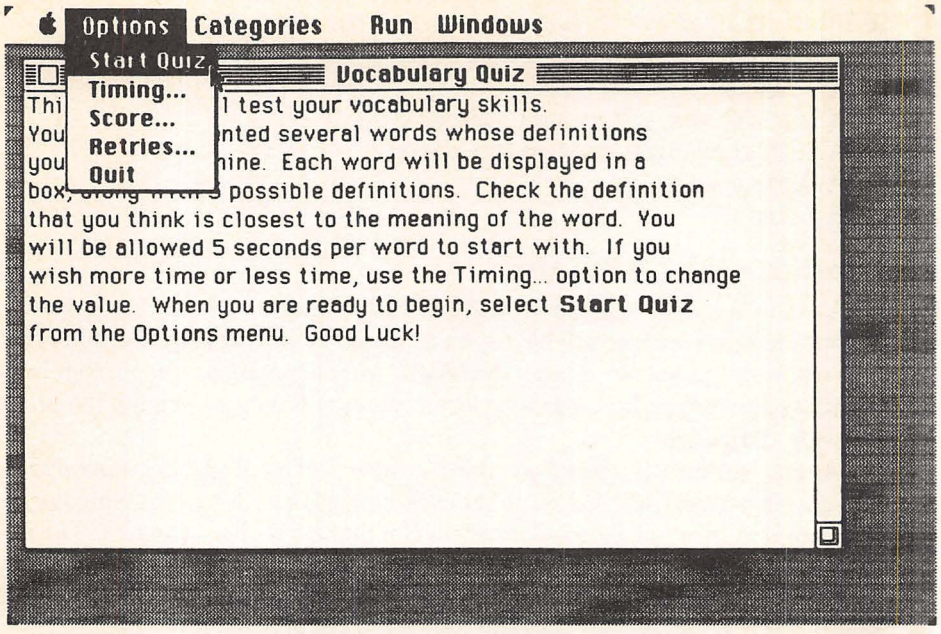


Fig. 24-2. The display as the user is about to begin a quiz session.

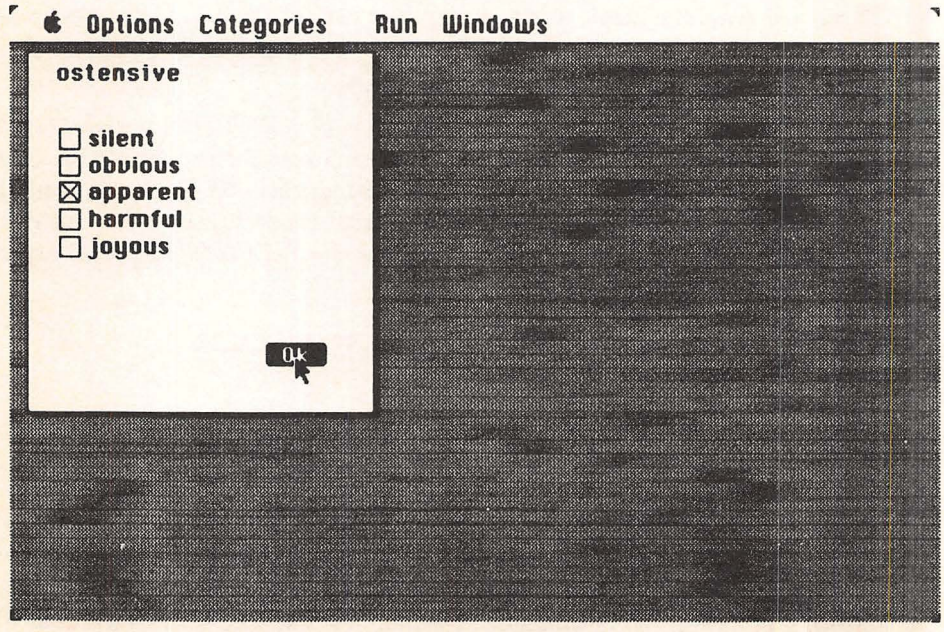


Fig. 24-3. The user clicks Ok when satisfied with the answer.

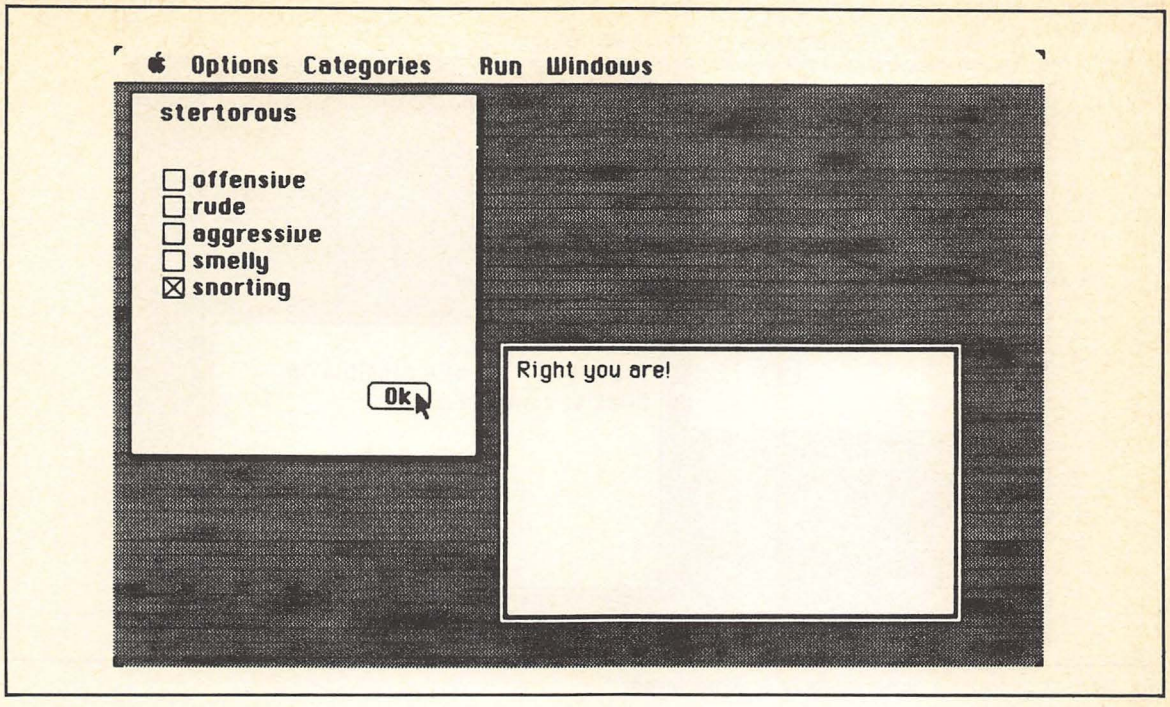


Fig. 24-4. How Vocabulary Quiz indicates a correct answer.

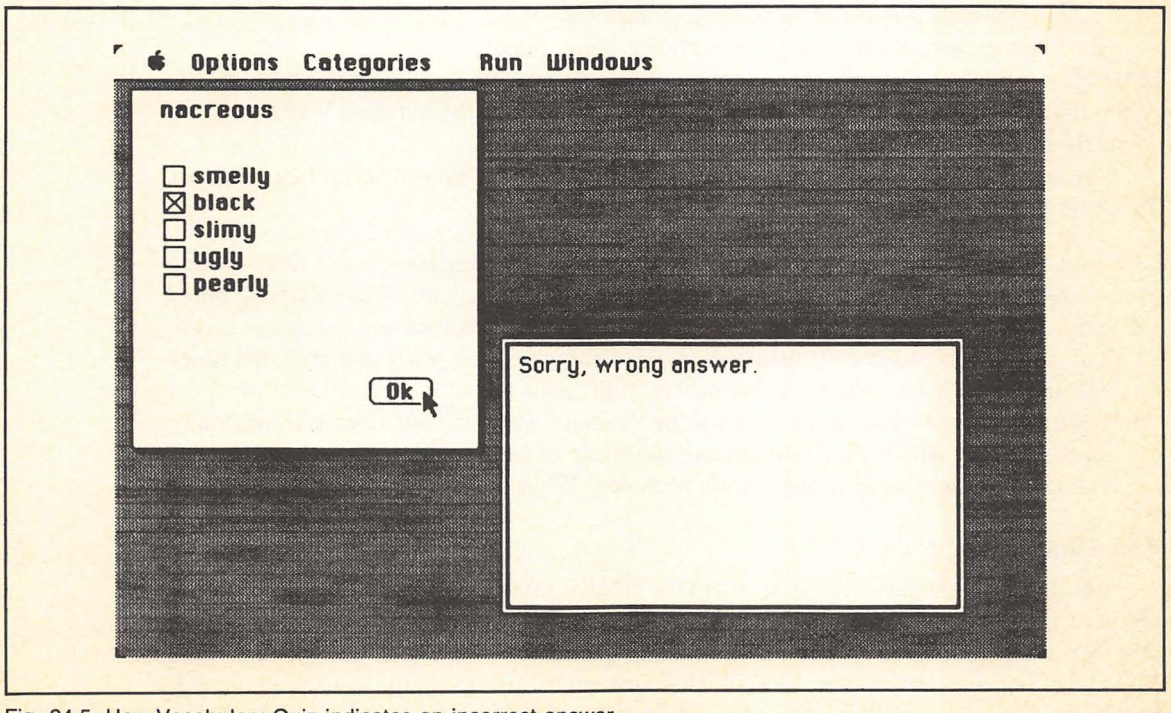


Fig. 24-5. How Vocabulary Quiz indicates an incorrect answer.

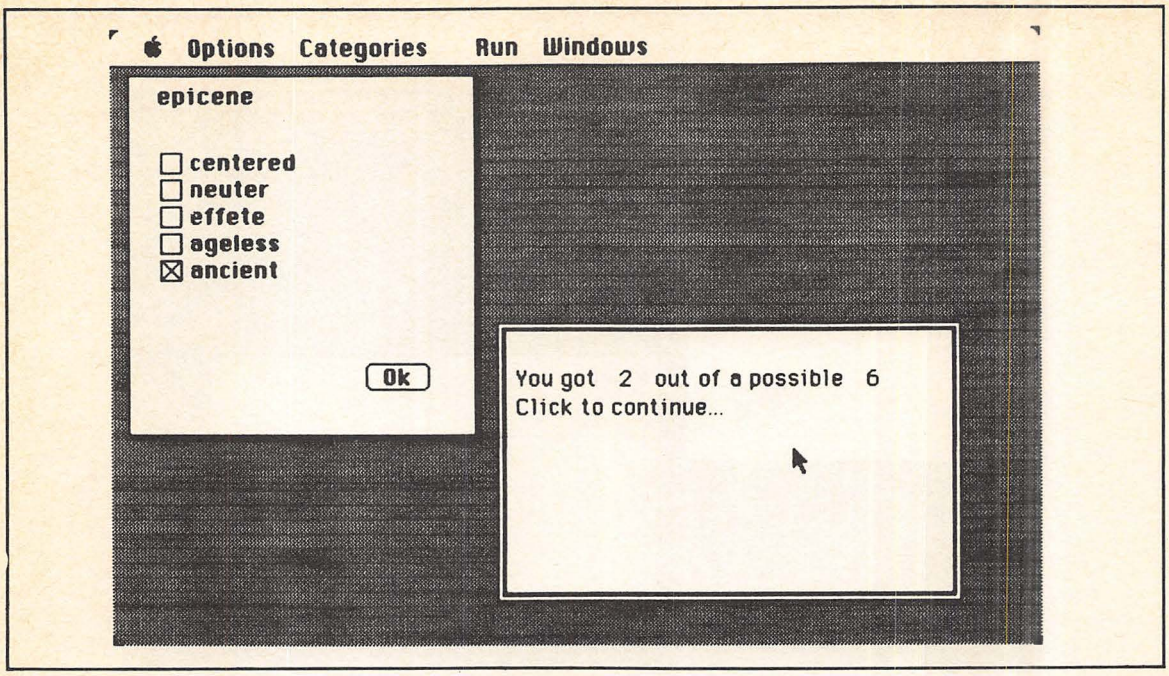


Fig. 24-6. Scoring for a given category is summarized.

The idea behind the loop is to wait for one of two events to occur:

1. A final choice is made: The user pushes Ok.
2. Time expires with or without a choice being made.

Meanwhile, the user may check an answer, which must also be detected and recorded. When the loop is finished, the program must report whether the question was answered correctly or incorrectly. That's the idea, anyway.

Now to the actual facts—here are some experiments for you to try. They point up the program's deficiencies.

1. Start the quiz and then just sit and wait. You are only supposed to get five seconds to answer. In fact, you can sit all day and nothing will happen. What's wrong here?
2. Now click the Ok button without having made a choice. Then sit and wait some more. This time, for some strange reason (a favorite phrase among programmers!), the timer seems to go off ok. The program scores your answer.
3. Continue to wait—the entire quiz will be "taken" without your having to push any more buttons. At the end, you get the summary of how you did. Amazing! You got one right without ever having made a choice! What is happening?!

The Timer Trap

Vocabulary Quiz uses the statements ON TIMER (n) GOSUB and TIMER ON to implement the timing feature. There is a variable called **Quiz.Seconds** that determines whether or not timing will occur. At the startup of each quiz, it is examined. If the value it contains is greater than zero, the TIMER statements are executed, and the quiz will be timed. At least it will be after you and I fix the dialog loop.

The ON TIMER (n) GOSUB statement works like other MS-BASIC event trapping statements. In this case the event is the expiration of a certain number of seconds. The time is measured from when the program executes the TIMER ON statement. The TIMER OFF statement can be used to stop the trap from occurring when you no longer want it to occur.

A Special Use of the RESTORE Command

The data for the quiz questions and answers is stored in DATA statements by the program. Rather than read the data into arrays, the program reads the questions and answers sequentially. This saves some memory to allow the program to be expanded. The problem is that I have decided to have different categories of questions. The program must locate the right place to start reading the DATA for each category.

Look at the categories subroutine. It first uses RESTORE to ensure that the data is being read from the start. It then uses a loop to go through the data one category at a time until the desired one is found. The data contains special entries beginning with " . . . " to mark the start of each category: . . . Adjectives, . . . Nouns, and . . . Verbs. The loop checks for those. If it doesn't get a match, it loops through all the questions in the current category and tries the next one. The subroutine Get.Word.Data reads one question. If the word being "defined" is the string "xyzzy", it knows that the end of a category has been reached.

Getting Out of a Loop in a Hurry

The Options menu has the choice Stop Quiz. This is for when you get tired of one category and you want to stop without going through all the questions. It is supposed to get the program out of the quiz immediately. The original implementation didn't quite work that way. It is interesting to see what the problem was and how it was solved.

My first try at the Stop Quiz feature used the loop control variable **Done** of the quiz loop. When the item was selected, the Stop.Quiz subroutine would eventually be reached and **Done** was set equal to −1. This worked, but only after a fashion. The user could select **Stop Quiz** anywhere in the loop, but the variable **Done** was not tested until the loop reached the WEND statement at the "bottom." So the user had to finish the question that was in progress when the **Stop Quiz** item was chosen. Picky, picky, picky! Well, little things like that can make or break the flow of a program.

I could have inserted a test of Done after every statement in the loop:

```
IF Done THEN GOTO Im.Outta.Here
```

but that would have been extremely tedious, annoying, and wasteful. I finally thought of a more compact solution and here it is. In the subroutine Stop.Quiz the program commits a Syntax Error. This is done on purpose. There is an ON ERROR GOTO statement earlier that routes the program to the label 'Ah.Fooed.You.' Once this label is reached the program can use a RESUME statement to get just about anywhere it wants. It is not obliged to go back and finish out its dull quiz loop. In the case of Listing 24-1, the RESUME statement takes us back to the program's main loop. It could just as easily have gone to Im.Outta.Here if a partial score of the quiz in progress was desired.

This technique is fun. It is simple yet subtle. It is also dangerous. If you leave the

ON ERROR GOTO statement active during program development, it can mask other errors that creep in. To be on the safe side, it is a good idea to activate the ON ERROR trap only when you are satisfied that the rest of the program has been adequately debugged. If you encounter any further strange behavior after you activate the trap, immediately take it out and solve the problem before proceeding.

There's a Whole Lot of Dimming Going On

Once a quiz has begun, the only menu item that makes any sense at all is **Stop Quiz** . Vocabulary Quiz contains two subroutines that alternately dim and undim all items except **Stop Quiz** . They are cutmenus and restoremenus, respectively. Figure 24-7 shows the main menu during a quiz, and Fig. 24-8 shows the Categories menu as it appears during a quiz.

WHAT STILL NEEDS TO BE DONE

There are a number of features of Vocabulary Quiz that are either partially or totally unimplemented. There is considerable scope for further development. If you find it to your liking, you might try to rectify one or more of the following problems.

More Questions, Please

Not very many questions have been included in Listing 24-1. In addition, many of the ones that are there are a bit esoteric. Why not try your hand at making up some more. It's not really programming, but then making up quizzes can sometimes be as good or better than taking them.

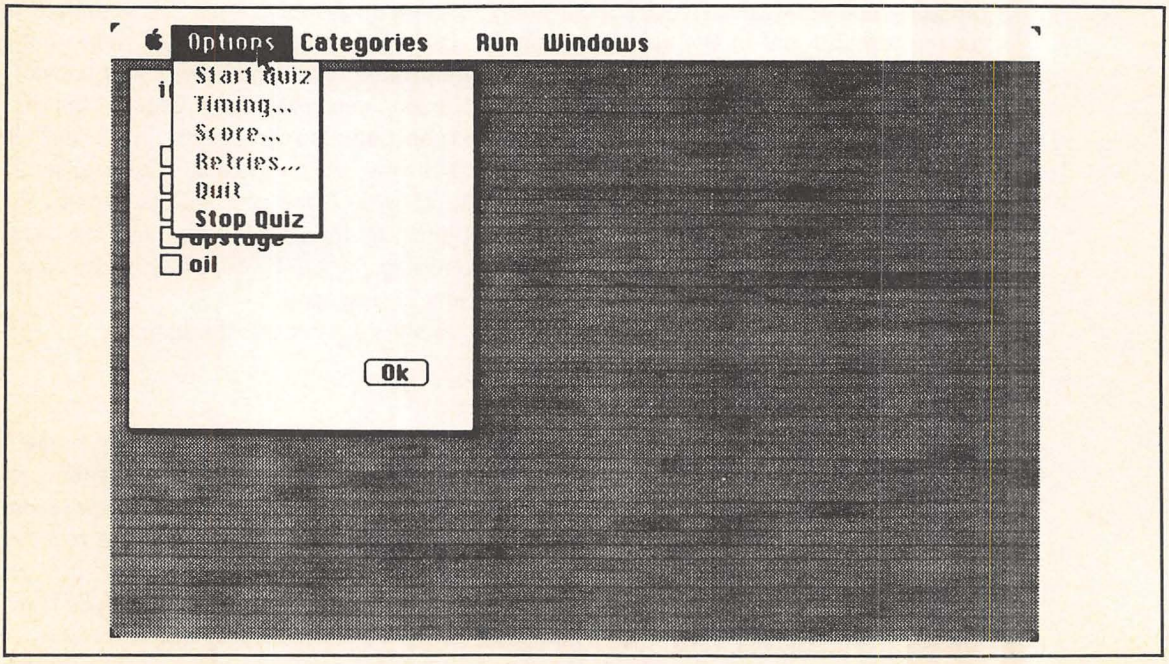


Fig. 24-7. Most menu items are dimmed while the quiz is being taken. The Stop Quiz option will abort the current quiz before all the questions have been presented.

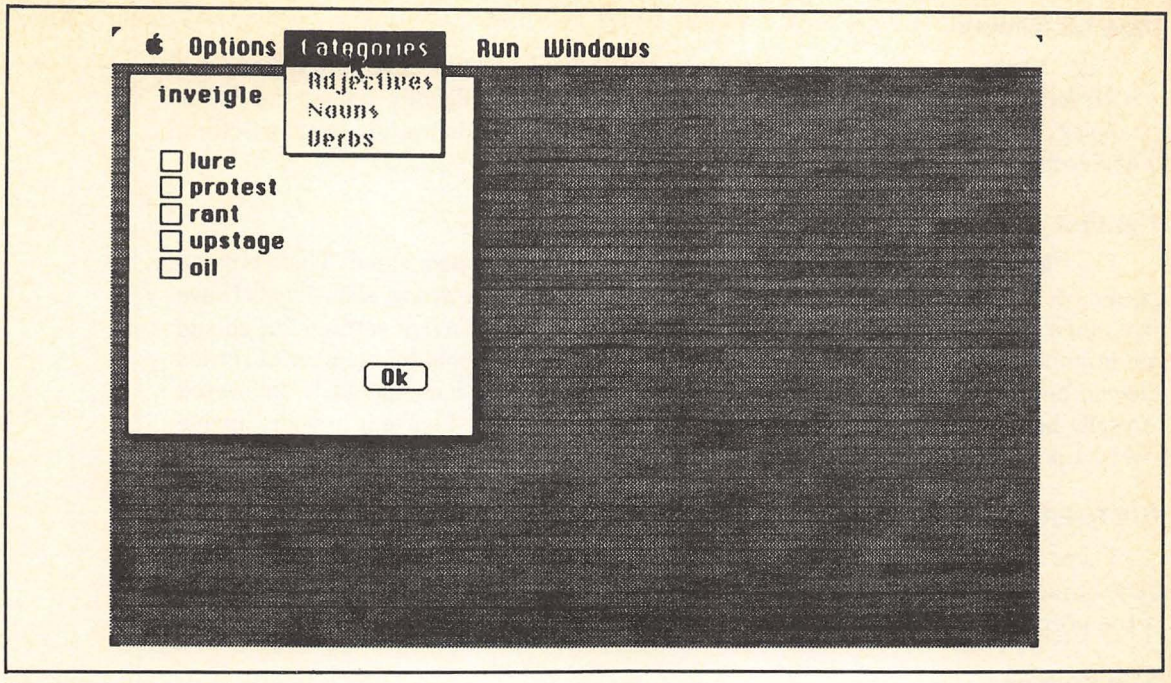


Fig. 24-8. The Categories menu during a quiz.

Frown and Smile

When the answer to a question is judged, the program merely prints the one liners, "Right you are!" or "Sorry, wrong answer." The idea I had in mind was to draw smiling or frowning faces in addition to these prosaic messages. I just never got around to doing it. Here is a chance for you to try out your Quickdraw skills by providing the missing graphics.

A Persistent Window

The judging window stays on the screen after the first answer is judged. This is a bit distracting to the quiz taker. It may be confusing as well, since it is easy to miss the judging message being printed in the window. If the message is the same twice in a row, it may appear that the program has forgotten to judge at all. The user can be trapped into staring at the window and forgetting that the next question is timed as well.

There are a number of things that need to be done. One is to make sure the window is always closed during the actual questions. Another is to make sure that it stays on screen enough for the judging message to be absorbed. You can experiment here. Try a delay loop, or use the mouse click utility. You could even use an Ok button if you like.

An Unclean Menu Bar

The MS-BASIC **Run** and **Windows** menus have been left on the menu bar to facilitate debugging the program. It is easier to get the screen back to normal that way. Don't forget to clean up the menu bar after you have finished with your changes.

Time for Timing

The **Timing...** item on the Options menu is implemented. The program uses a predetermined five second time limit right now. If you have diligently studied the earlier examples of dialog, you should have no difficulty setting up a dialog box to allow setting Quiz.Seconds.

If at first . . .

The **Retries...** choice in the **Options** menu is also not implemented. There is some question of how this should work. When the user gets a question wrong, should you always give him a retry up to the maximum number allowed by the current setting? Or, should you prompt the user as to whether a retry is desired? How should the number of retries allowed and used affect the scoring? These and other questions should be answered carefully before you attempt to implement the retries feature. I leave it in your capable hands. I have not worked on this feature in my version at all.

FURTHER DEVELOPMENT AND ASSOCIATED PROGRAMS

There are a number of ideas that have occurred to me as to extending the Vocabulary Quiz. Some of these have already been mentioned as part of the development process of the program itself. Others involve writing additional programs.

Quiz Builder

To make the process of creating data for the program to use more convenient, you could design and implement a Quiz Builder Program. This would perhaps use a form with edit fields to build the questions interactively. It could then write out a text file with the DATA statements corresponding to the questions. Like Cursor Editor earlier, this could involve more elaborate capabilities such as editing previous quizzes, merging groups of questions, and so on.

Study Programs

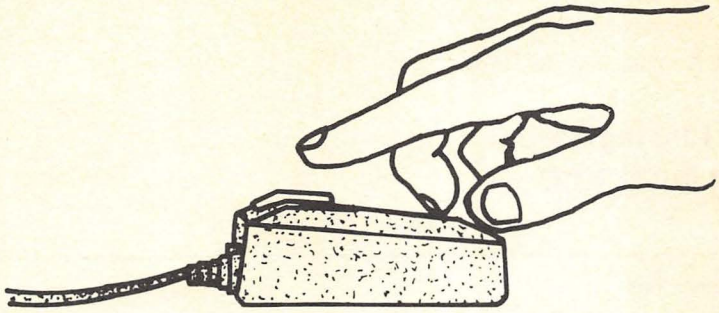
Vocabulary Quiz is but one possibility in a collection of programs designed to increase one's vocabulary. Drill and practice programs could be built that allowed intensive study of different areas of vocabulary: root words, words derived from Latin and Greek, words borrowed from French and other Romance languages, prefixes, suffixes, word families, technical terms, and on and on. Each such program could provide a form of electronic flash card for study purposes. Of course, writing such programs will be easier if you first build the necessary tools. Other programs could be written to build and maintain databases of word information. The list could go on almost endlessly. I leave it to you to decide where to stop.

CHALLENGES

This chapter has already had perhaps more challenges than you might care to answer. For good measure, here are one or two last little ones.

- Change Vocabulary Quiz into a Trivia game.
- Investigate other styles of quiz: matching or fill in the blanks.

Chapter 25



Dialog Designer

This chapter presents a rather different application called Dialog Designer. The program allows you to design dialog boxes or data entry forms directly on the Macintosh screen. The boxes may contain any type of button, decorative text, and edit fields. You use a pair of flying crosshairs to position the elements as you add them to the box. You can delete an element if you don't like its position. Then you can add it back where you really want it to be. The program uses an invisible grid to make it easier to line things up in rows and columns if you want. When you are all done adding elements to your dialog box or form, Dialog Designer will write a text file containing BASIC statements. When you load that file into BASIC and run it as a BASIC program, the form you designed will be reproduced before your very eyes. You can then include the form in an application, either by itself or with other forms you have designed with Dialog Designer.

Dialog Designer is only one small example of an entire category of exciting utility programs. I hope you will be motivated by studying it to write your own program generators and screen design aids.

THE PROGRAM LISTING

This program listing contains the symbol ✥. It is used to identify BASIC statements that in the listing occupy two lines, but that when typed into an MS-BASIC program should occupy only a single line. This notation is used because certain lines are too long to fit on a single printed line of text. If you should choose to enter the program yourself, be sure that such lines are joined back together. Always leave at least one or more blanks in the place of the ✥ ; for example, if the listing shows

```
IF XO < X THEN    ↵
    inside = 0 : RETURN
```

then your program should look like:

```
IF XO < X THEN inside = 0 : RETURN
```

Program Listing 25-1
Dialog Designer

```
'-----------------------------------------------------------------------
' Dialog Designer One
'
' Startup Program - Creates dialog box on screen,
' then chains to program to add buttons and edit
' fields
'
' Written by
' Richard C. Vile, Jr.
' February 1, 1985
'-----------------------------------------------------------------------
DEFINT a-z

COMMON top.line(), bottom.line(), left.line(), right.line()
DIM top.line(200), bottom.line(200), left.line(300), right.line(300)

WINDOW CLOSE 1
GOSUB makemenus
MENU ON
ON MENU GOSUB make.box

still.thinking = -1
WHILE still.thinking
  '----------------
  ' Wait for actions
  '----------------
WEND
CHAIN "Fill in Dialog Box"

make.box:
  MENU OFF
  menunumber = MENU(0)
  menuitem   = MENU(1)

  ON menunumber GOSUB actions, type, for.heavens.sake.help.me.out
  MENU ON
```

```
  MENU
  RETURN

actions:
  ON menuitem GOSUB preview, add.controls, quit
  RETURN

preview:
  WINDOW 1, "", (15,30)-(495,345), type.of.box
  CALL MOVETO (10, 30)
  PRINT "Drag open a box inside me"
  GOSUB open.rectangle
  GOSUB show.box
  WINDOW CLOSE 2
  GOSUB clear.dialog.queue
  RETURN

show.box:
  WINDOW CLOSE 1
  WINDOW 2, "", (x0+15, y0+30) - (x1+15, y1+30), type.of.box
  dbx0 = x0+15 : dby0 = y0+30 : dbx1 = x1+15 : dby1 = y1+30
  BUTTON 1, 1, "Ok", (0, 0) - (30, 16)
  GOSUB clear.dialog.queue
  WHILE DIALOG (0) = 0
  WEND
  btn = DIALOG (1)
  RETURN

add.controls:
  still.thinking = 0
  RETURN

quit:
  MENU RESET
  WINDOW CLOSE 1
  WINDOW CLOSE 2
  END

type:
  MENU 2, type.of.box-1, 1
  MENU 2, menuitem, 2
  type.of.box = menuitem+1
  RETURN

for.heavens.sake.help.me.out:
  ON menuitem GOSUB what.its.for, how.to.preview, how.to.add, how.to.type, how.to.quit
  RETURN
```

```
what.its.for:
  WINDOW 1,"",(15,30)-(490,300), type.of.box
  CLS
  CALL MOVETO (10,30)
  PRINT "What is this program for?" : PRINT
  PRINT PTAB(10); "This is a dialog box design program. It is intended"
  PRINT PTAB(10); "to make the task of laying out dialog boxes and"
  PRINT PTAB(10); "writing the Basic code to implement them easier."
  PRINT PTAB(10); "You will create your own dialog box on the screen,"
  PRINT PTAB(10); "add the controls you want and be able to see at all"
  PRINT PTAB(10); "stages exactly what the final box will look like."
  PRINT PTAB(10); "You may save the design in a special coded file to be"
  PRINT PTAB(10); "worked on later - or you may request that Basic"
  PRINT PTAB(10); "program statements be generated. The Basic statements"
  PRINT PTAB(10); "when later included in a program will recreate the dialog"
  PRINT PTAB(10); "box exactly as you constructed it with this designer."
  PRINT
  GOSUB ok
  WINDOW CLOSE 1
  RETURN

how.to.preview:
  WINDOW 1,"",(15,30)-(490,325), type.of.box
  CLS
  CALL MOVETO (10,30)
  PRINT "Setting up your dialog box" : PRINT
  PRINT PTAB(10); "To set up your dialog box, first choose setup from the Dialog Box"
  PRINT PTAB(10); "menu. Then in the window use your mouse to open a"
  PRINT PTAB(10); "rectangle. Move the mouse to the spot where you want"
  PRINT PTAB(10); "the upper left hand corner of the box to be located."
  PRINT PTAB(10); "Click the mouse and hold the button down while you move"
  PRINT PTAB(10); "the mouse to the spot where you want the lower right"
  PRINT PTAB(10); "hand corner of the box to be located. As long as you hold"
  PRINT PTAB(10); "down the mouse button, a blinking outline of the box"
  PRINT PTAB(10); "you are forming will appear on the screen."
  PRINT
  PRINT PTAB(10); "When you release the mouse button, a copy of the box"
  PRINT PTAB(10); "will be drawn on the screen. When you want to proceed"
  PRINT PTAB(10); "from there, click the Ok button."
  GOSUB ok
  WINDOW CLOSE 1
  RETURN

how.to.add:
  WINDOW 1,"",(15,30)-(490,250), type.of.box
  CLS
  CALL MOVETO (10,30)
```

```
  PRINT "Continuing the design of your dialog box" : PRINT
  PRINT PTAB(10); "Simply choose 'Add Controls...' from the Dialog Box menu."
  PRINT PTAB(10); "You will then be shown an image of your dialog box"
  PRINT PTAB(10); "and the chance to set up buttons and edit fields"
  PRINT PTAB(10); "inside it. For details, you may use the new choices"
  PRINT PTAB(10); "that appear in the Help! menu."
  PRINT
  GOSUB ok
  WINDOW CLOSE 1
  RETURN

how.to.type:
  WINDOW 1, "", (15,30)-(490, 200), type.of.box
  CLS
  CALL MOVETO (10, 30)
  PRINT "How to determine the type of dialog box you get" : PRINT
  PRINT PTAB(10); "Use the Types menu "; : CALL TEXTFACE (1) : PRINT "before";
  CALL TEXTFACE (0) : PRINT " you choose 'Setup' from the Dialog Box"
  PRINT PTAB(10); "menu. The item with the check mark describes the"
  PRINT PTAB(10); "type of box you will end up with."
  PRINT
  GOSUB ok
  WINDOW CLOSE 1
  RETURN

how.to.quit:
  WINDOW 1, "", (15,30)-(490,150), type.of.box
  CALL MOVETO (10, 30)
  PRINT "How to call the whole thing off" : PRINT
  PRINT PTAB (10); "Just choose 'Quit' from the Dialog Box menu"
  PRINT
  GOSUB ok
  WINDOW CLOSE 1
  RETURN

makemenus:
  '---------------------------------------
  ' Number the menus to make reordering
  ' them on the menu bar easier.
  ' *============> N O T E <============
  ' *If you re-number the menus, you must
  ' *change the ON MENUNUMBER GOSUB
  ' *ordering in the code above.
  ' *============> N O T E <============
  '-----------------------------------
  box.menu = 1
  types.menu = 2
```

```
  help.menu = 3

  MENU box.menu, 0, 1, "Dialog Box"
  MENU box.menu, 1, 1, "Setup"
  MENU box.menu, 2, 1, "Add Controls..."
  MENU box.menu, 3, 1, "Quit"

  MENU types.menu, 0, 1, "Types"
  MENU types.menu, 1, 2, "Framed"
  MENU types.menu, 2, 1, "One-line border"
  MENU types.menu, 3, 1, "Shadowed"
  type.of.box = 2

  MENU help.menu, 0, 1, "Help!"
  MENU help.menu, 1, 1, "What is this program for?"
  MENU help.menu, 2, 1, "How to preview your dialog box"
  MENU help.menu, 3, 1, "How to proceed with adding buttons and edit fields"
  MENU help.menu, 4, 1, "How to change the type of dialog box"
  MENU help.menu, 5, 1, "How to call the whole thing off"

  MENU 4,0,0,""
  MENU 5,0,0,""

  RETURN

ok:
  wide = WINDOW (2)
  high = WINDOW (3)
  BUTTON 1, 1, "Ok", (wide - 40, high -30)-(wide - 10 , high - 14)
  WHILE -1
    d = DIALOG (0)
    IF d = 1 THEN RETURN
  WEND

open.rectangle:

  ms = MOUSE (0)
  WHILE MOUSE (0) = 0 : WEND
  x0 = MOUSE (3) : y0 = MOUSE (4)
  x1 = x0 : y1 = y0
  WHILE -1

    GOSUB get.lines
    GOSUB draw.rect
    xold = x1 : yold = y1
    IF MOUSE (0) >= 0 THEN GOSUB verify : IF its.ok THEN RETURN  +
      CLS : GOTO open.rectangle
```

```
    GOSUB erase.rect
    GOSUB redraw.screen
    x1 = MOUSE (5) : y1 = MOUSE (6)

  WEND

erase.rect:
  LINE (x0,y0) - (x1,y0), 30
  LINE (x0,y0) - (x0,y1), 30
  LINE (x1,y0) - (x1,y1), 30
  LINE (x0,y1) - (x1,y1), 30
  RETURN

redraw.screen:
  PUT (x0,y0) - (x1,y0), top.line, OR
  PUT (x0,y0) - (x0,y1), left.line, OR
  PUT (x1,y0) - (x1,y1), right.line, OR
  PUT (x0,y1) - (x1,y1), bottom.line, OR
  RETURN

draw.rect:
  LINE (x0,y0) - (x1,y0)
  LINE (x0,y0) - (x0,y1)
  LINE (x1,y0) - (x1,y1)
  LINE (x0,y1) - (x1,y1)
  RETURN

get.lines:
  GET (x0,y0) - (x1,y0), top.line
  GET (x0,y0) - (x0,y1), left.line
  GET (x1,y0) - (x1,y1), right.line
  GET (x0,y1) - (x1,y1), bottom.line
  RETURN

verify:
  its.ok = -1 : high.enough = -1 : wide.enough = -1
  IF ABS (x1-x0) < 30 THEN its.ok = 0 : wide.enough = 0
  IF ABS (y1-y0) < 16 THEN its.ok = 0 : high.enough = 0
  IF NOT its.ok THEN GOSUB complain
  RETURN

complain:
  CLS
  IF NOT high.enough THEN PRINT "Not high enough!"
  IF NOT wide.enough THEN PRINT "Not wide enough!"
  FOR delay = 1 TO 5000 : NEXT delay
  RETURN
```

```
clear.dialog.queue:
   WHILE DIALOG (0) <> 0 : WEND
   RETURN
```

Program Listing 25-2
Dialog Designer

```
'-----------------------------------------------------------------------
' Fill in Dialog Box Elements
'
' Allow user to create and position
' controls within dialog box being
' constructed.  At each stage, the box
' is previewed and the user may back
' up or continue.
'
' Written by
' Richard C. Vile, Jr.
' February 3, 1985
' Revised March and April 1985
'-----------------------------------------------------------------------
DEFINT a-z

COMMON top.line ( ), bottom.line ( ), left.line ( ), right.line ( )
COMMON dx0, dx1, dy0, dy1, type.of.box

maxbuttons = 50 : maxfields = 50 : maxnotes = 50
maxelements = maxbuttons + maxfields + maxnotes

DIM x.axis ( 400), y.axis ( 400)
DIM btitle$ ( maxbuttons), btype ( maxbuttons)
DIM bx0 ( maxbuttons), bx1 ( maxbuttons)
DIM by0 ( maxbuttons), by1 ( maxbuttons)
DIM bwide ( maxbuttons), bhigh ( maxbuttons)
DIM notes$ ( maxnotes)
DIM nx ( maxnotes), ny ( maxnotes)
DIM ftext$ ( maxfields)
DIM fx0 ( maxfields), fx1 ( maxfields)
DIM fy0 ( maxfields), fy1 ( maxfields)
DIM fwide ( maxfields), fhigh ( maxfields)
DIM history ( maxelements)

debug = - 1
IF debug THEN dx0 = 15 : dx1 = 495 : dy0 = 40 : dy1 = 345 : type.of.box = 2
vertical = 1 : horizontal = 2
fix.x = - 1 : fix.y = - 1
```

```
btn.high = 16
edt.high = 16
next.btn = 1
next.edt = 1
next.note = 1
num.elts = 1
grid.is.on = -1
grid.grain = 10

GOSUB stage2.menus
WINDOW 2, "", (dx0, dy0) - (dx1, dy1), type.of.box
MENU ON
ON MENU GOSUB handle.it...handle.it:

WHILE -1
  '-------------------------------------
  ' Seems like we're always sitting around
  ' waiting for something to happen
  '-------------------------------------
WEND

GOSUB ok
END

handle.it...handle.it:

  MENU OFF
  menunumber = MENU (0)
  menuitem   = MENU (1)
  ON menunumber GOSUB elements, actions, for.heavens.sake.help.me, dbg
  MENU ON
  MENU
  RETURN

elements:
  '---------------------------------
  ' Whatever is checked in this menu is
  ' what will be added the next time
  ' Add Element is chosen from the
  ' actions menu.
  '---------------------------------
  MENU 1, next.element, 1
  MENU 1, menuitem, 2
  next.element = menuitem
  RETURN

actions:
```

```
  ON menuitem GOSUB add.it, program, change.grid, kill.it,  +
     toggle.debug, noop, noop, noop, noop, quit
  RETURN

add.it:
  IF text$ = "" THEN CALL MOVETO (5, 250) : INPUT "Text "; text$
  GOSUB fix.axis
  ON next.element GOSUB add.btn, add.btn, add.btn, add.edt, noop, add.txt
  x1 = fix.x : y1 = fix.y : GOSUB erase.axes
  RETURN

add.btn:
  IF next.btn > maxbuttons THEN PRINT CHR$(7); : RETURN
  CALL MOVETO (0,0) : CALL TEXTFACE (1) : PRINT text$;
  btn.wide = WINDOW (4) : CALL TEXTFACE (0)
  GOSUB make.history
  btitle$ (next.btn) = text$
  btype (next.btn) = next.element
  bx0 (next.btn) = fix.x \ grid.grain
  bx1 (next.btn) = btn.wide
  by0 (next.btn) = btn.high
  by1 (next.btn) = fix.y \ grid.grain
  GOSUB plunk.button
  next.btn = next.btn + 1
  GOSUB redraw.box
  RETURN

plunk.button:
  b.x0 = bx0 (next.btn) * grid.grain + 1
  b.x1 = bx0 + bx1 (next.btn) + 20
  b.y1 = by1 (next.btn) * grid.grain
  b.y0 = b.y1 - by0 (next.btn)
  BUTTON next.btn, 1, text$, (b.x0, b.y0) - (b.x1, b.y1), next.element
  text$ = ""
  RETURN

add.edt:
  edt.text = -1
  CALL MOVETO (0,0) : PRINT text$; : edt.wide = WINDOW (4)
  IF NOT edt.text THEN text$ = ""
  GOSUB make.history
  ftext$ (next.edt) = text$
  fx0 (next.edt) = fix.x \ grid.grain
  fx1 (next.edt) = edt.wide
  fy0 (next.edt) = edt.high
  fy1 (next.edt) = fix.y \ grid.grain
  GOSUB plunk.edit.field
```

```
    next.edt = next.edt + 1
    GOSUB redraw.box
    RETURN

plunk.edit.field:
    f.x0 = fx0 (next.edt) * grid.grain - 5
    f.x1 = fx0 + fx1 (next.ext) + 5
    f.y1 = fy1 (next.edt) * grid.grain
    f.y0 = f.y1 - fy0 (next.edt)
    EDIT FIELD next.edt, text$, (f.x0, f.y0) - (f.x1, f.y1), 1
    text$ = ""
    RETURN

add.txt:
    CALL TEXTFACE (1)
    CALL MOVETO (fix.x + 1, fix.y - 4) : PRINT text$;
    GOSUB make.history
    nx (next.note) = fix.x \ grid.grain : ny (next.note) = fix.y \ grid.grain
    notes$ (next.note) = text$
    text$ = ""
    next.note = next.note + 1
    CALL TEXTFACE (0)
    GOSUB redraw.box
    RETURN

kill.it:
    IF num.elts <= 1 THEN PRINT CHR$ (7); : RETURN
    last.element = history (num.elts - 1)
    IF debug THEN CALL MOVETO (10,30) : PRINT "last.element= "; last.element;
    IF last.element >= 1 AND last.element <= 3 THEN next.btn = next.btn - 1 :  ✤
      BUTTON CLOSE next.btn
    IF last.element = 4 THEN next.edt = next.edt - 1 : EDIT FIELD CLOSE next.edt
    IF last.element = 6 THEN next.note = next.note - 1
    num.elts = num.elts - 1
    GOSUB redraw.box
    RETURN

make.history:
    history (num.elts) = next.element
    num.elts = num.elts + 1
    RETURN

program:
    INPUT "Program file name "; pgm$
    OPEN pgm$ FOR OUTPUT AS #1
    PRINT #1, "Window 2, null$, (";dx0;",";dy0;") - (";dx1;",";dy1;"), ";type.of.box
    PRINT #1, "DIM notes$ (";next.note-1;")"
```

```
  PRINT #1, "GOSUB read.notes"
  PRINT #1, "CLS"
  PRINT #1, "CALL TEXTFACE (1)"
  FOR nt = 1 TO next.note - 1
    PRINT #1, "CALL MOVETO (";nx(nt)*grid.grain+1;",";
    PRINT #1, ny(nt) * grid.grain - 4;")"
    PRINT #1, "PRINT notes$ (";nt;")"
  NEXT nt
  PRINT #1, "CALL TEXTFACE (0)"
  FOR btn = 1 TO next.btn - 1
    b.x0 = bx0 (btn) * grid.grain + 1
    b.x1 = b.x0 + bx1 (btn) + 20
    b.y1 = by1 (btn) * grid.grain
    b.y0 = b.y1 - by0 (btn)
    PRINT #1, "BUTTON ";  btn; ", 1, "; CHR$ (34);btitle$(btn);CHR$ (34);
    PRINT #1, ", (";b.x0;",";b.y0;") - (";b.x1;",";b.y1;
    PRINT #1, "), "; btype (btn)
  NEXT btn
  FOR edt = 1 TO next.edt - 1
    f.x0 = fx0 (edt) * grid.grain - 5
    f.x1 = f.x0 + fx1 (edt) + 5
    f.y1 = fy1 (edt) * grid.grain
    f.y0 = fy1 (edt) * grid.grain - fy0 (edt)
    PRINT #1, "EDIT FIELD ";edt; ", "; CHR$ (34);ftext$ (edt);CHR$ (34);
    PRINT #1, ", (";f.x0;",";f.y0;") - (";f.x1;",";f.y1;
    PRINT #1, "),  1"
  NEXT edt
  PRINT #1, "END"
  PRINT #1, "read.notes:"
  PRINT #1, "FOR i = 1 TO ";next.note-1
  PRINT #1, "   read notes$ (i)"
  PRINT #1, "NEXT i"
  PRINT #1, "RETURN"
  FOR nt = 1 TO next.note - 1
    PRINT #1, "DATA ";notes$ (nt)
  NEXT nt
  RETURN

toggle.debug:
  IF debug THEN GOSUB deactivate.debug ELSE GOSUB activate.debug
  IF debug THEN MENU act.menu, 5, 2 ELSE MENU act.menu 5, 1
  RETURN

for.heavens.sake.help.me:
  RETURN

dbg:
```

```
  ON menuitem GOSUB dump.hist, sh.nxtelt, sh.nxtb, sh.next.edt, sh.num
  RETURN

dump.hist:
  CALL MOVETO (10, 30)
  FOR i = 1 TO num.elts - 1
    PRINT "history("; i; ") = "; history(i)
  NEXT i
  RETURN

sh.nxtelt:
  CALL MOVETO (10, 30)
  PRINT "next.element = "; next.element;
  RETURN

sh.nxtb:
  CALL MOVETO (10, 30)
  PRINT "next.btn = "; next.btn;
  RETURN

sh.nxt.edt:
  CALL MOVETO (10, 30)
  PRINT "next.edt = "; next.edt;
  RETURN

sh.num:
  CALL MOVETO (10, 30)
  PRINT "num.elts = "; num.elts;
  RETURN


change.grid:
  INPUT "Grid size "; grid.grain
'  fix.x = (mfix.x \ grid.grain) * grid.grain
'  fix.y = (mfix.y \ grid.grain) * grid.grain
  GOSUB redraw.box
  RETURN

noop:
quit:
  '---------------------------------------
  ' Shut everything down in an orderly fashion
  '---------------------------------------
  END

stage2.menus:
  FOR mn = 1 TO 5 : MENU mn, 0, 0, "" : NEXT mn
```

```
   elem.menu = 1
   MENU elem.menu, 0, 1, "Elements"
   MENU elem.menu, 1, 1, "Push Button"
   MENU elem.menu, 2, 1, "Check Box"
   MENU elem.menu, 3, 1, "Radio Button"
   MENU elem.menu, 4, 1, "Edit Field"
   MENU elem.menu, 5, 0, "-"
   MENU elem.menu, 6, 2, "Decorative Text"
   next.element = 6

   act.menu = 2
   MENU act.menu, 0, 1, "Actions"
   MENU act.menu, 1, 1, "Add Element"
   MENU act.menu, 2, 1, "Generate Basic"
   MENU act.menu, 3, 1, "Change Grid Size"
   MENU act.menu, 4, 1, "Delete Element"
   MENU act.menu, 5, 2, "Debug"
   MENU act.menu, 10, 1, "Quit"

   help.menu = 3
   MENU help.menu, 0, 1, "Help!"
   MENU help.menu, 1, 1, "General"

   GOSUB activate.debug

   RETURN

activate.debug:
   debug.menu = 4
   MENU debug.menu, 0, 1, "Debug"
   MENU debug.menu, 1, 1, "Dump History"
   MENU debug.menu, 2, 1, "Show: next.element"
   MENU debug.menu, 3, 1, "Show: next.btn"
   MENU debug.menu, 4, 1, "Show: next.edt"
   MENU debug.menu, 5, 1, "Show: num.elts"
   debug = -1
   RETURN

deactivate.debug:
   MENU debug.menu, 0, 0, ""
   debug = 0
   RETURN

ok:
   wide = WINDOW (2)
   high = WINDOW (3)
   BUTTON maxbuttons+1, 1, "Ok", (wide - 40, high - 30) - (wide - 10, high - 14)
```

```
  WHILE -1
    d = DIALOG (0)
    IF d =1 THEN RETURN
  WEND

clear.dialog.queue:
  WHILE DIALOG (0) <> 0 : WEND
  RETURN

fix.axis:

  gosub clear.dialog.queue
  ms = MOUSE (0)
  mx1 = MOUSE (1) : my1 = MOUSE (2)
  IF grid.is.on THEN x1 = (mx1\grid.grain)*grid.grain : y1 = (my1\grid.grain)*grid.grain

  WHILE -1

    GOSUB get.lines
    GOSUB draw.axes
    xold = x1 : yold = y1
    IF MOUSE (0) <> 0 THEN freeze
    GOSUB erase.axes
    GOSUB redraw.screen
    mx1 = MOUSE (1) : my1 = MOUSE (2)
    IF grid.is.on THEN x1 = (mx1\grid.grain)*grid.grain : y1 = (my1\grid.grain)*grid.grain

   WEND

freeze:
  fix.x = x1 : mfix.x = mx1
  fix.y = y1 : mfixy = my1
  RETURN

erase.axes:
  LINE (x1, 0) - (x1, 350), 30
  LINE (0,y1) - (500,y1), 30
  RETURN

redraw.screen:
  PUT (x1, 0) - (x1, 350), y.axis, OR
  PUT (0,y1) - (500,y1), x.axis, OR
  RETURN

draw.axes:
  LINE (x1, 0) - (x1, 350)
  LINE (0,y1) - (500, y1)
```

```
  RETURN

get.lines:
  GET (x1, 0) - (x1, 350), y.axis
  GET (0, y1) - (500, y1), x.axis
  RETURN

redraw.box:
  CLS
'  FOR btn = 1 TO next.btn-1
'     BUTTON CLOSE btn
'  NEXT btn
'  FOR edt = 1 TO next.edt - 1
'     EDIT FIELD CLOSE edt
'  NEXT edt
  CALL TEXTFACE (1)
  FOR nt = 1 TO next.note - 1
    CALL MOVETO (nx (nt) * grid.grain + 1, ny (nt) * grid.grain - 4)
    PRINT notes$ (nt);
  NEXT nt
  CALL TEXTFACE (0)
  FOR btn = 1 TO next.btn - 1
    b.x0 = bx0 (btn) * grid.grain + 1
    b.x1 = b.x0 + bx1 (btn) + 20
    b.y1 = by1 (btn) * grid.grain
    b.y0 = b.y1 - by0 (btn)
    BUTTON btn, 1, btitle$ (btn), (b.x0, b.y0) - (b.x1, b.y1), btype (btn)
  NEXT btn

  FOR edt = 1 TO next.edt - 1
    f.x0 = fx0 (edt) * grid.grain + 1
    f.x1 = f.x0 + fx1 (edt)
    f.y1 = fy1 (edt) * grid.grain + 3
    f.y0 = fy1 (edt) * grid.grain - fy0 (edt) - 3
    EDIT FIELD edt, ftext$ (edt), (f.x0, f.y0)-(f.x1, f.y1), 1
  NEXT edt

  RETURN
```

THE MOTIVATION BEHIND DIALOG DESIGNER

I have always been annoyed by all the rectangles involved in Macintosh programming. Almost everything you deal with on the screen, and not just graphics, requires you to specify one of those pesky things. With a screen that is 512 pixels by 342 pixels, it takes quite a while to get used to the landscape. If you want to plunk down a row or column of buttons or edit fields, mentally calculating all those coordinates can be mentally exhausting, and you quickly discover that you don't get it right the first try—at least I don't.

So you wind up going through a painful trial and error process—I call it pixel bingo!

One day I said to myself, "Why don't I use the Macintosh itself to do this work for me?" One thing led to another and eventually I wound up with the program I am about to describe for you. In the process, I wrote a program that does more than just calculate numbers. It illustrates in a limited way some of the ideas behind interactive screen design and BASIC program generation.

Dialog Designer also demonstrates the difficulty of designing programs such as itself. You will no doubt find Dialog Designer lacking in many respects. You will think of a million and one things that you would like it to do that it doesn't. You are likely to take issue with the way it goes about its job. You will wish it did things differently, that it was less restrictive, or that it was more restrictive, and so on. If so, good! The idea here is that you are going to be writing your own applications. What better way to start than by doing Dialog Designer the way you want it to be. As in many other chapters, I will be making my own suggestions for possible changes, improvements, enhancements, and so on.

THE STARTUP PROGRAM

There are actually two programs in this chapter. The first is of somewhat limited interest. It was written to act as a starter for the dialog box design process. Its job is simply to allow the user to specify the size of the dialog box being designed. The drag open rectangle utility discussed earlier is used to specify the rectangle. After I wrote the program, I decided that specifying the dialog box first might turn out to be too limiting. The user might not have a good idea of how big the final box would be. So the second program now uses a box that pretty much takes up the whole screen. Its size can be changed after the BASIC program that reproduces the dialog box is generated.

The remaining point of interest in Listing 25-1 is the help menu there. It was the inspiration for the programs in Chapters 17 and 18. The help messages are implemented using BASIC print statements. They wind up taking an enormous amount of space in the program. This space is better used for code that can't be replaced by subsidiary information in a disk file.

PROGRAM OPERATION

Let me first describe exactly what the program does. By reading this section, you will learn how to use Dialog Designer to produce dialog boxes and forms for your own programs. Dialog Designer has four menus:

Elements **Help**

Actions **Debug**

The **Debug** menu is not always present, but may be swapped in and out using the appropriate choice in the **Actions** menu. You won't ever need it unless you want to change or enhance the program.

The Elements Menu

This menu always has exactly one item checked. The choices in the menu represent the kinds of elements you can put into your dialog box. They are:

Push Button	**Edit Field**
Radio Button	**Decorative Text**
Check Box	

Whichever choice is checked in this menu will be the kind of element you get when you choose **Add Element** from the **Actions** menu.

The Actions Menu

The program in Listing 25-2 offers six choices:

Add Element	**Delete Element**
Generate Basic	**Activate/Deactivate Debug**
Change Grid Size	**Quit**

Between the fifth and last choice there is space in the menu for four additional choices. I will be suggesting several additions in the last section of this chapter.

Add Element . When you choose this item, you are first prompted for text. The text will be used in different ways depending on what kind of element you are adding. For buttons, it is the title of the button. For edit fields, it is the default value of the edit field contents. For decorative text, it is the text itself. After you type in the text you want to use, the flying crosshairs of Chapter 11 appear on the screen. You move them around with the mouse until you find a spot on the screen where you want your new element to be located (Fig. 25-1). If there are already elements on the screen, the crosshairs help you to line up the new element with old elements.

Generate Basic . When you make this choice, Dialog Designer generates a BASIC program that will reproduce your form exactly as you see it on the screen (Fig. 25-2). The program goes into a text file that can later be loaded into BASIC and saved as a BASIC program document. You can use the form in your applications. If you want to "tweak" the code, you are perfectly welcome to change any of the statements generated by Dialog Designer.

Change Grid Size . This choice allows you to change the spacing of elements on the screen. All elements are placed at coordinates that are multiples of the grid "grain." By forcing this on the user, it makes it much easier to produce neat forms in which everything lines up exactly. You don't have to rely on your eyes or your hand-eye coordination. The program does it for you as long as you get close. The default grid grain built into the program is 10. This choice allows you to change that if you like.

Delete Element . If you use Dialog Designer, you will rapidly understand why this choice was included. Even if you plan everything in advance, you will find that it is easy to make mistakes. This choice gives you a chance to get everything right, eventually. If at first you don't succeed, . . .

The Help! Menu

This is a vestigial menu as it stands. It provides only the promise of help but no actual substance. You might say that Dialog Designer helps those who help themselves. Of course,

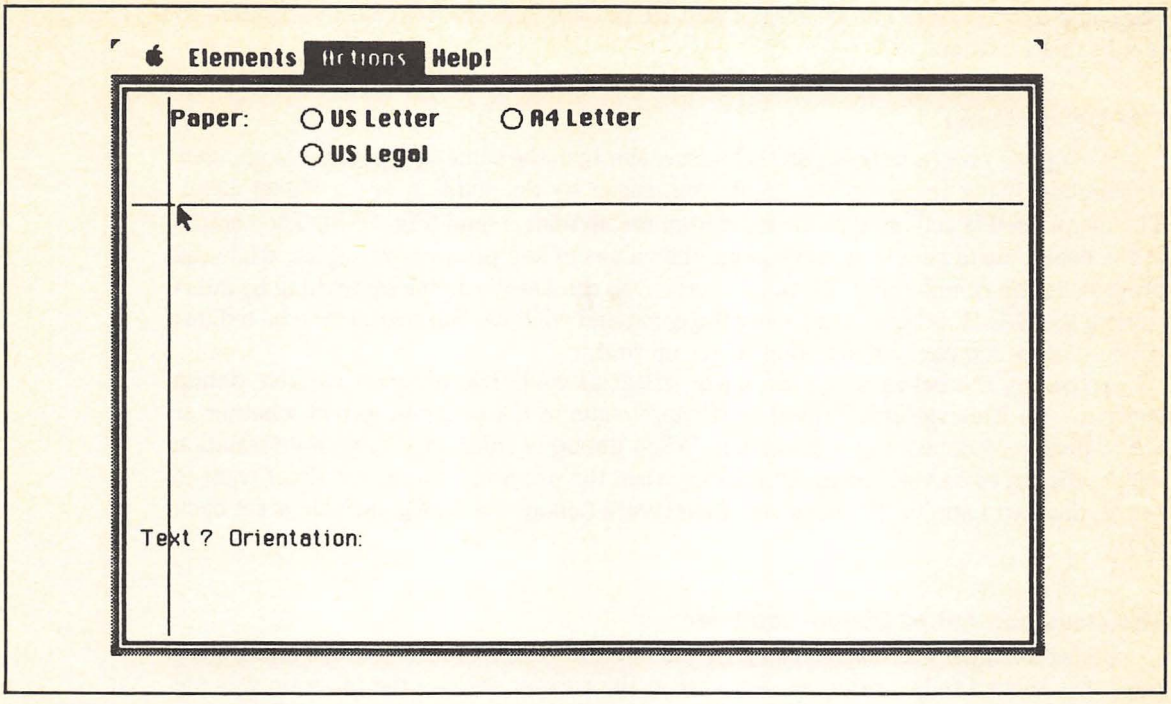


Fig. 25-1. Positioning a text element with text "Orientation."

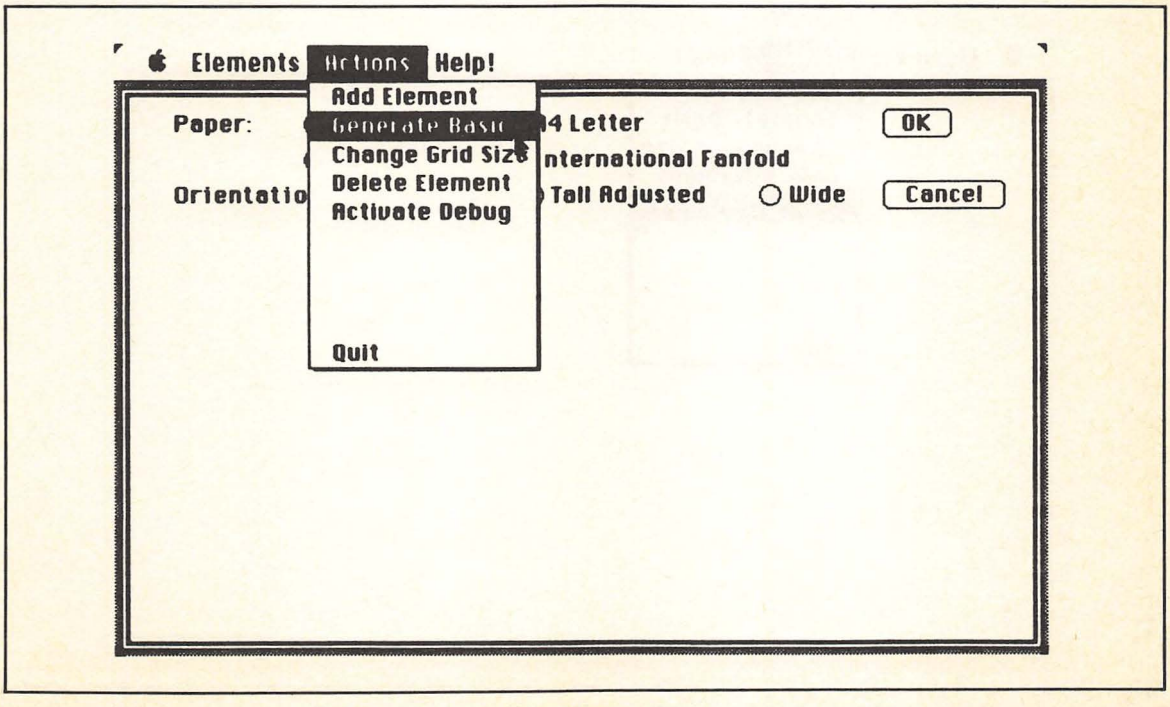


Fig. 25-2. About to generate a BASIC program for the dialog box being constructed.

you could use the Help File Generator and Help Code Support from earlier chapters to rectify this situation.

The Debug Menu

Debugging programs in BASIC is much easier than the same task in other languages. Debugging Dialog Designer was made even easier by the addition of the **Debug** menu. The menu itself is activated by a choice from the **Actions** menu (Fig. 25-3). The choices in the **Debug** menu allow you to examine the values of key program variables, while the program is still running (Fig. 25-4). Of course, you can always do the same thing by interrupting the BASIC program and using the command window. But that gets to be tedious if you have a complicated situation to set up first.

Activating the **Debug** menu has a side effect as well. The program variable **debug** is set to −1. This variable is used in IF statements in the program to test whether or not to dump out debugging information. When **debug** is equal to −1, such information will be displayed on the screen. Of course, when the program works, you don't want to see all this extra output. So when you **Deactivate Debug**, the debug variable is set back to 0.

The Development of Dialog Designer

Dialog Designer was both challenging and fun to develop. There were interesting questions that needed to be answered. Arriving at the current state of the program was not a simple matter of doing the first thing that came into my head. I worked many long hours

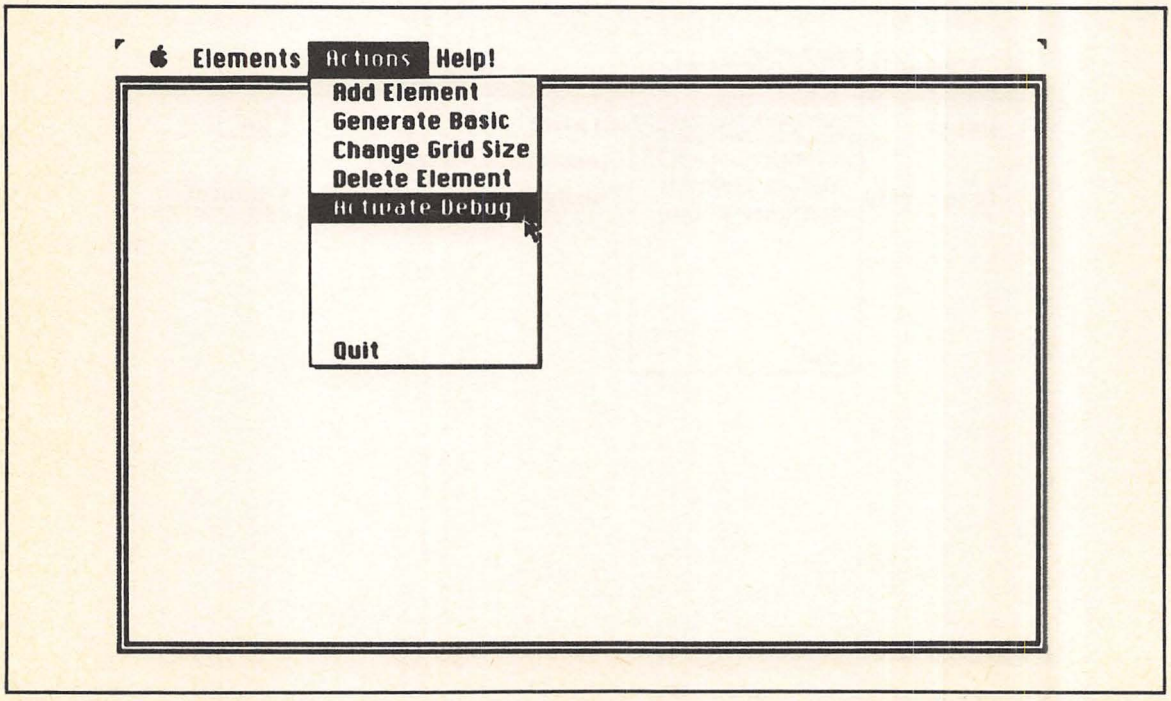


Fig. 25-3. Activating the debug menu.

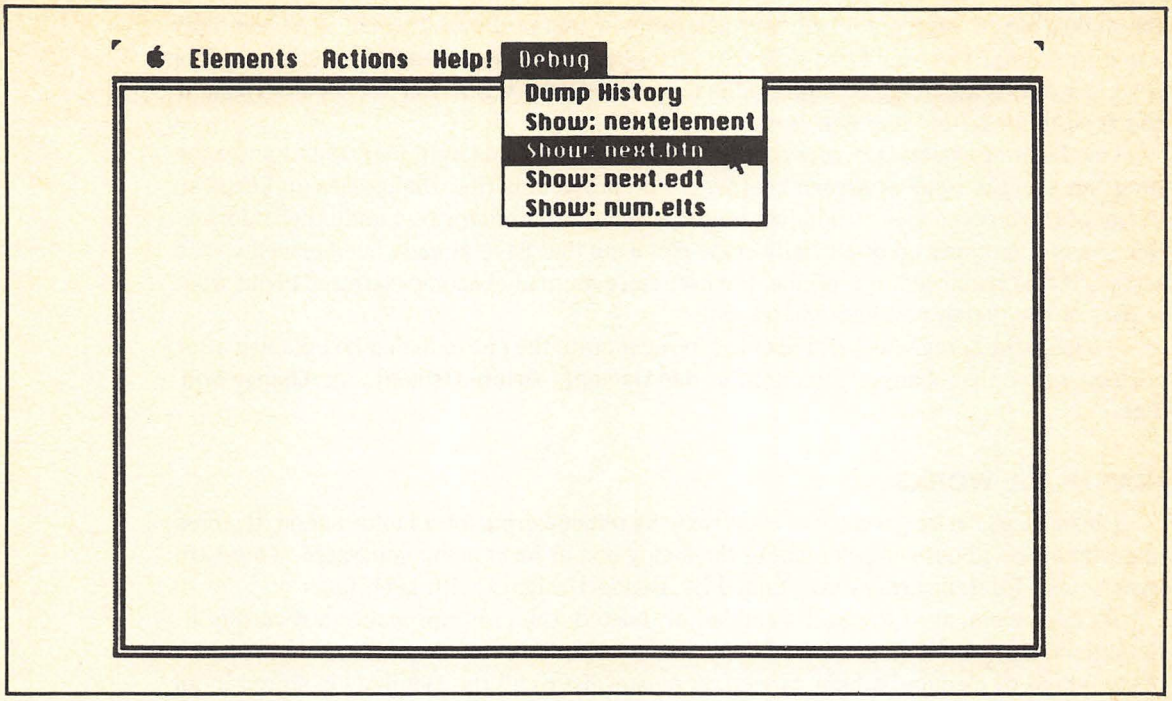


Fig. 25-4. About to display a variable's value (while the program is still running).

refining the program and trimming it down. At one point, there were seven menus! And the program was much harder to use.

The first and most fundamental problem was this: "How should the user specify to the program the size and position of the elements to be added to the dialog box?" My first thought in this regard was the drag open rectangle technique. This had the advantage of flexibility—the user could put an element anywhere and its rectangle could have any size desired. So I wrote version one using the drag open rectangle technique. I quickly became disappointed with the result—it was difficult if not impossible to produce several elements of the same height or width. It was also difficult to line several elements up in a row or column. One goal I had in mind when I started Dialog Designer was to be able to reproduce the typical dialog box produced by the **Print** item contained in most **File** menus. I could not get close with Version 1 of the program.

So it was back to the drawing board. I mulled the problem over and eventually came up with the idea of fixing the horizontal and/or vertical position at which elements would be placed. I wrote a subprogram to accomplish that task. It allowed you to position one axis with the mouse. The fixing of x and y axes required two separate operations. I rewrote the program with this approach and played with it awhile. I realized quickly that I was always using both axes at the same time. Thus, the program always did two operations when one would have sufficed. This led me to combine the fixing of the axes into the crosshairs approach. The original use of crosshairs in Dialog Designer was the drag crosshairs variety. I finally hit upon the idea of flying crosshairs and decided that it was easier to use.

Another problem was how to specify the text that accompanied various elements of

the dialog box. I have not spent as much time on this problem. Instead, I have simply opted to prompt the user for a new text string each time **Add Element** is chosen. This is simple, but is annoying at times. This is especially so when you use **Delete Element** and then immediately use **Add Element** with the same text.

One facet of Dialog Designer that did not become obvious until I started to write the program was the issue of screen neatness. The problem is that Dialog Design keeps an image of the box the user is building on the screen. Prompts for text and other information sometimes cover up or partially erase elements that have already been placed on the screen. If this is allowed to continue, the user can eventually become confused about what is part of the dialog box and what is not.

To keep the screen neater, I decided to regenerate the entire dialog box display after each operation that changed its contents: **Add Element**, **Delete Element**, or **Change Grid Size**

HOW IT ALL WORKS

Dialog Designer keeps track of everything by remembering lots of information. It stores important facts about each element of the dialog box or form being generated. There are twenty one different arrays maintained by Dialog Designer with such facts.

As each element of the form is added or deleted, the program makes a record of it. Enough information is remembered to be able to reproduce the element exactly. A count of each kind of element is kept as well. A master list of all the elements is stored in an array called **history**. Each entry in this list is a number that tells what kind of element was added:

Push Button	1
Check Box	2
Radio Button	3
Edit Field	4
Text	6

These numbers correspond to the numbering of the items in the **Elements** menu.

Consider the buttons, for example. Each button added to the dialog box has a title string, a type (radio button, push button, check box), and four numbers representing the corners of its enclosing rectangle. The arrays **btitle$, btype, bx0, by0, bx1,** and **by1** store this information. A variable called **maxbuttons** is used to determine how many buttons are allowed. If you have trouble getting Dialog Designer to run on a 128K Macintosh, try lowering **maxbuttons** from 50 to 25 or even less. The variable **nent.btn** is used as in index for all of the aforementioned arrays. Its value is always equal to the next free slot in the arrays. It is also the number of the next button to be generated. Therefore it is initially set equal to 1, not 0.

The use of a grid, as mentioned earlier, makes it easier to align elements in straight rows and columns. The flying crosshairs technique must be modified in order to implement this. The point where the crosshairs cross is always forced to be a grid point. This is a point whose coordinates are evenly divisible by the value of the variable **grid.grain**. This value is set equal to 10 when the program begins, but may be changed by the user. Figures 25-5 through 25-7 illustrate the same dialog box before and after a change to the grid size. The crosshairs are forced to a grid point by the following assignments:

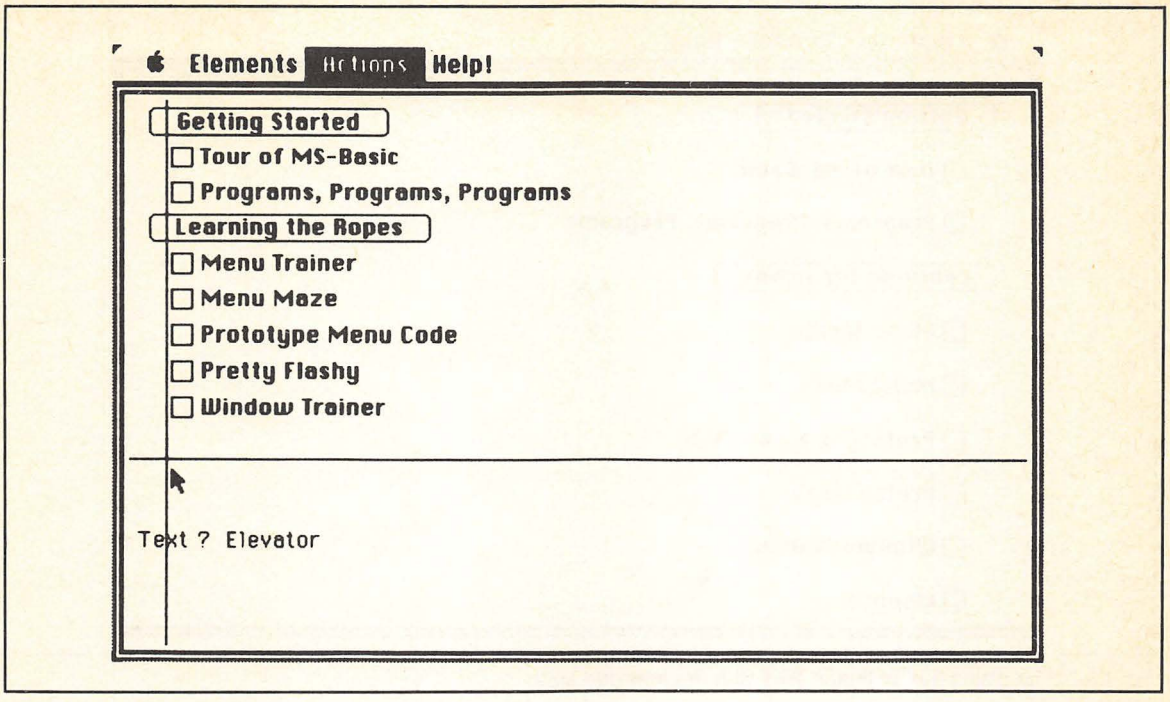


Fig. 25-5. Positioning a new element with the crosshairs.

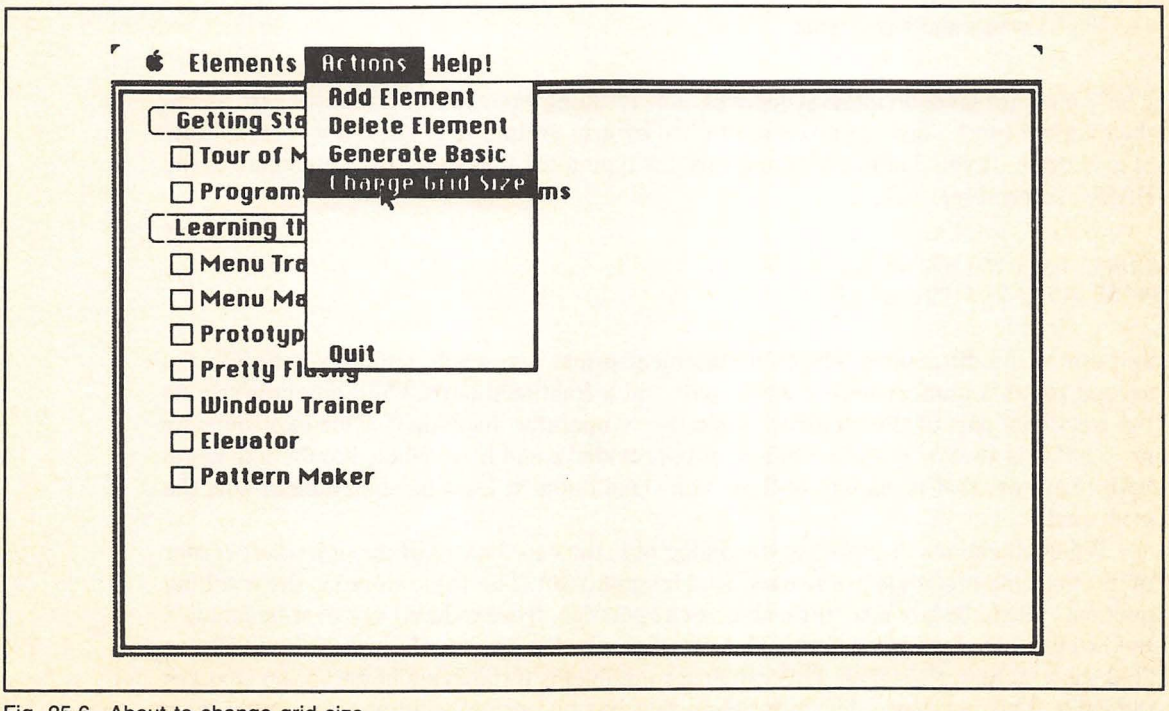


Fig. 25-6. About to change grid size.

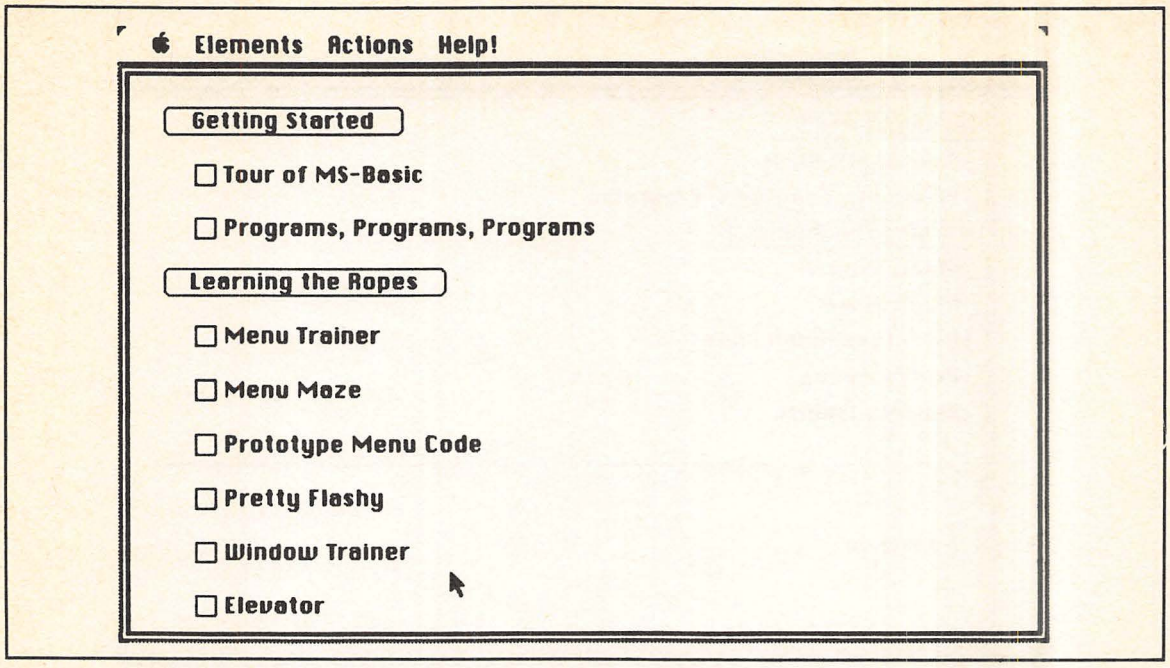


Fig. 25-7. The dialog box of Figure 25-1 with the new grid size.

```
x1 = (mx1 \ grid.grain) * grid.grain
y1 = (my1 \ grid.grain) * grid.grain
```

The \ operator gives an integral quotient. The resulting value has no fractional part. When the quotient (mx1 \ grid.grain) is multiplied by grid.grain, the result is a whole multiple of grid.grain. If you don't understand this, try typing the following two statements to the BASIC Interpreter:

```
PRINT (100 \ 7), (100 \ 7) * 7
PRINT (100 / 7), (100 / 7 ) * 7
```

and notice the difference. The first statement prints two whole numbers, whereas the second prints a number with a whole part and a fractional part. The / operator retains the fractional part of the quotient, while the \ operator discards it. This explains why (a \ b) * b is always an even multiple of b, provided a and b are whole numbers to begin with. Anyway, that is enough of that. You should now at least have an idea of how the grid works.

When an element is added to the dialog box, the coordinates of the upper left corner of its enclosing rectangle are also reduced by **grid.grain**. The value stored is the resulting quotient. When the box is redrawn after each operation, these reduced values are multiplied by the **current** value of **grid.grain**. This explains why the boxes of Figure 25-1 and Figure 25-3 show different spacing. This can lead to problems if the value of **grid.grain** changes radically. This is a basic design problem that has no perfect solution. You might wish

to experiment with a different approach: you could store the **adjusted** values of the coordinates rather than the **reduced** values. The adjusted value of x0 is the value of the expression: (x0 \ grid.grain) * grid.grain. The reduced value of x0 is the value of the expression: (x0 \ grid.grain).

GENERATING A BASIC PROGRAM

When the user is satisfied with the appearance of the dialog, it is time to preserve it for future use. This is done by generating a BASIC program. Dialog Designer prompts for the name of a file to contain the program. The program is generated as a text document, which may later be read into the BASIC interpreter and saved as a BASIC program document if so desired.

The program is generated using PRINT #1 statements. To give you the flavor of how this is done, consider a typical BUTTON statement:

```
BUTTON 4, 1, "Standard", (211, 4) - (294, 20), 3
```

Some of this statement may be generated literally. For example, the PRINT statement:

```
PRINT #1, "BUTTON ";
```

will generate the string: "BUTTON" into the text file. Other parts of the statement must be generated using values stored in variables. For the parts of the BUTTON statement, these are as follows:

button-number	btn (index into button information arrays)
button-title	btitle$ (btn)
button-rect	b.x0, b.y0, b.x1, b.y1 (values calculated using the numbers stored in bx0 (btn), by0 (btn), bx1 (btn), by1 (btn)
button-type	btype (btn)

These variables are used in PRINT statements, mixed in with literal strings to produce the desired result.

There is one remaining trick to generating a BUTTON statement. Notice that the BUTTON statement contains a pair of quotation marks, ". . . ." surrounding the button-title. Unfortunately, quotation marks in a PRINT statement are already used to surround literal text to be printed. So the question becomes: "How do you generate quotation marks into a text file with a PRINT command?" The answer is : "Use the CHR$ function." The expression CHR$ (34) when used in a PRINT statement produces a double quotation mark character.

Now putting all of the above information together, I arrive at the BASIC statements needed to generate a BUTTON statement into the text file:

```
PRINT #1, "BUTTON ";
PRINT #1, btn;
PRINT #1, ", 1, ";
PRINT #1, CHR$(34);
```

```
PRINT #1, btitle$(btn);
PRINT #1, CHR$(34);
PRINT #1, ",(";
PRINT #1, b.x0,
PRINT #1, ",";
PRINT #1, b.y0;
PRINT #1, ")-(";
PRINT #1, b.x1;
PRINT #1, ",";
PRINT #1, b.y1;
PRINT #1, "),";
PRINT #1, btype(btn)
```

Some of these statements are combined in Dialog Designer to save space.

The EDIT FIELD statements are generated in much the same fashion. Text is generated as strings into DATA statements, and a subroutine that reads the strings into an array is created in the target program. Then the program generates a FOR NEXT loop that uses CALL MOVETO (. . ., . . .) and PRINT statements to position and print the strings in the dialog box.

The BUTTON statements and EDIT FIELD statements in the generated program are all individually generated. You might consider changing Dialog Designer to generate DATA statements for the BUTTON and EDIT FIELD information and then generate a pair of FOR—NEXT statements to create the BUTTONs and EDIT FIELDs.

The following two programs were generated by Dialog Designer. They roughly reproduce the **Page Setup** and **Print...** dialog boxes found in the **File** menu in many programs (e.g. MacWrite):

Program Listing 25-3
Generated Basic for Print Setup Dialog Box

```
WINDOW 2,null$,(11,31)-(501,331), 2
DIM notes$(2)
GOSUB read.notes
CLS
CALL TEXTFACE (1)
CALL MOVETO (21,16)
PRINT notes$(1)
CALL MOVETO (21,56)
PRINT notes$(2)
CALL TEXTFACE (0)
BUTTON 1,1,"US Letter",(91,4)-(179,20), 3
BUTTON 2,1,"US Legal",(91,24)-(172,40), 3
BUTTON 3,1,"A4 Letter",(201,4)-(290,20), 3
BUTTON 4,1,"Tall",(121,44)-(170,60), 3
BUTTON 5,1,"International Fanfold",(201,24)-(374,40), 3
BUTTON 6,1,"Tall Adjusted",(211,44)-(329,60), 3
BUTTON 7,1,"Wide",(341,44)-(395,60), 3
```

```
BUTTON 8,1,"OK",(411,4)-(449,20), 1
BUTTON 9,1,"Cancel",(411,44)-(478,60), 1
END
read.notes:
FOR i=1 TO 2
  READ notes$(i)
NEXT i
RETURN
DATA "Paper:"
DATA "Orientation:"
```

Program Listing 25-4
Generated Basic for Print Dialog Box

```
WINDOW 2,null$,(9,29)-(503,333), 2
DIM notes$(5)
GOSUB read.notes
CLS
CALL TEXTFACE (1)
CALL MOVETO (11,16)
PRINT notes$(1)
CALL MOVETO (11,36)
PRINT notes$(2)
CALL MOVETO (11,56)
PRINT notes$(3)
CALL MOVETO (11,76)
PRINT notes$(4)
CALL MOVETO (311,36)
PRINT notes$(5)
CALL TEXTFACE (0)
BUTTON 1,1,"High",(111,4)-(162,20), 3
BUTTON 2,1,"All",(111,24)-(153,40), 3
BUTTON 3,1,"Continuous",(111,64)-(209,80), 3
BUTTON 4,1,"Standard",(211,4)-(294,20), 3
BUTTON 5,1,"From:",(211,24)-(270,40), 3
BUTTON 6,1,"Cut Sheet",(211,64)-(300,80), 3
BUTTON 7,1,"Draft",(311,4)-(369,20), 3
BUTTON 8,1,"OK",(401,14)-(439,30), 1
BUTTON 9,1,"Cancel",(401,64)-(468,80), 1
EDIT FIELD 1,"",(115,42)-(144,60), 1
EDIT FIELD 2,"",(275,22)-(304,40), 1
EDIT FIELD 3,"",(345,22)-(374,40), 1
END
read.notes:
FOR i=1 TO 5
  READ notes$(i)
```

```
NEXT i
RETURN
DATA "Quality:"
DATA "Page Range:"
DATA "Copies:"
DATA "Paper Feed:"
DATA "To:"
```

WHAT NEXT, WHAT ELSE, AND WHAT BESIDES?

Dialog Designer offers many possibilities for change and enhancement. I have already hinted at some of the features that you might decide to change. Let me take this last section to suggest some additions to the program.

The current implementation does not allow the user to turn off the grid. I suppose you could set grid.grain equal to 1, but that is not really the point. It would be better to allow some elements to use the grid and others to ignore it. In addition, it would be a good addition to allow separate grains for the x and y axes. Currently, Dialog Designer uses the same value for adjusting both x and y values.

If you add grid and nongrid options, allow them to be mixed in the program. This requires remembering which elements were specified with the grid on and which were not. If x and y have separate grids, you have to remember separately whether the x grid or the y grid was on.

A more convenient redo facility is another enhancement to consider. As the program is now, if you don't like what you did, you have to go through the follcwing steps to redo it: **Delete Element**, **Add Element**, answer text prompt, and reposition axes. A **Redo** action could collapse all these into fewer steps and make life considerably easier on the user.

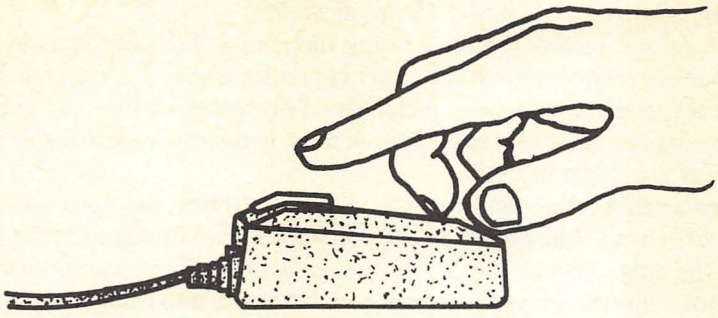
Dialog Designer now stores coordinates in reduced form when the grid grain is on. It recalculates the actual values each time an element is regenerated and when the BASIC code is produced. This is not necessary. It was done, however, with the idea that the reduced values could be put into the generated program instead of the actual coordinates. Then, the same generated program could produce the same dialog box with different grains for different spacing. It is not clear how useful this might turn out to be. You might devote some more thought to the issue and decide whether you want to change to storing the actual coordinates used to display the elements each time.

There is currently no way to edit the dialog box after it is built. To change any item, you have to delete all the way back to it and rebuild the dialog box from there onwards. It would be nice to be able to select elements of the dialog box and edit them without this awkward approach. I have not devoted any effort to this task, so I am not sure if it is easy, medium, difficult, or nigh to impossible.

Here is a super challenge: generate the BASIC code to conduct the dialog represented by the dialog box. There are specification issues here—exactly what actions should be taken when various events are detected? This information could be provided to Dialog Designer in some way during the dialog box creation process.

Dialog Designer itself could use dialog boxes. You could use the current form of the program to "bootstrap" a better interface: use it to generate one or more dialog boxes to replace some or all of the **Elements** and **Actions** menus. Then those dialog boxes could be integrated into a new version of Dialog Designer.

Chapter 26



Now It's Your Turn

The time has finally come to bring this book to a close. I hope that you have had as much fun reading it as I did writing it. As I mentioned earlier, I have not been able to exhaust even so narrowly defined a topic as Interactive Programming Using MS-BASIC on the Macintosh. There are many topics that I wished I had had the time and space to include. There are many programs that I wanted to write and present to you that I didn't get around to working on.

But then if I had written all the programs I could think of, I wouldn't have any left to suggest for you. The rest of this chapter is devoted to a list of ideas for you to pursue. Of course, you will have lots of your own, but just in case you need a few to get you started, here are a few.

MORE MOUSING AROUND

There are any number of mouse techniques that come to mind that you could try to program in BASIC. I hope you have been pleasantly surprised already by what you can accomplish with the MOUSE function and its use in conjunction with other commands such as GET and PUT. Here are several more techniques that you might try out:

- See if you can write a set of subroutines that will allow you to drag the contents of a rectangle around the screen. This is easy enough with GET and PUT until I add the constraint that you must preserve the screen over which the rectangle passes. That is, suppose the rectangle is dragged across part of the screen, momentarily obscuring it. When the rectangle leaves that part of the screen, the original contents of the screen

should reappear. This is what happens, for example, in MacPaint when you drag a selected region of the screen around. I have only made half-hearted attempts at this and failed miserably. That doesn't mean it is difficult! My first attempts at the drag open rectangle code were a pitiful failure as well.

- Think about further techniques for locating the mouse. Consider situations in which you have multiple rectangles that are either not uniform in size or not contiguous. This means that the modular arithmetic techniques I presented earlier may not be easy to apply. Also consider situations in which you must detect the mouse inside some other shape such as a triangle or circle.
- Can you write a subroutine that uses the mouse to "throw" an object on the screen. The idea is to detect a drag operation and use some sort of timing to detect the relative velocity of the drag. Also by measuring the angle of the line joining the starting and ending points, a direction may be determined. If you are into calculus, you might even try to integrate and find the position function determined by such a throw. These kinds of techniques could be used in writing games.
- Experiment further with analog controls (see Rose Patterns for an example). One obvious Macintosh control that you could try to simulate would be a scroll bar. You might not necessarily need to invoke the BASIC SCROLL command. Can you think of simple programs in BASIC in which scroll bars might be effectively used?
- Finally, I wanted to write a maze tracing game. The idea was that the walls of the maze were electrified so that you had to avoid them at all costs. The mouse would be used to trace the maze. The program would have to constantly monitor the mouse and detect whenever any of the walls were touched. This could turn out to be quite a challenging project, especially in the areas of setting up the maze, representing it in memory, and mapping the cursor location on the screen into the internal representation in "real time." You might try starting with a simple straight corridor that must be traced from left to right or top to bottom. Then branch out into more complicated mazes.

GRAPHICS AND THE MOUSE

The drag open rectangle utility of Chapter Eleven is a simple technique that could be put to use in a graphics setting. When I set out to write this book, I originally wanted to write a BASIC Paint program that used this and other ideas to implement a subset of MacPaint. I just never got around to that project, but maybe you will have more time and more ambition.

- Try freehand drawing and pens first. The ingredients of the use of patterns in a program are already presented in Pattern Maker and Mouse Kaleidoscope. You could use the style of Cursor Editor to make things a bit more convenient—those buttons do get tiresome after awhile!
- Erasing should be reasonably easy. First create an eraser shape to use as a cursor. Then you could use ERASERECT to erase the parts of the screen that the eraser cursor touches with the mouse button held down. Make the cursor hot spot the upper left corner. Then the rectangle for the ERASERECT call should be easy to calculate. But wait, that's all I'm going to say. Any more and I'll wind up writing the code for you!
- Saving the pictures you create can be done with GET and PUT format arrays and files of integers. If you have a 128K Mac things are going to be tougher on you. You won't

have enough memory to GET the whole screen into one array. There are two approaches you can take. You can limit the size of the pictures you allow to be drawn to those that may be plucked off the screen with a single GET, or you can break up a larger picture into multiple parts. The first alternative may limit you to pretty tiny pictures. The second alternative could be fiendishly difficult to get right. I leave it to you. By all means, consider a 512K upgrade as a third alternative!

- Limited fat bits capabilities have already been presented in the Cursor Editor application. See if you can interface it to your BASIC Paint efforts. The tricky part is deciding how to specify what part of the screen goes into the fat bits image. The limitation, of course, is that you won't be able to move things around with the hand. If you can manage that in BASIC with acceptable speed, then quit your job and become a professional programmer!
- The output from your BASIC Paint programs could be put to use in creating animations. The technique is described in the MS-BASIC Interpreter manual in the discussion of GET and PUT.

PERSONAL APPLICATIONS

Your local computer store is flooded with programs for the Macintosh. Many of these fall into the category of personal applications. There are things like simple filing systems, desk calendars, appointment books, tax preparation programs, money management aids, games of all sorts, and so on. These commercial programs are orders of magnitude slicker than what you could ever hope to achieve in BASIC. But the cost is a magnitude higher, and perhaps you don't need all the fancy capabilities some of these programs offer. So, do it yourself. You probably won't use three quarters of the features in the commercial versions anyway. And by writing your own application, you will learn a great deal, you will end up with something, and you will achieve a tremendous sense of satisfaction. Here are some of the types of programs you may want to write:

- Desk calendars. Just figuring out how to display a single month and get the days of the week lined up correctly can be fun. Deciding on the graphic display you want leaves scope for originality and the personal touch. Beyond a simple picture of the calendar, the possibilities are too manifold to mention. For inspiration, look at the advertisements for some of the popular commercial offerings.
- Simple filing systems. Here is an area where you might want to attempt a "one-shot" program or two. Instead of a general purpose of filing system, maybe you just want to keep track of a few recipes, a stamp collection, or your round robin bridge tournament records. This is an area in which the form filling techniques barely introduced in Chapter 10 need to be extended.
- Study aids. Programs similar in spirit to Vocabulary Quiz could be used by students. Even better would be a program or collection of programs that assisted in organizing material for study. This could perform many tasks, such as transcribing class notes, or creating special purpose data bases or electronic flash cards.

CONVERSION OF EXISTING BASIC PROGRAMS

I wanted to include at least one program that was converted from an existing BASIC program. I worked for a long while on a Macintosh version of the game Camel that ap-

pears in popular collections of BASIC games. I even got a version to more or less work, but somehow I just was not satisfied with it. I put it on the back burner and never got back to it. I did learn enough from the attempt to make some suggestions.

I still think that converting existing BASIC programs to run on the Macintosh is a worthwhile endeavor. I would strongly suggest that you not attempt to make a literal translation. Instead, start with the idea of the program you are going to convert. Then write an original implementation of that idea from scratch. Make use of the interactive programming features and techniques that you have gleaned. I intend to take the idea in Camel and write an original implementation for the Macintosh. Perhaps it will be called Desert or maybe Oasis.

In addition to converting published BASIC programs, take some of your own, if you have written BASIC programs on other computers. Concentrate on using the Mac features, but also try to improve the structure of your code. Rework the organization to take advantage of such statements as WHILE . . . WEND. Make use of the identifiers as labels and write more subroutines with names that suggest their function in the program. But above all, have fun.

MORE FUN AND GAMES

There is a whole world of fun awaiting you with Macintosh games. The Ups 'N Downs puzzle is just the beginning. I have also written but did not have space to include a sliding blocks puzzle. This is similar to the puzzle contained in the Macintosh desk accessories. The idea is to maneuver a large square block and several smaller square and rectangular blocks around the inside of a tray. The large block is supposed to start at the middle of the top of the tray and end up in the middle of the bottom of the tray. The mouse is used to drag pieces from one spot to another in the playing area. I have also started writing and collecting "mouse games" in BASIC.

There are arcade games waiting to be invented and written. How about Mac Man? The idea of using a large array of buttons in a space ship like control panel has occurred to me as a potentially good start for a Star Trek style game. How about Elevator Adventure? Starting with the Elevator example program, make up an adventure. One idea would be that you are in a department store and have to collect a certain item from each floor.

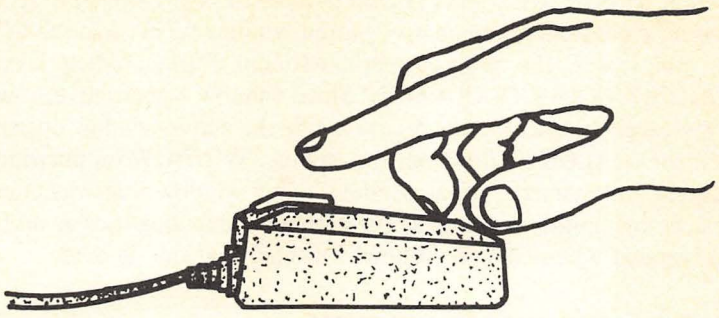
THE SOUNDS OF SILENCE

I really goofed here. There is absolutely no coverage of the SOUND and WAVE commands. Rather than give it a half-hearted coverage, I decided to omit it entirely. Maybe in the next book . . . Meanwhile, the sample program delivered on the MS-BASIC Interpreter disk provides a good starting point. It is a little bit difficult to figure out given the comments contained in it, but if you persist and experiment a lot, you should be able to modify it to play tunes other than Jesu, Joy of Man's Desiring. I managed to program the First Noel before Christmas (I got my copy of MS-BASIC on Christmas Eve).

DECIMAL OR BINARY

Every one of the programs in this book was written using the Binary version of the interpreter. I have not explored the use of decimal arithmetic at all. If you have a business minded bent and are interested in manipulating large monetary figures, then by all means work with MSB-d.

Chapter 27



Challenges Revisited

In this chapter I return to some of the challenges issued during the course of the book. A very small selection of these challenges will be explained and answered here. For the others, I am afraid that you are on your own. Many of them were more in the order of suggestions for further work on the programs in the book. Others I have not attempted to solve myself, but I thought they would be interesting and challenging, so I posed them.

Each section in this chapter discusses one or more challenges. The section heading includes the chapter number in which the challenge was originally given.

A SMALL MATTER OF INTERPRETATION–CHAPTER SIX

Now you can return to the bug in Pretty Flashy. After writing the program, I noticed that under certain circumstances, the window containing the delay value (window number 4) would fail to respond to the mouse. The failure to respond always took place after the **Input** choice was processed from the menu provided that the previously flashing window was *not* window number 4. That made some sense at least, since the input of the delay value was processed in a new incarnation of window 4 with an edit field. After that the old window 4 was restored in the upper right hand part of the screen.

As soon as one of the other three windows was selected, the Delay window would begin to respond again. This was baffling. Finally, out of sheer desperation or strictly by accident (I forget which), I changed the statement WINDOW OUTPUT outputwindow to WINDOW outputwindow. Ah hah! No more problems with window 4, but I was still too dumb to see why! Have you figured it out yet? If so, congratulations: if not, don't feel bad.

Here is the answer: when you say WINDOW outputwindow, you make "outputwindow" both the ACTIVE and the OUTPUT window. This means that window 4 is no longer the active window. It becomes the active window automatically when it gets recreated in the upper right part of the screen. (You don't have to say WINDOW OUTPUT in order for it to later be considered an inactive output window.) This was the link that I was missing! Boy did I sing the programmer's national anthem when I realized that, OOOOOOOOOOOOOOOOOOOOOOOOH! Since window 4 was active, it would not respond to the mouse click. Clicking the mouse inside the active window does not generate a value of 3 for the DIALOG (0) function. But saying "WINDOW outputwindow" makes window number 4 an "inactive output window"—any window that was once active and has been replaced by another window as the active window qualifies as an inactive window and will respond when clicked. Whew! I'm glad that one is over.

CHANGING CURSORS—CHAPTER ELEVEN

Program Listing 27-1 contains a short program that randomly changes cursor shapes whenever the mouse moves inside or outside a rectangle set up by the program. The program uses a pair of variables to keep track of where the mouse is now and where it was the last time it was checked. These variables can take on any value between 0 and −4. The value −4 represents the situation when the mouse is inside the rectangle. All the other values represent various instances of the mouse being outside the rectangle.

The subroutine where.mouse calculates the value of **where**. This represents the location of the mouse now. Before calculating the new value of **where**, the old value is saved in the variable **oldwhere**. The program uses the values in **where** and **oldwhere** to detect transitions from outside to inside the rectangle and vice versa. This is accomplished with the two IF statements that appear at the end of the main WHILE loop of the program:

```
IF (where=inside) AND (oldwhere>inside) THEN GOSUB new.cursor
IF (where>inside) AND (oldwhere=inside) THEN CALL INITCURSOR
```

The first IF statement detects the transition from outside to inside. When that happens, a cursor is chosen at random from the cursor data included at the end of the program. The second IF statement detects the opposite transition. When the cursor moves outside the rectangle it is restored to the standard pointer by a call to the ROM routine INITCURSOR.

All the cursors used when the mouse goes inside the rectangle were constructed using Cursor Editor from Chapter 22.

Program Listing 27-1
Changing Cursors

```
'-----------------------------------------------------------------
' Changing Cursors
'
' Written by
' Richard C. Vile, Jr.
' March 25, 1985
```

```
DEFINT a-z
numcurs = 4
DIM cursors(33 * numcurs)

RANDOMIZE TIMER

WINDOW 1, "", (0,0)-(512,330), 3
GOSUB getcursors
re(0) = 100 : re(1) = 100 : re(2) = 225 : re(3) = 300
CALL FRAMERECT (VARPTR (re(0)))
re(0) = re(0) + 1 : re(1) = re(1) + 1
re(2) = re(2) - 1 : re(3) = re(3) -1
CALL FRAMERECT (VARPTR (re(0)))

 inside = -4
 delay = 1000

 x = MOUSE (1) : y = MOUSE (2)
 GOSUB where.mouse
 oldwhere = where

 WHILE -1
   FOR de = 1 TO delay : NEXT de
   ms = MOUSE (0)
   x = MOUSE (1) : y = MOUSE (2)
   GOSUB where.mouse
   IF debug CALL MOVETO (10, 300): PRINT where, oldwhere;
   IF (where = inside) AND (oldwhere > inside) THEN GOSUB new.cursor
   IF (where > inside) AND (oldwhere = inside) THEN CALL INITCURSOR
 WEND

 where.mouse:
   oldwhere = where
   where = (x >=100) + (y >= 100) + (x <= 300) + (y <= 225)
   RETURN

new.cursor:
  IF debug CALL MOVETO (10, 316) : PRINT "In new.cursor";
  wc = INT (RND * numcurs)
  CALL SETCURSOR (VARPTR (cursors(wc*33)))
  RETURN

getcursors:
  FOR i = 0 TO numcurs - 1
    READ cursorname$ (i)
    FOR j = 0 TO 33
```

```
      READ cursors(i*33 + j)
    NEXT j
  NEXT i
  RETURN

DATA Plus
DATA 0,0,0,0
DATA 16368,16368,15600,15600
DATA 12336,12336,15600,15600,16368,16368,0,0
DATA -1,-1,-1,-1,-1,-1,-1,-1
DATA -1,-1,-1,-1,-1,-1,-1,-1
DATA 8,8

DATA X
DATA 0,0,12300,14364,7224
DATA 3696,2016,960,960
DATA 2016,3696,7224
DATA 14364,12300,0,0
DATA 0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0
DATA 8,8

DATA Beams
DATA 0,256,8456,8176,8456,256,256,640
DATA 0,640,256,256,8456,8176,8456,256
DATA 0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0
DATA 8,8

DATA Eyeball
DATA 0,2340,-15032,9552,8176,14280,20452,-25486,-25487
DATA -25486,20452,10184,5008,4064,0,0
DATA 0,0,0,0,0,0,0,0
DATA 0,0,0,0,0,0,0,0
DATA 1,1
```

A BETTER DOUBLE CLICK DETECTOR–CHAPTER TWENTY

Program Listing 27-2 shows an extended version of the double click detection technique. The idea is to call MOUSE (0) several times instead of just twice. If a value of 2 or −2 appears, a double click has been detected. This is better than simply calling MOUSE (0) after a delay and hoping that for a double click, the value will actually be 2 or −2. Actually a delay is used here too, but the probability of catching a double click is further increased through the collection of several values.

The values of MOUSE (0) are first collected into an array and then analyzed in a separate subroutine. The version of the program given in Listing 27-2 also checks for two single clicks and stops analyzing if it finds them. This prevents the pattern single click, slight pause, double click from being recognized as a "real" double click. If you think

that that pattern should be allowed, then simply take out the statements that deal with the variable **sawone** and it will be.

Program Listing 27-2
Double Click Analyze

```
'-----------------------------------------------------------------
' Double Click - Second Technique
'
' Written by
' Richard C. Vile, Jr.
' March 21, 1985
'-----------------------------------------------------------------
DEFINT a-z
DIM msq(10)

delay = 250
WHILE -1
  GOSUB click
  FOR i = 1 TO 10
    PRINT msq(i);" ";
  NEXT i
  PRINT
  GOSUB analyze
  IF double THEN PRINT "double-click" ELSE PRINT "single-click"
WEND

click:
  ms = MOUSE(0)
  WHILE MOUSE(0) = 0 : WEND
  FOR i = 1 TO 10
    msq(i) = MOUSE(0)
    FOR de=1 TO delay : NEXT de
  NEXT i
  WHILE MOUSE(0) <> 0 : WEND
  RETURN

analyze:
  double = 0 : sawone = 0 : sawzero = 0
  FOR i = 2 TO 10
    IF msq(i) = 1 THEN sawone = sawone + 1
    IF sawone >= 2 THEN RETURN
    IF ABS(msq(i)) = 2 THEN double = -1 : RETURN
  NEXT i
  RETURN
```

VARIABLE SIZE HELP WINDOWS--CHAPTER EIGHTEEN

After a bit of experimentation, I have found that the following formula works fairly well. Allow 16 pixels for each line of text in the message and 64 pixels extra. The extra accounts for the title and blank line following it and some white space at the bottom to accommodate the Ok button.

Using this formula in Quickdraw Trainer for example, you would change the window statement in the help: subroutine. Add the following statement:

```
btm = 30 + hlp.len (hn) * 16 + 64
```

and then the WINDOW command becomes:

```
WINDOW 2, "", (50,30) - (455, btm), 2
```

The BUTTON command in the ok: subroutine changes as well:

```
BUTTON 1, 1, "Ok", (360, btm - 60) - (390, btm - 40)
```

That's all there is to it.

Appendix A

Command/Program Cross Reference

You can use the figures in this appendix to quickly locate programs that illustrate specific BASIC statements. Each of the Figures A-1 through A-4 cross references a set of commands. The commands are listed across the top of the figure, and the names of sample programs are listed down the side. When a program uses a command there will be a • in the row and column of that command and program combination.

COMMANDS / PROGRAMS	MENU command	MENU	MENU RESET	MENU ON/ MENU OFF	MENU STOP	ON MENU GOSUB	MENU (0), MENU (1)	WINDOW command	WINDOW OUTPUT n	WINDOW CLOSE n	WINDOW (0)/(1)	WINDOW (2)-(5)	ON DIALOG GOSUB	DIALOG ON/OFF	DIALOG STOP
Menu Trainer	●	●	●	●	●	●	●								
Menu Maze	●	●	●	●	●	●	●								
Prototype Menu Handler	●	●		●		●	●								
Pretty Flashy	●	●	●	●	●	●	●	●	●	●	●		●	●	●
Window Trainer	●	●	●	●	●	●	●	●	●	●	●		●	●	
Elevator								●	●				●	●	
Pattern Maker								●		●					
Address Book													●	●	
Mouse Utilities															
Quickdraw Trainer	●	●	●	●	●	●	●	●		●					
Scribble and Video								●							
Rose Patterns								●							
Ups 'N Downs															
Fancy Mouse (0)															
Help File Generator															
Help File Support	●														
Cursor Editor															
Mouse Kaleidoscope	●	●	●	●		●	●	●							
Vocabulary Quiz	●	●	●	●	●	●	●	●		●		●			
Dialog Design (Startup)	●	●		●	●	●	●	●				●			
Dialog Box	●	●		●		●	●	●				●			

PROGRAMS \ COMMANDS	DIALOG (0)	DIALOG (1)	DIALOG (2)	DIALOG (3)	DIALOG (4)	DIALOG (5)	EDIT FIELD command	EDIT FIELD CLOSE n	EDIT$ (n)	BUTTON command	BUTTON (n)	BUTTON CLOSE n	MOUSE (0)	MOUSE (1) / (2)	MOUSE (3) / (4)
Menu Trainer															
Menu Maze															
Prototype Menu Handler															
Pretty Flashy	●			●		●	●	●	●	●					
Window Trainer	●			●	●	●							●		
Elevator	●	●								●					
Pattern Maker	●	●								●	●		●		
Address Book	●	●	●				●	●	●	●		●	●		
Mouse Utilities													●	●	●
Quickdraw Trainer	●									●		●			
Scribble and Video													●		
Rose Patterns													●		●
Ups 'N Downs													●		●
Fancy Mouse (0)													●		●
Help File Generator							●	●		●		●			
Help File Support	●									●		●			
Cursor Editor													●		
Mouse Kaleidoscope	●	●					●	●	●	●	●	●	●	●	
Vocabulary Quiz	●	●								●			●		
Dialog Design (Startup)	●	●								●					●
Dialog Box	●						●	●		●		●	●	●	

COMMANDS / PROGRAMS	MOUSE (5) / (6)	ON MOUSE GOSUB	MOUSE ON/OFF/STOP	ON TIMER GOSUB	TIMER ON/OFF/STOP	SCROLL	LINE/CIRCLE/PSET	GET	PUT	CALL ...RECT	CALL ...ROUNDRECT	CALL ...OVAL	CALL ...ARC	CALL ...POLY	CALL MOVETO/LINETO
Menu Trainer															
Menu Maze															
Prototype Menu Handler															
Pretty Flashy										●					●
Window Trainer															
Elevator						●				●					●
Pattern Maker										●					●
Address Book															
Mouse Utilities	●						●	●	●						
Quickdraw Trainer										●	●	●	●	●	●
Scribble and Video									●						●
Rose Patterns							●			●					●
Ups 'N Downs							●			●					●
Fancy Mouse (0)										●					●
Help File Generator															
Help File Support															
Cursor Editor	●						●			●					●
Mouse Kaleidoscope		●	●					●	●	●	●	●			
Vocabulary Quiz				●	●										●
Dialog Design (Startup)	●						●	●	●						●
Dialog Box							●	●	●						●

COMMANDS / PROGRAMS	CALL TEXTFONT/etc.	CALL PENSIZE	CALL PENPAT	CALL PENMODE	CALL BACKPAT	CLS	PICTURE command	PICTURE$	PICTURE ON/OFF	WAVE	SOUND command	SOUND WAIT/RESUME	RESTORE	RESUME	ON ERROR GOTO
Menu Trainer															
Menu Maze															
Prototype Menu Handler															
Pretty Flashy	●					●									
Window Trainer						●									
Elevator															
Pattern Maker			●			●									
Address Book						●									
Mouse Utilities						●									
Quickdraw Trainer	●	●	●		●	●							●	●	●
Scribble and Video		●	●		●	●									
Rose Patterns															
Ups 'N Downs	●														
Fancy Mouse (O)															
Help File Generator															
Help File Support															
Cursor Editor						●									
Mouse Kaleidoscope			●	●		●	●						●		
Vocabulary Quiz	●												●	●	●
Dialog Design (Startup)	●					●									
Dialog Box	●					●									

Appendix B

Program Highlights

The following list provides a quick reference to the major programming topics illustrated. Most of the programs in Sections 2 through 4 are listed.

Menu Trainer

- Expressions in a MENU command
- Read menu titles from DATA statements

Menu Maze

- Command key equivalents for menu items
- Pause until the mouse is moved

Prototype Menu Handler

- Menu handling techniques
- Menu code layout and the use of subroutines

Pretty Flashy

- Multiple windows in the same program
- Switching the output window
- An edit field with a modal dialog
- Dialog trap to detect a click in a nonactive window
- Refreshing windows

Window Trainer

- Multiple windows
- The output window as distinct from the active window
- Dialog trapping
- Large menus (note: won't run on 128K Mac)

Elevator

- Push buttons
- Large number of buttons
- The SCROLL command
- A dialog trap for a button push
- Drawing with FRAMERECT commands

Pattern Maker

- The use of radio buttons
- Mixing different kinds of buttons in the same dialog
- Remembering program information in button state (pattern)

Address Book

- Forms defined with lots of edit fields
- Dialog handling for form filling
- The EDIT$ function

Mouse Utilities

- The use of MOUSE (0) to detect simple click-unclick sequences
- xy axes which follow the cursor
- The use of GET and PUT to preserve the screen
- Detecting when the cursor is inside a rectangle
- An application of dragging to define a screen rectangle

Quickdraw Trainer

- A large number of menus (seven)
- An Application of the Help facility
- All combinations of graphic shapes and modes:

RECT	FRAME
ROUNDRECT	PAINT
OVAL	ERASE
ARC	FILL
POLYGON	INVERT

- Pen pattern and background pattern
- Pen patterns with FILL and LINETO

- PSET
- Pen height and width
- Text combinations

Scribble and Video

- Pens and patterns in combination
- MOVETO and LINETO to produce lines with patterns
- Reading patterns from DATA statements

Rose Patterns

- Program converted from Version 1.1
- Screen menu drawn with RECT and LINE
- Surrounding text with boxes (print text first)
- Which rectangle was clicked
- Simple analog control

Ups 'N Downs

- Omitting a check for unclick—the consequences
- The style of a 1.1 program
- Multiple-rectangle click detection revisited

Fancy MOUSE (0)

- Patterns of MOUSE (0) values over time
- Dependence on a delay to detect a double click

Help File Generator

- Edit fields with RETURN allowed
- Converting EDIT$ into an array of lines
- Random access techniques

Help Support Code

- Accessing the help data file via an index file
- The OK button as a "Click to continue . . ." equivalent

Cursor Editor

- Multiple rectangles—which one was clicked
- Tracking the cursor through rectangles during a drag
- The representation of GET and PUT arrays
- Building GET and PUT arrays from BASIC
- Modular arithmetic in rectangle click detection
- Dragging conventions: all pixels dragged across turn same color as the first one
- A self supporting program—bootstrapping a menu screen by using an earlier version of the cursor editor to create the pictures

- Simulating a menu with the screen display (Needed in Version 1.1)
- Relocating a grid, possible through the use of X0 and Y0.

Mouse Kaleidoscope

- A program too big for 128K—uses **IF mac512k THEN . . .**
- Entering changes to program parameters with edit fields
- The use of ON MOUSE to freeze screen or detect double click for CLS
- Buttons used in conjunction with edit fields
- The reuse of the earlier program—Pattern Maker
- A trick with check versus uncheck:

```
MENU 1, 1, 1 + ABS (shape = 1), "Square"
```

and so on.

Vocabulary Quiz

- A trick with ON ERROR GOTO for getting out of the quiz loop
- Leaving the RUN and WINDOWS menus in during debugging
- Multiple windows design issues
- ON TIMER used to time quiz questions
- Check boxes for answers
- The use of DATA tricks to READ and locate categories
- Dimming a large subset of menu items during the quiz

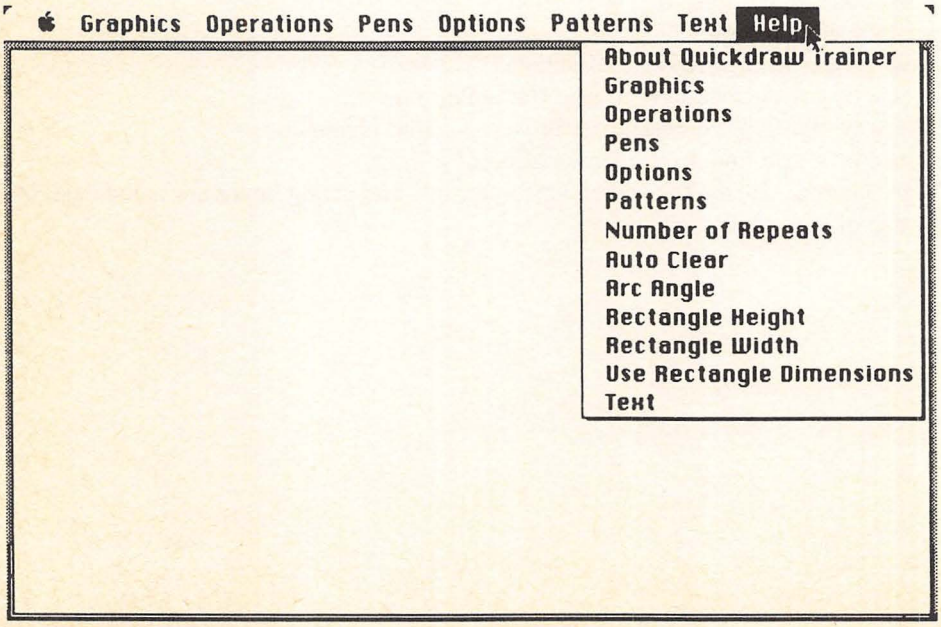
Dialog Design and Dialog Box

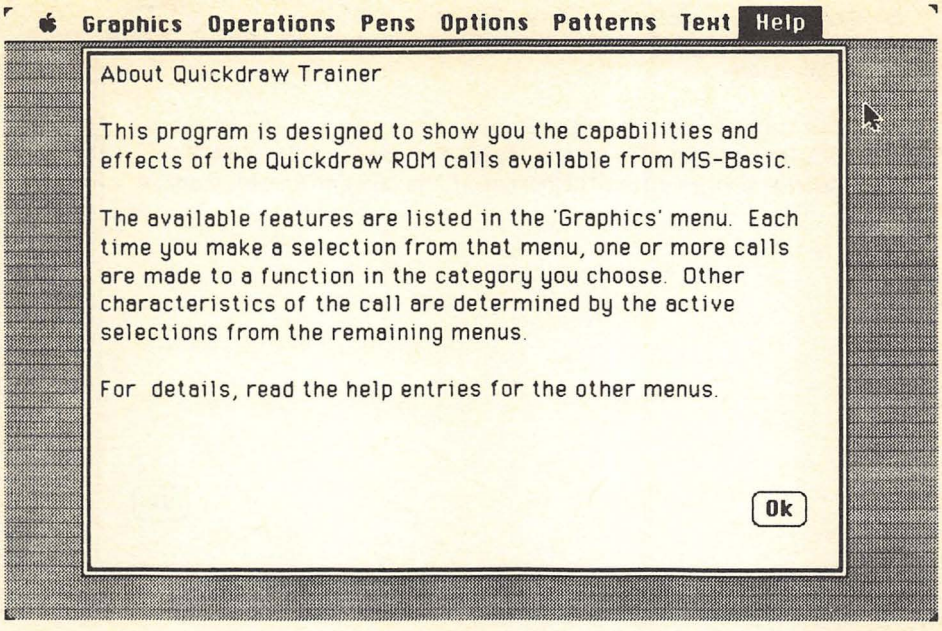
- BASIC code generation
- Startup program use
- A help screen using PRINT statements
- The use of a drag rectangle to size the dialog box
- The use of the flying crosshairs utility to position elements
- The use of a grid and an implementation of a grid
- The problem of sharing the screen when prompts and dialog boxes are being designed
- The use of clear.dialog.queue

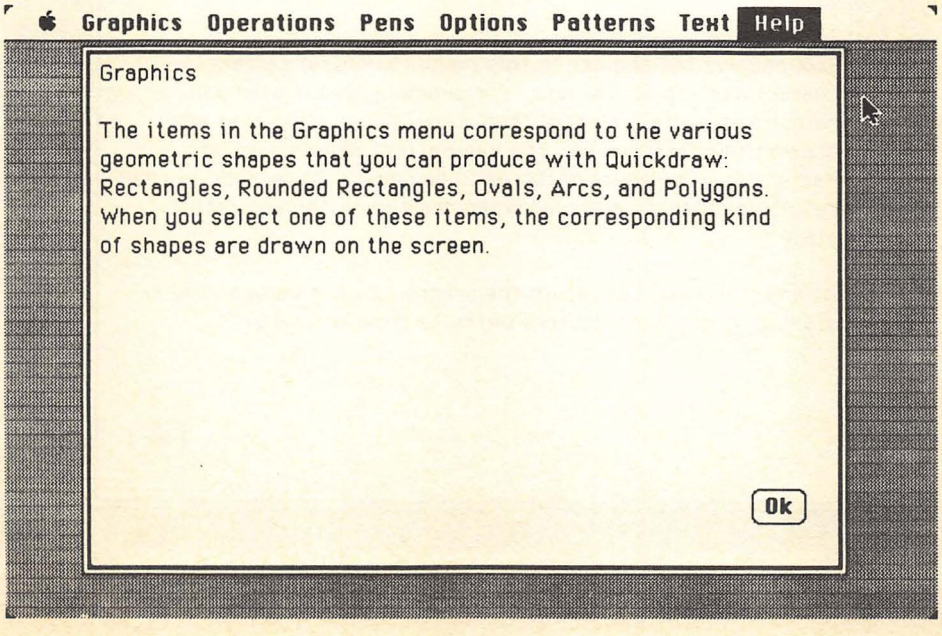
Appendix C

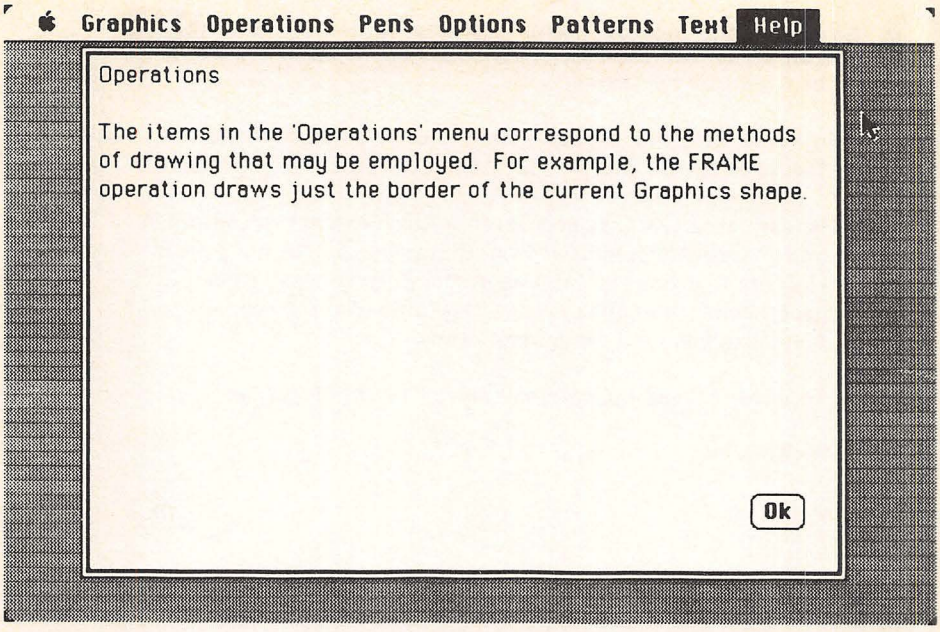
Help File Listings

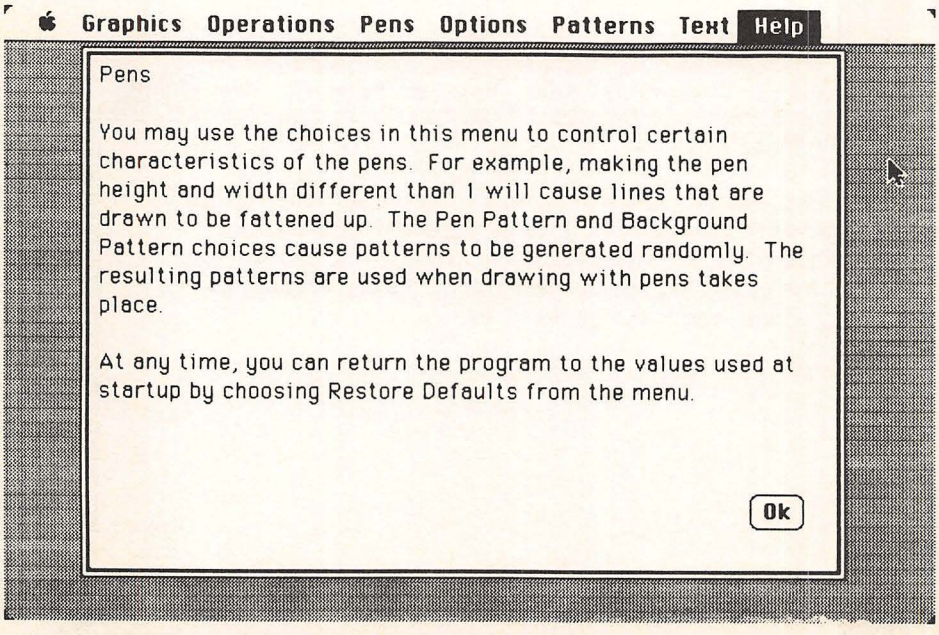
Figures C-1 through C-14 show the help screens for Quickdraw Trainer. By using Help File Generator, you can add this help information to the program.

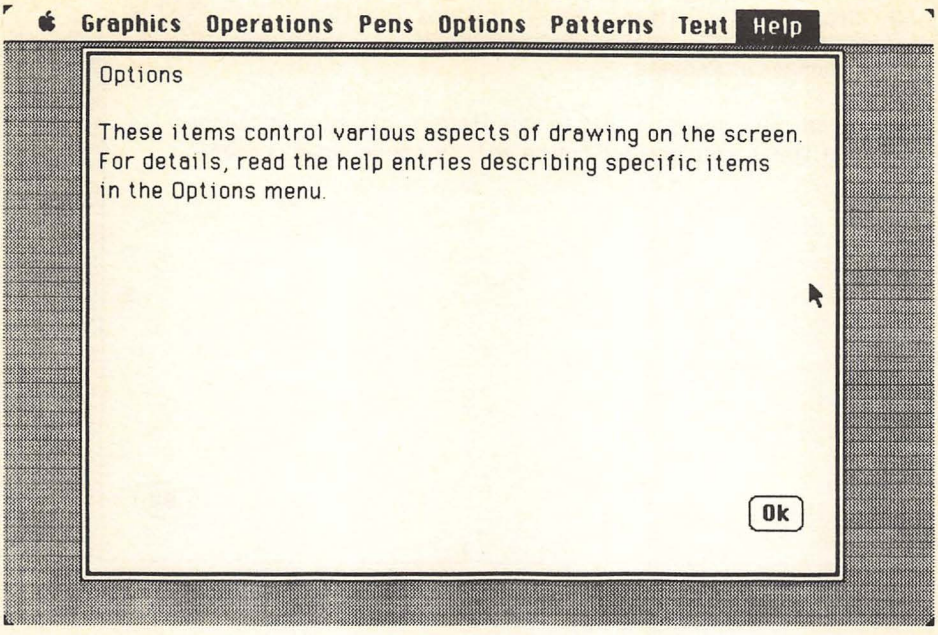


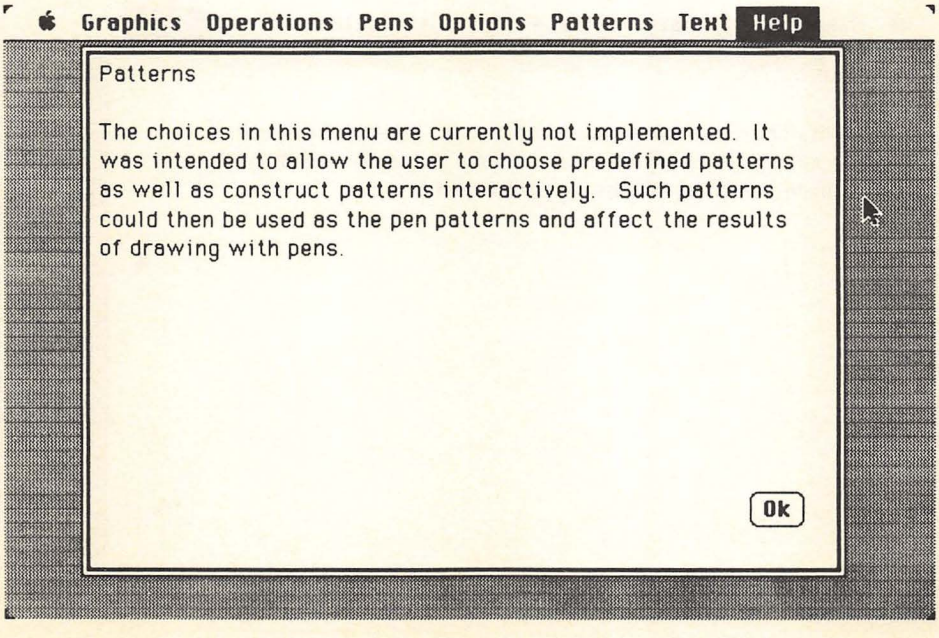

Graphics Operations Pens Options Patterns Text Help
About Quickdraw Trainer
This program is designed to show you the capabilities and effects of the Quickdraw ROM calls available from MS-Basic.
The available features are listed in the 'Graphics' menu. Each time you make a selection from that menu, one or more calls are made to a function in the category you choose. Other characteristics of the call are determined by the active selections from the remaining menus.
For details, read the help entries for the other menus.
Ok

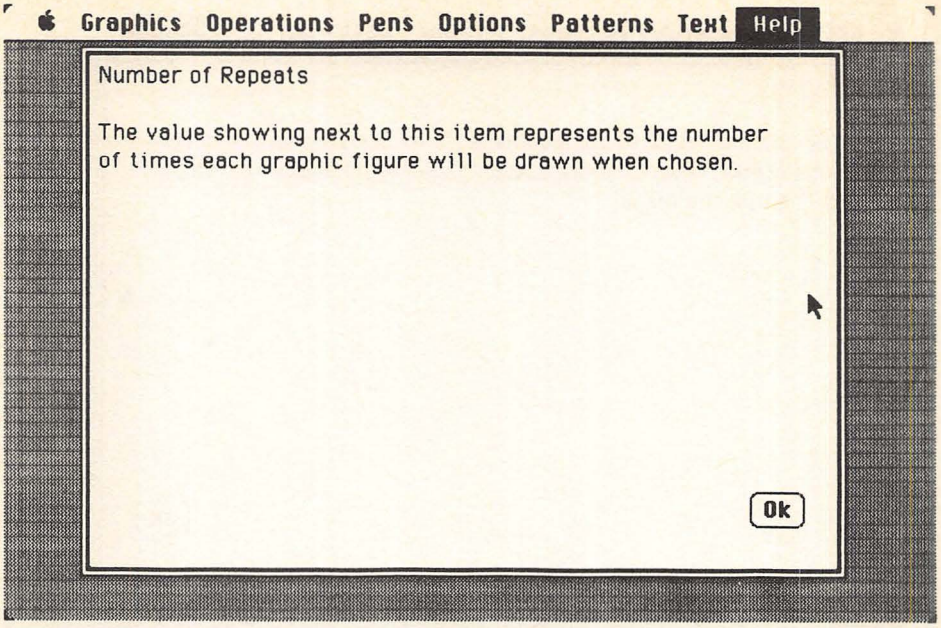

Graphics Operations Pens Options Patterns Text Help
Graphics
The items in the Graphics menu correspond to the various geometric shapes that you can produce with Quickdraw: Rectangles, Rounded Rectangles, Ovals, Arcs, and Polygons. When you select one of these items, the corresponding kind of shapes are drawn on the screen.
Ok

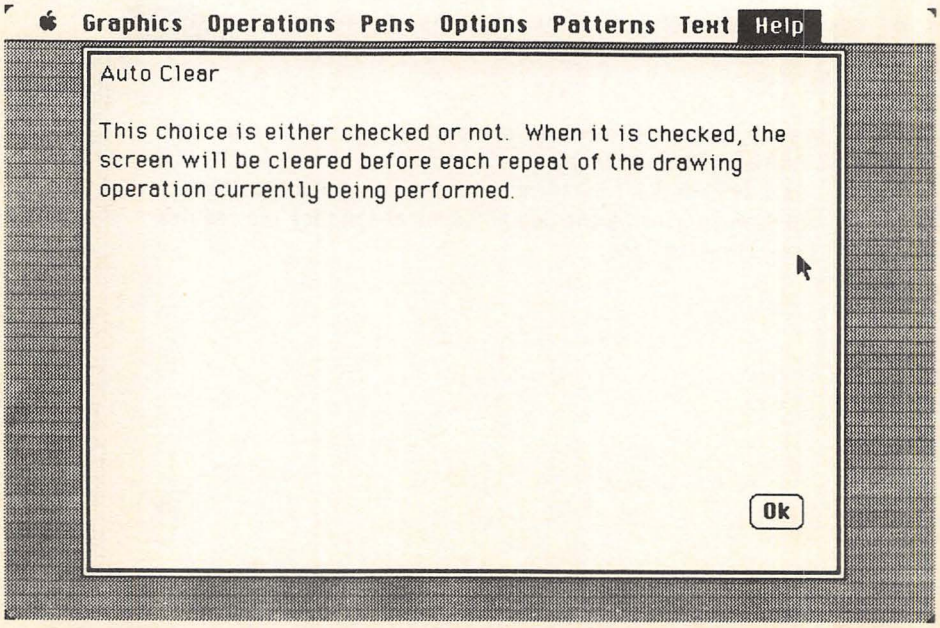

Graphics Operations Pens Options Patterns Text Help
Operations
The items in the 'Operations' menu correspond to the methods of drawing that may be employed. For example, the FRAME operation draws just the border of the current Graphics shape.
Ok

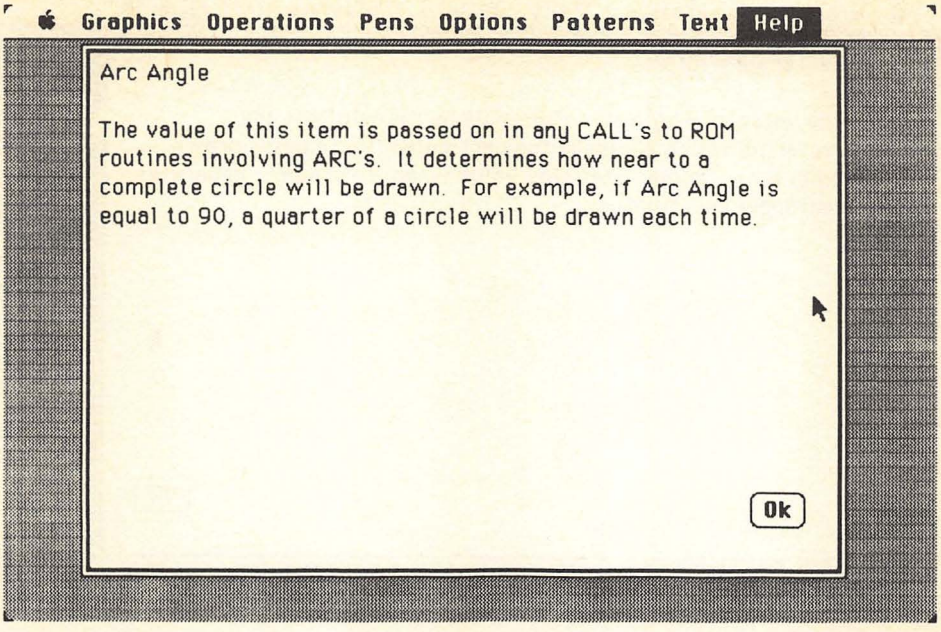

Graphics Operations Pens Options Patterns Text Help
Pens
You may use the choices in this menu to control certain characteristics of the pens. For example, making the pen height and width different than 1 will cause lines that are drawn to be fattened up. The Pen Pattern and Background Pattern choices cause patterns to be generated randomly. The resulting patterns are used when drawing with pens takes place.
At any time, you can return the program to the values used at startup by choosing Restore Defaults from the menu.
Ok

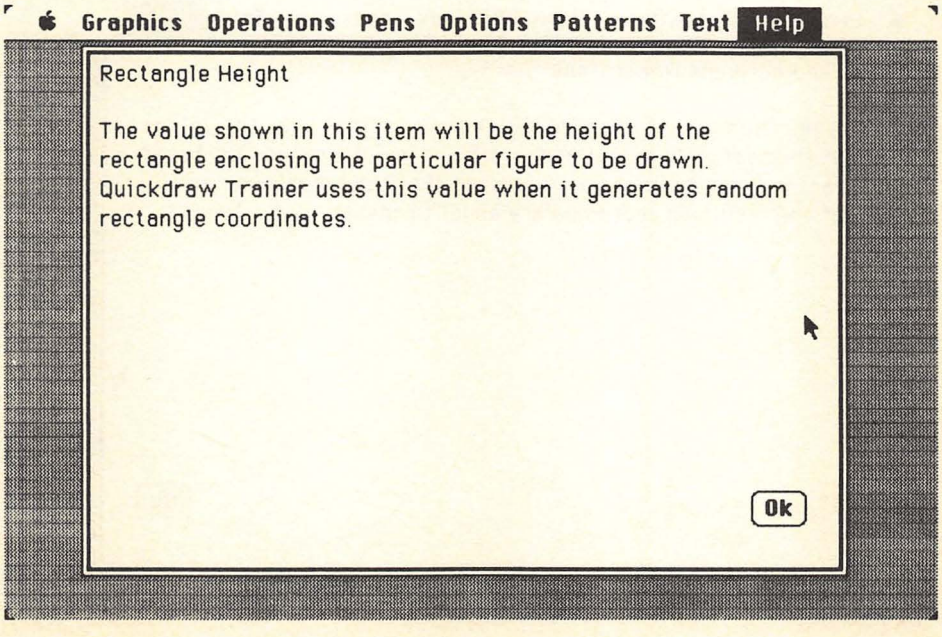

Graphics Operations Pens Options Patterns Text Help
Options
These items control various aspects of drawing on the screen. For details, read the help entries describing specific items in the Options menu.
Ok

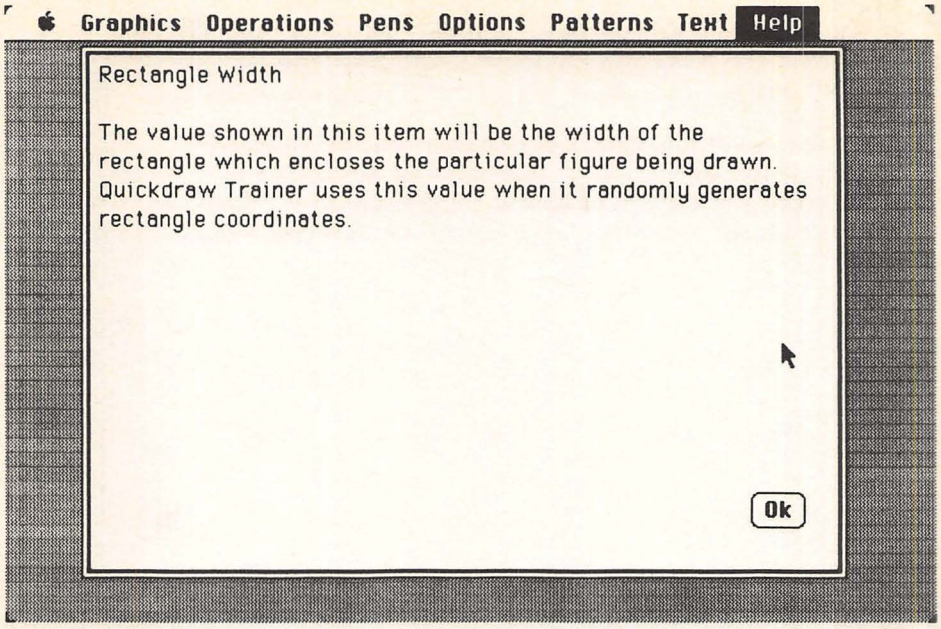

Graphics Operations Pens Options Patterns Text Help
Patterns
The choices in this menu are currently not implemented. It was intended to allow the user to choose predefined patterns as well as construct patterns interactively. Such patterns could then be used as the pen patterns and affect the results of drawing with pens.
Ok

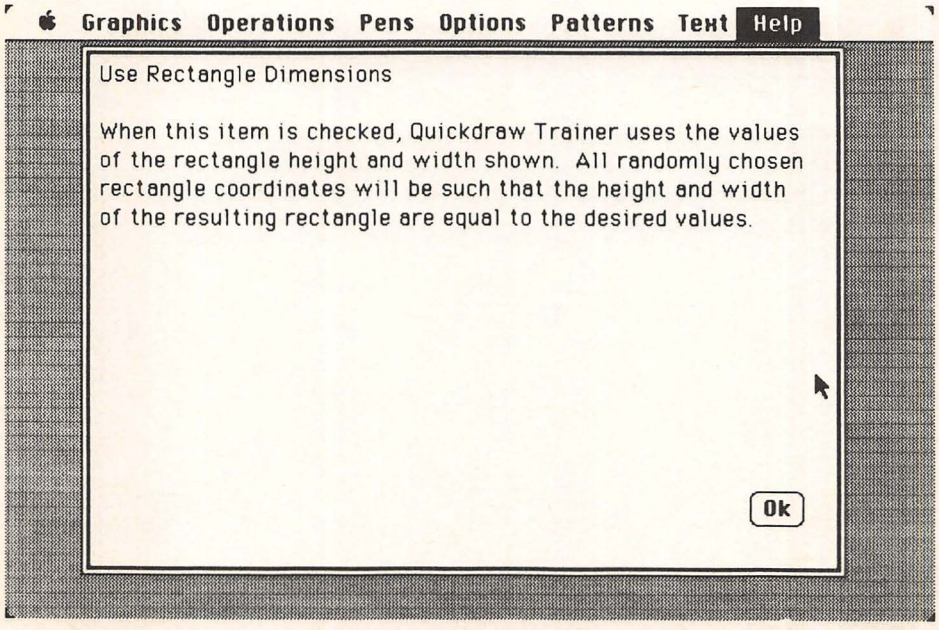

Graphics Operations Pens Options Patterns Text Help
Number of Repeats
The value showing next to this item represents the number of times each graphic figure will be drawn when chosen.
Ok

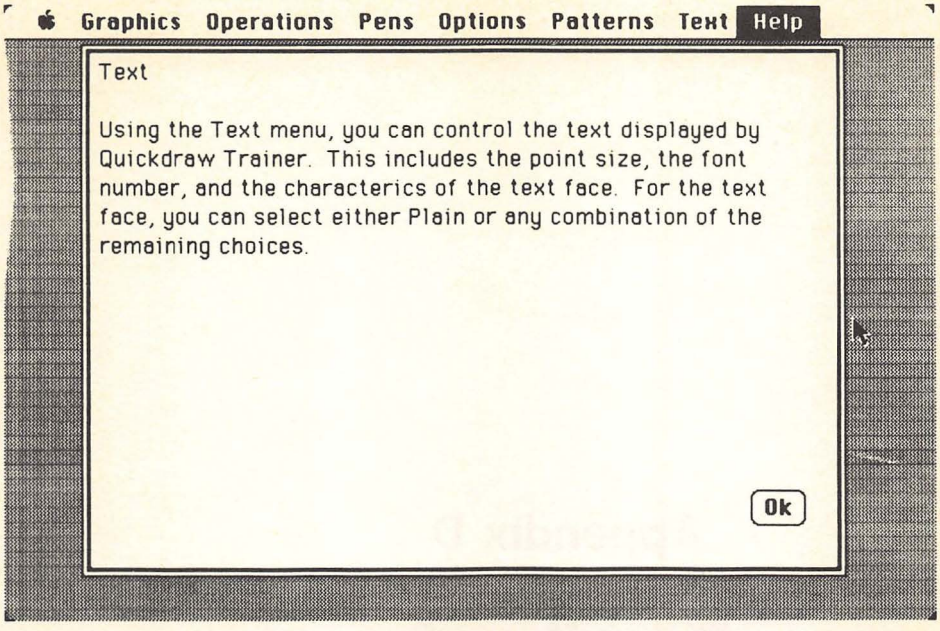

Graphics Operations Pens Options Patterns Text Help
Auto Clear
This choice is either checked or not. When it is checked, the screen will be cleared before each repeat of the drawing operation currently being performed.
Ok


Graphics Operations Pens Options Patterns Text Help
Arc Angle
The value of this item is passed on in any CALL's to ROM routines involving ARC's. It determines how near to a complete circle will be drawn. For example, if Arc Angle is equal to 90, a quarter of a circle will be drawn each time.
Ok


Graphics Operations Pens Options Patterns Text Help
Rectangle Height
The value shown in this item will be the height of the rectangle enclosing the particular figure to be drawn. Quickdraw Trainer uses this value when it generates random rectangle coordinates.
Ok


Graphics Operations Pens Options Patterns Text Help
Rectangle Width
The value shown in this item will be the width of the rectangle which encloses the particular figure being drawn. Quickdraw Trainer uses this value when it randomly generates rectangle coordinates.
Ok


Graphics Operations Pens Options Patterns Text Help
Use Rectangle Dimensions
When this item is checked, Quickdraw Trainer uses the values of the rectangle height and width shown. All randomly chosen rectangle coordinates will be such that the height and width of the resulting rectangle are equal to the desired values.
Ok


Graphics Operations Pens Options Patterns Text Help
Text
Using the Text menu, you can control the text displayed by Quickdraw Trainer. This includes the point size, the font number, and the characterics of the text face. For the text face, you can select either Plain or any combination of the remaining choices.
Ok

Appendix D

Feedback

If you have any suggestions, comments, or criticisms regarding the material in this book, I would welcome hearing from you. I would like to know what you liked and didn't like, what you found helpful, what excited you, what disappointed you. I would enjoy hearing stories about your own work with MS-BASIC. I won't promise to answer all queries, but if you have questions, feel free to send those along as well. Who knows, there may even be enough to warrant a second volume on interactive programming.

Send your feedback to me in care of TAB BOOKS Inc.:

Richard C. Vile, Jr.
c/o TAB BOOKS, Inc.
Blue Ridge Summit,
PA 17214

Index

A

active window, 29, 33, 43
Address Book Entry program, 58-60
analog controls, 98
AND operator, 176
angle, 98
anthem, programmers, 137, 138
arithmetic, modular, 111
arrays, 130

B

bar, menu, 16, 17
BASIC program, generating, 241
binary representation, 173, 174
binary version of MS BASIC, 250
bit images, 173, 174
bit-manipulation operations, 175
boxes, dialog, 219
bugs, 9, 156-161
BUTTON statement, 48, 130, 131, 211, 241
buttons, 3, 55

C

CALL FRAMERECT, 140
CALL MOVE TO statement, 242
CALL statement, 70, 159
calls, ROM, 70
CHAIN statement, 204
Changing Cursors program, 252-254
chinese rings puzzle, 112
click detector, 254
click, double, 148
clicks, mouse, 109, 116, 147
command keys, 22
command, BUTTON, 48
command, WINDOW, 152
commands used in programs, 257-261
controls, analog, 98
conversions of existing programs, 249, 250
coordinates, 139
coordinates, polar, 98
crosshairs, 63, 64, 68
Cursor Editor program, 164-170
cursors, 252

D

DATA statements, 84, 142, 215
decimal version of MS BASIC, 250
destructive reference, 157
DIALOG (0), 157, 161
DIALOG (1), 161, 202
dialog boxes, 219
Dialog Designer program, 220-234
DIALOG function, 33, 34, 60, 130, 135
dialog loop, 211
DIALOG queue, 156
DIALOG trap, 152
dialog trapping, 45
dialog, 5
DIALOGUE OFF statement, 50
DIALOGUE STOP statement, 50
dimension statements, 130
dimming items, 216
dimming menus, 146
Double Click program, 255
double click, 148
drag operation, 150
dragging operation, 178
dragging, 69
dwarfs, seven, 11

E

EDIT FIELD statements, 60, 202, 242
edit fields, 3
EDIT$ function, 60, 157, 158, 202
editor, cursor, 162
Elevator program, 46-48
Enter key, 60, 61
entertainment, 91, 180
ERASERECT. 248
error trapping, 138
error, overflow, 57
errors, program, 156-161
event handling, 14
event trapping, 135
events, 136
events, dialog, 33, 34

F

Fancy MOUSE (0) program, 119, 120
fat bits, 162
feedback, reader, 274
fields, edit, 3, 157
flashing windows, 29
FOR NEXT loop 242
FRAMERECT, CALL, 140
function, DIALOG, 33, 34

G

Generated BASIC for Print Setup Dialog Box program, 242
generating a BASIC program, 241
GET arrays, 172, 173
GET statement, 68, 247
GET/PUT array, 162
graphics viewport, 159
graphics, 5, 81, 139

H

handler, menu, 23
handling, event, 14
Help File Generator program, 125, 126
Help File Generator, 266
Help File Support program, 128, 129
help messages, 124
hexadecimal notation, 175

I

idle-loop, 15
images, bit, 173, 174
INKEY$, 97

interface, user, 1
items, menu, 22

K

kaleidoscopes, 180

L

label, 6
LINE command, 140
line numbers,6, 117
loop, exiting, 215
loop, idle-, 15

M

maintenance of programs, 9
mania, button, 55
maze, menu, 18
memory, size, 8, 159
MENU command, 1, 2, 14
MENU function, 1, 2
menu handler, 23
Menu maze program, 19-22
MENU OFF statement, 16, 144
MENU ON statement, 1, 2, 16, 158
MENU RESET command, 17
MENU STOP statement, 16, 144
menu toggles, 145
Menu Trainer program, 11-14
menu trainer, 11
menus, 81, 141-146, 194
menus, dimming, 146
menus, fat, 159
MERGE command, 128
mini-finder window, 99
modular arithmetic, 111
mouse clicks, 109, 116
MOUSE function, 67, 118, 247
Mouse Kaleidoscope program, 181-192
mouse techniques, 162, 177
MOUSE trap, 158
Mouse Utilities programs, 62-67
mouse, 5, 146-151

N

negation, logical, 68
NOT operator, 176
notebook, programmers, 132, 133
numbers, line, 6, 117
numeric values, 144

O

ON DIALOG GOSUB, 153
ON GOSUB statement, 15
ON MOUSE statement, 150
ON MENU statement, 1,2
ON TIMER GOSUB statement, 214
ON TIMER statement, 161, 214
One-Pixel Window Borders program, 139, 140
OR operator, 176
output window, 29, 33, 43
overflow error, 57

P

Pattern Maker program, 53-55
patterns, 198
pixels, 170
polar coordinates, 98
polling, 97, 137
Pretty Flashy program, 29-32
PRINT statements, 242
program, Address Book Entry, 58-60
program, Cursor Editor, 164-170
program, Dialog Designer, 220-234
program, Double Click, 255
program, Elevator, 46-48
program, Fancy Mouse (0), 119, 120
program, Generated BASIC for Print Setup Dialog Box, 242
program, Help File Generator, 125, 126
program, Help File Support, 128, 129
program, Menu Maze, 19-22
program, Menu Trainer, 11-14
program, Mouse Kaleidoscope, 181-192
program, One-Pixel Window Borders, 139, 140
program, Pattern Maker, 53-55
program, Pretty Flashy, 29-32
program, Prototype Menu Handler, 23-26
program, Quickdraw Trainer, 71-81
program, Rose Patterns, 99-103
program, Scribble and Video, 91-96
program, Ups 'N Downs, 113-116
program, Vocabulary Quiz, 205-211
program, Window Trainer, 35-42
programming topics, 262-265
programs, 7
programs, maintenance of, 9
programs, Mouse Utilities, 62-67
Prototype Menu Handler program, 23-26
PUT arrays, 172, 173
PUT statement, 68, 247
puzzle, chinese rings, 112

Q

queue, 33
queue, dialog, 156
Quickdraw graphics, 5
Quickdraw Routines, 49
Quickdraw Trainer program, 71-81
Quickdraw Trainer, help information, 266

R

radians, 98
READ statement, 84
readability of programs, 5, 6
reader feedback, 274
rectangle, dragging, 69
rectangles, 68, 140, 141
reference, destructive, 157
regions, 159
RESTORE command, 215
RESUME statement, 138
Return key, 60, 61
ROM calls, 70
ROM routines, 49, 83
Rose Patterns program, 99-103
roses, 98

S

Scribble and Video program, 91-96
size, memory, 8
STR$, 202
subroutines, 16
subroutines, arranging, 26-28
symmetry, 192
syntax error, forcing, 138

T

techniques, mouse, 147
TEXTSIZE routine, 159
theta, 98
TIMER ON statement, 214
toggles, menu, 145
topics, programming, 262-265
trainer, menu, 11
trap, dialog, 48, 49, 152
trap, MOUSE, 158
trapping events, 136
trapping, 14, 97, 135

U

update region, 159
Ups 'N Downs program, 113-116
user interface, 1
utilities, mouse, 62-67

V

VAL, 202
values, numeric, 144
viewport, graphics, 159
Vocabulary Quiz program, 205-211

W

WEND statement, 97, 215
WHILE loop, 130
WHILE, 97
WINDOW 1, 154
WINDOW command, 29, 152
Window Trainer program, 35-42
WINDOW, 158
window, active, 33
window, mini-finder, 99
window, output, 33
windows, 2, 43, 139, 152-155, 251, 252, 256
windows, multiple, 29

X

XOR operator, 176

Edited by Marilyn L. Johnson

Macintosh Programming Using MS-BASIC 2.0



If you are intrigued with the possibilities of the programs included in *Macintosh Programming Using MS-BASIC 2.0* (TAB Book No. 2621), you should definitely consider having the ready-to-run disk containing the software applications. This software is guaranteed free of manufacturer's defects. (If you have any problems, return the disk within 30 days, and we'll send you a new one.) Not only will you save the time and effort of typing the programs, the disk eliminates the possibility of errors that can prevent the programs from functioning. Interested?

Available on disk for Macintosh 128K or greater at $19.95 for each disk plus $1.00 shipping and handling. (Requires Microsoft BASIC 2.0, which may be purchased from your Apple dealer.)

I'm interested. Send me:

_____ disk for Macintosh 128K (6241)

_____ TAB BOOKS catalog

_____ Check/Money Order enclosed for $19.95 plus $1.00 shipping and handling for each disk ordered.

_____ VISA _____ MasterCard

Account No. _____ Expires _____

Name _____

Address _____

City _____ State _____ Zip _____

Signature _____

Mail To: **TAB BOOKS Inc.**
P.O. Box 40
Blue Ridge Summit, PA 17214

(Pa. add 6% sales tax. Orders outside U.S. must be prepaid with international money orders in U.S. dollars.)

TAB 2621

Other Bestsellers From TAB

☐ **MACINTOSH™ ASSEMBLY LANGUAGE PROGRAMMING**

Delve below the surface-level capabilities of your Macintosh and discover the incredible power that assembly language can unlock. You'll learn all about the fundamentals of machine code . . . gain an understanding of editors and assemblers . . . and tap into the 68000's addressing modes and instruction set. It's a book no Mac owner can afford to miss! 208 pp., 31 illus. 7″ × 10″.

Paper $16.95 **Hard $24.95**
Book No. 2611

☐ **DATA COMMUNICATIONS AND LOCAL AREA NETWORKING HANDBOOK**

With data communications and LANs being the area of greatest growth in computers, this sourcebook will help you understand what this emerging field is all about. Singled out for its depth and comprehensiveness, this clearly-written handbook will provide you with everything from data communications standards and protocols to the various ways to link together LANs. 240 pp., 209 illus. 7″ × 10″.

Hard $25.00 **Book No. 2603**

☐ **TRUE BASIC® PROGRAMS AND SUBROUTINES—Craig**

Explore the powerful, built-in features of True BASIC—a new language that is destined to standardize microcomputer programming. Now professional programmer and consultant John Clark Craig shows you hands-on how True BASIC can make your programming easier and less time-consuming than traditional languages. You'll discover the features that make True BASIC unmatched: coherent syntax, compiled operating speed, greatly improved graphics capabilities, structured language features, and portability. 224 pp., 50 illus. 7″ × 10″.

Paper $16.95 **Hard $24.95**
Book No. 1990

☐ **TRUE BASIC® A COMPLETE MANUAL—Simpson**

Have you heard about the new, improved version of BASIC that's taking the microcomputer industry by storm? Now, this groundbreaking guide makes it possible for you to understand and even start programming in True BASIC. Written by microcomputer programmer and consultant Henry Simpson, *True BASIC—A Complete Manual* covers all the main features of True BASIC including commands/statements/and functions, program control, input/output, file-handling, and even graphics. 208 pp., 53 illus. 7″ × 10″.

Paper $14.95 **Hard $22.95**
Book No. 1970

*Prices subject to change without notice.

☐ **THE COMPUTER SECURITY HANDBOOK—Baker**

Electronic breaking and entering into computers systems used by business, industry and personal computerists has reached epidemic proportions. That's why this up-to-date sourcebook is so important. It provides a realistic examination of today's computer security problems, shows you how to analyze your home and business security needs, and gives you guidance in planning your own computer security system. 288 pp., 61 illus. 7″ × 10″.

Hard $25.00 **Book No. 2608**

☐ **MACINTOSH™ EXPANSION GUIDE**

What kinds of accessories and add-ons are available for the Mac? Which ones are the best buys for the applications you have in mind? How can products from other manufacturers be interfaced with the Macintosh? You'll find the answers to these and just about any other question you have on Macintosh peripherals here, in this exceptionally thorough, time-saving guide. Far more than just a product listing or a rehash of manufacturers' sales brochures, these are the best of the peripherals currently on the market for the Mac . . . each one chosen for value and performance after exhaustive testing and examination. In addition, the authors provide you with evaluative reviews of the products and tables that list the vital statistics and features for quick comparative analysis. 224 pp., 31 illus. 7″ × 10″.

Paper $16.95 **Hard $22.95**
Book No. 2601

☐ **JAZZ!™—Bolocan**

Let software expert David Bolocan guide your masterfully through all of Jazz's capabilities—word processing, spreadsheet analysis, database management, communications, and business graphics capabilities. Written in easy-to-understand, plain-English, this hands-on tutorial takes you from an introduction to using Jazz on the Macintosh and fundamental commands to exploring its tremendous applications and intergrating them. 304 pp., 249 illus. 7″ × 10″.

Paper $17.95 **Hard $24.95**
Book No. 1978

☐ **MAKING MONEY WITH YOUR MICROCOMPUTER—2nd Edition**

Let your PC pay for itself by putting it to work in your own profitable part-time business. This newly revised, expanded, and updated idea book is overflowing with practical, proven business suggestions for getting started. Plus you'll find sources for software needed to get started. From setting up your office to locating the best market, all the factors that equal success are provided. 208 pp., 78 illus.

Paper $10.95 **Hard $16.95**
Book No. 1969

Look for these and other TAB BOOKS at your local bookstore.

TAB BOOKS Inc.
P.O. Box 40
Blue Ridge Summit, PA 17214

Send for FREE TAB Catalog describing over 900 current titles in print.

ISBN 0-8306-0321-2