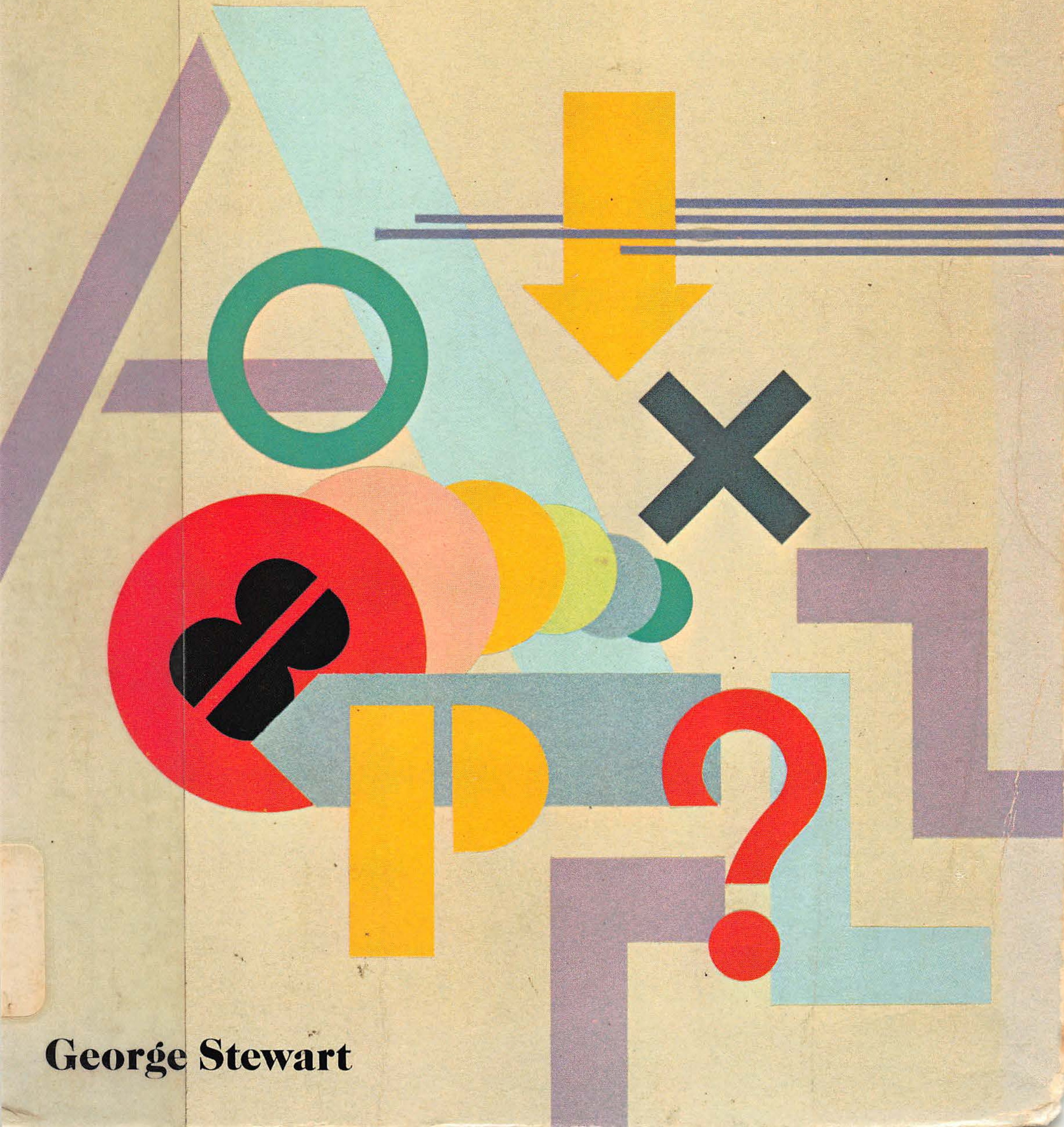
Osborne McGraw-Hill

Macintosh™ PROGRAM FACTORY™



George Stewart

The Macintosh™ Program Factory™

The Macintosh™ Program Factory™

George Stewart

Osborne McGraw-Hill
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

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

—The Macintosh™ Program Factory™

1234567890 DODO 898765

ISBN 0-07-881175-9

Karen Hanson, Acquisitions Editor
David Ushijima, Technical Editor
Ted Gartner, Copy Editor
Judy Wohlfrom, Text Design
Deborah Wilson, Composition
Yashi Okita, Cover Design

Acknowledgments

The author is grateful to the editors at Osborne/McGraw-Hill for their professionalism and kindness during the writing of this book. Thanks in particular to Karen Hanson, Jon Erickson, and Harry Wong.

To Francina, Nathanael, and Bethany

Table Of Contents

Introduction

Your Macintosh computer has a tremendous amount of power inside, power that you've probably seen harnessed to a specific application like word processing or picture drawing. The programming projects in this book will put you in control of that computing potential, setting your Macintosh to work as a puzzle generator, entertainer, teacher, creative assistant, and general helper.

Most of the programs in this book let you contribute something as well so that the program's operation or its results have your own personal touch. You'll be able to enjoy these programs for a long time to come, changing them every now and then to suit a special purpose or simply for variety.

If you're interested in how the programs work, you'll get an inside view from the commentary that accompanies the program listings. Many of the techniques and ideas can be adapted to your own programming projects.

The programs are written in Microsoft BASIC, the most widely used form of BASIC. All programs take advantage of the Macintosh graphics and mouse, and many can be used in conjunction with other Macintosh programs, such as MacWrite and MacPaint.

The step-by-step method of presentation and many of the programs in this book are adapted from my column, "The Program Factory," which appeared for a number of years in *Popular Computing* magazine.

Contents of the Book

The 20 programs in this book fall into five categories:

- *Puzzle-generators* produce graphic and word puzzles that may be printed on paper. The printed puzzles may then be used without the computer.
- *Games and simulations* for one or more persons; the computer plays an active role.

- *Education and self-improvement projects* that teach and exercise your mind.
- *Creativity and art projects* in which the computer becomes a way of extending your imagination.
- *Handy tools* or application programs for use around the home or office that has entered the computer age.

Chapter Organization

Each chapter starts off with a little background and introductory material about the subject at hand. A description of the main programming methods or techniques used in the program follows.

The program listing comes next. It is presented in functional blocks accompanied by explanatory comments. With some of the longer projects, test points are provided so that you can check your work as you go along.

A concluding section of each chapter gives hints and tips for using the program.

Computer Requirements

To run these programs you'll need a Macintosh computer equipped with Microsoft BASIC version 2.0 or later. *Be sure to use the binary version of BASIC, not the decimal or "business" version.* All of the programs will run in a 128K RAM Macintosh, but most of them run faster and with more capabilities in the 512K RAM machine. Your computer system should also include the Apple Imagewriter printer; other printers may not reproduce the graphics accurately.

Suggestions for Entering Programs

Before typing in any of these programs, find out how to enter and edit a program using Microsoft BASIC. Step-by-step instructions are given in Chapter 2 of the *Microsoft BASIC Interpreter* manual that is included with the Microsoft BASIC disk.

Type slowly and carefully when entering the program lines. Check your work as you go along. Before trying to run a program, save it on disk and get a printout on paper. Compare the printout line for line with the listing that appears in this book. A program is like a genetic code—one bit out of place and a useless mutation may result.

Be especially careful to distinguish the letter O from the numeral 0 and the letter l from the number 1. Whenever you see a pair of quotes in a listing, as in " ", count the number of empty spaces between the quotes and be sure to type in the same number on your computer. Sometimes there are no spaces at all inside the quotes. We call that a null string or nu$ in the listings, and it is important that such null strings be truly null (empty).

Some of the program lines are too wide to fit on a page of this book so they are continued on the next line with an indentation. When you come to the end of a line, check to see if the following line is indented. If it is, don't press RETURN until you have typed in the indented line as well.

Always type with your keyboard in the lowercase mode. As soon as you press RETURN at the end of the program line, Microsoft BASIC will capitalize and display in boldface those words it recognizes. The resultant capitalization should match the listings shown in this book; if it doesn't, you probably made a typing error.

Test points are provided for some of the longer projects. To test an incomplete program, you will often have to enter a few temporary lines, run the program, and then delete the temporary lines when the test is complete. When it is time to delete the lines, they are presented again, this time highlighted with gray shading. The text also provides explicit directions for deleting the test lines.

After making a line-for-line check of your program, try to run it. To determine whether your version is working or not, compare your results with the sample screen figures shown in the chapter.

Program Disks

All the programs in this book are available on a 3 1/2-inch disk. Price of the disk is $35. To order your disk, complete the form and mail it with your personal check or money order to: The Macintosh Program Factory™, Box 137, Hancock, NH 03449. Allow three weeks for delivery (add ten days if sending a personal check).

— — — — — — — — — — — cut along this line — — — — — — — — — — —

Please send me the Macintosh Program Factory on disk. My $35 payment is enclosed.
______ check ______ money order

Name ____

Address ______________________________________

City ______________________ State ________ ZIP ________

The Macintosh Program Factory™, P.O. Box 137, Hancock, New Hampshire 03449

Chapter 1

Making Mazes

If you enjoy the challenge of a good maze, consider the task of designing one. That turns out to be every bit as difficult, and quite a bit more interesting. In this chapter, we'll explore the process of maze construction and then program your Macintosh to produce an endless supply of mind-boggling mazes of varying complexity.

One way to start a maze is to picture the floorplan of a house with the walls in place but with no doors. You then add doors until there's just one path between any two rooms in the house. Last of all, you add an entrance and an exit anywhere you like.

Figure 1-1 shows a 4 × 4 maze. Verify for yourself that there is exactly one path between any two rooms. Try closing the entrance and exit and making new ones: you will still have a perfectly good maze.

—Constructing Mazes

During construction, a maze is divided into the following three types of rooms:

- Living quarters (LQ): rooms that are connected by doorways.

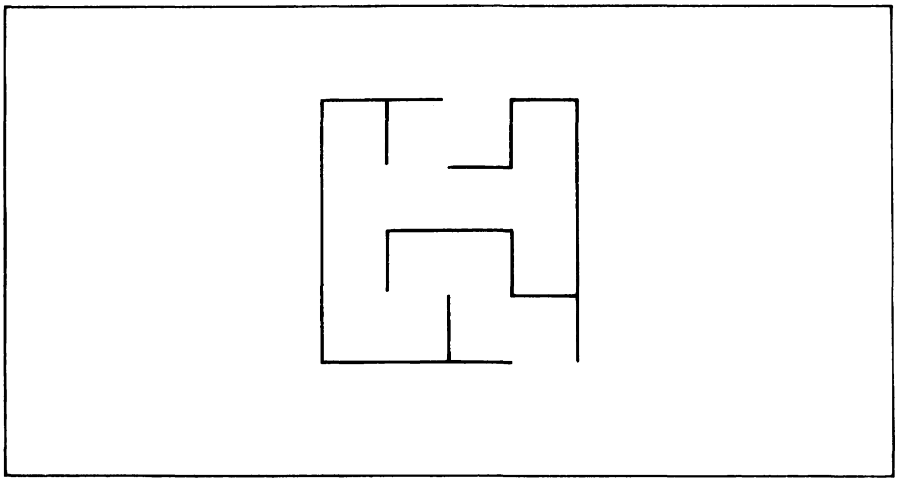


Figure 1-1. A simple maze

- Planned expansion (PE): rooms that are adjacent to the living quarters but don't have doors yet.
- Unused space (US): rooms that are not adjacent to the living quarters and have no doors.

The steps for building a maze are as follows:

1. Divide the maze into rooms and mark all rooms US.
2. Randomly select a room to be the LQ.
3. Locate all US rooms adjacent to the LQ and add them to the PE list.
4. If no PE rooms remain, go to step 8; otherwise, continue.
5. Randomly select a room from the PE list. Add a connecting door to the LQ (if more than one LQ room is adjacent, randomly select one).
6. Mark the new room as LQ; mark all PE rooms resulting from this addition.
7. Go back to step 3, using the new LQ room as the starting point.
8. Randomly select an entrance on the top and an exit on the bottom.

Verify that this procedure works by using it to create a 4 × 4 maze. Figure 1-2 shows the first four iterations of the process.

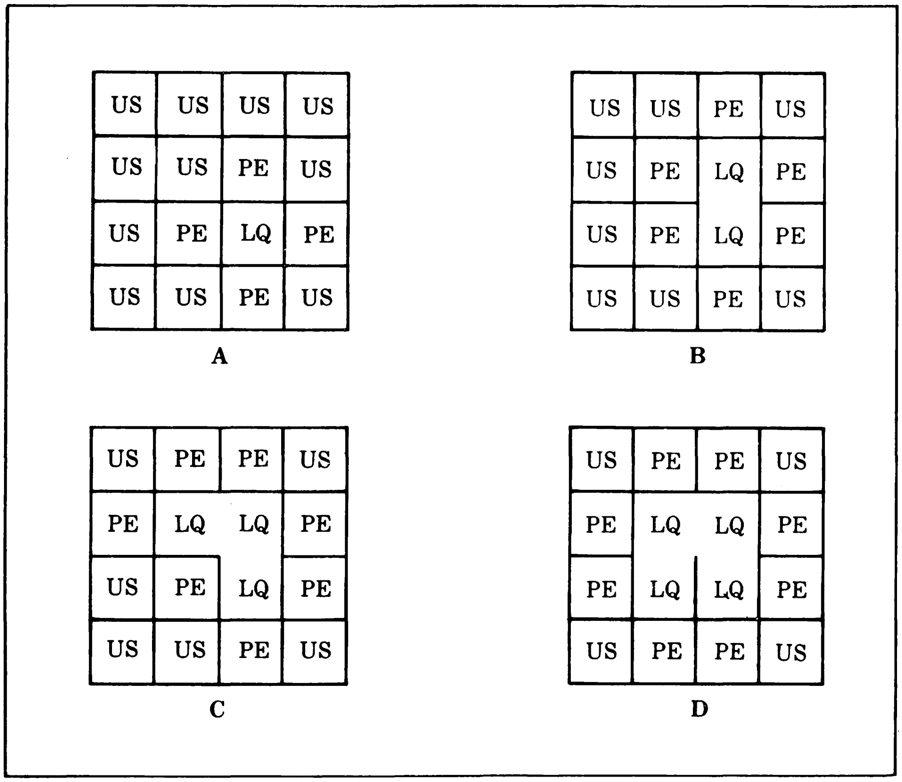


Figure 1-2. First four iterations of the maze construction process

—A Computerized Maze

The maze is stored inside the computer as a two-dimensional array called M(,). The room at row R, column C corresponds to the array element M(R,C). The number stored in each element indicates whether the room is LQ, PE, or US.

US rooms are represented by 0. PE rooms are represented by −1. LQ rooms are represented by a positive number from 1 through 15, with the exception of the first LQ room.

The number of an LQ room is calculated by assigning the numbers 1, 2, 4, and 8 to the east, south, west, and north walls, respectively. The numbers of all walls with doors are then added to produce a door code.

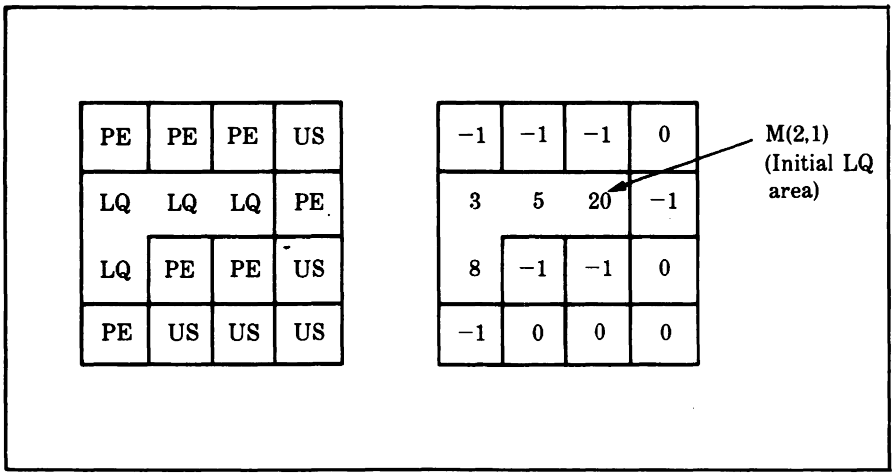


Figure 1-3. Maze under construction

Figure 1-3 shows a maze under construction using the LQ/PE/US coding system and again using the numerical coding system.

Note that the very first room of the living quarters is a special case because when it is first selected, it has no doors. This gives it a door code of 0, the same as unused space (US). To distinguish it from unused space, we add 16 to its initial door code.

—The Program

The maze program is best explained in logical blocks. Feel free to skim through the explanations and concentrate on entering the actual listings. You can always return to the explanation later on.

Defining the Maze Window

The first block allocates memory and defines the parameters for the maze window.

```
CLEAR ,35000!
LET cXr.lim%=1920
LET wnd.w%=72*5   : REM window specifications
LET wnd.l%=72*3.5
LET wnd.x%=72*.25
LET wnd.y%=72*.5
LET border%=6
LET image.w%=wnd.w%-border%*2
LET image.l%=wnd.l%-border%*2
LET max.columns%=(image.w%-1)\2
LET max.rows%=(image.l%-1)\2
```

Users of 128K Macintoshes must change the first program line; see the important note at the end of this chapter.

The size of the maze is limited by the screen dimensions and by BASIC's string length, 32767 (since the maze drawing commands are stored as a string). The product of the maze length times its width (counted in cells) can be no greater than cXr.lim%. Assuming that this criterion is met, the maze must also be small enough to fit within the 5 × 3.5-inch maze window.

Wnd.w% and wnd.l% are the length and width of the maze window. Wnd.x% and wnd.y% are the coordinates of the upper-left corner of the window.

Construction Constants

The next block defines certain constants for the maze construction procedure.

```
LET empty.cell%=0
LET first.cell%=16
LET pe.cell%=-1
LET boundary.cell%=-2
LET image.xy%=border%
DIM exp2%(3),dc%(4),dr%(4),gray%(3),black%(3),white%(3)
FOR d%=1 TO 4
READ dc%(d%),dr%(d%)
LET exp2%(d%-1)=2^(d%-1)
NEXT d%
DATA 1,0,0,1,-1,0,0,-1
READ gray.code%,black.code%,white.code%
DATA -21931,-1,0
FOR code%=0 TO 3
LET gray%(code%)=gray.code%
LET black%(code%)=black.code%
LET white%(code%)=white.code%
NEXT code%
```

Array exp2%( ) holds the values 2^n for n=0 to 3. Since these values are used repeatedly, it is faster to recall them from an array than to recalculate them over and over. Arrays dr%( ) and dc%( ) hold row and column increments corresponding to the directions east, south, west, and north. Arrays gray%( ), black%( ), and white%( ) hold the codes for the three corresponding color patterns.

Implementing the Specification Dialog Box

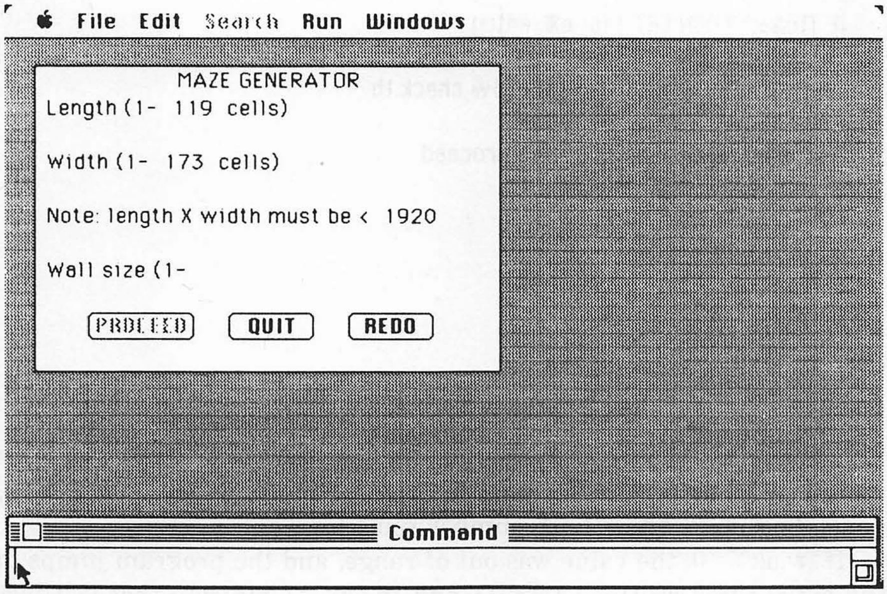
The next block sets up a dialog box so the program operator can specify the maze dimensions and wall thickness.

```
spec.maze:
WINDOW 1,,(18,36)-(288,214),3
PRINT TAB(11);"MAZE GENERATOR"
PRINT " Length (1-";max.rows%;"cells)"
PRINT
PRINT " Width (1-";max.columns%;"cells)"
PRINT
PRINT " Note: length X width must be <";cXr.lim%
PRINT
PRINT " Wall size (1-";
BUTTON 1,0,"PROCEED",(30,144)-(92,160)
BUTTON 2,1,"QUIT",(112,144)-(162,160)
BUTTON 3,1,"REDO",(182,144)-(232,160)
```

After typing in this block, you can run the first part of the program. (Close the listing window and type COMMAND-R.) Your screen should display the dialog box shown in Figure 1-4. Notice that the PROCEED button is inactive.

Now reopen the listing window (COMMAND-L) and continue typing in the program. The next block starts the dialog:

```
begin.dialog:
LET last.c%=15
LET last.r%=10
LET th%=10   : REM wall thickness
LET fld%=1   :REM active field
LET nxt.fld%=1
GOSUB check.lw
EDIT FIELD 3,STR$(th%),(198,112)-(240,127)
EDIT FIELD 2,STR$(last.c%),(198,48)-(240,63)
EDIT FIELD 1,STR$(last.r%),(198,16)-(240,31)
```



MAZE GENERATOR
Length (1- 119 cells)

Width (1- 173 cells)

Note: length X width must be < 1920

Wall size (1-

PROCEED QUIT REDO

Figure 1-4. The preliminary maze specification dialog box

Last.c%, last.r%, and th% are the initial settings for the maze width, length, and wall thickness. Each time you run the program, these values will appear as preset values in the dialog box.

The next block waits until you click the mouse in a field or button, or until you press RETURN or ENTER:

```
get.size:
LET act%=DIALOG(0)
IF act%=1 THEN ON DIALOG(1) GOTO check.fld,quit,begin.dialog
IF act%=2 THEN LET nxt.fld%=DIALOG(2): GOTO check.fld
IF act%=6 THEN LET nxt.fld%=(fld% MOD 3)+1: GOTO check.fld
GOTO get.size
```

Whenever you enter a field by clicking the mouse or pressing ENTER or RETURN, the following block checks the contents of that field:

```
check.fld:
LET entry=VAL(EDIT$(fld%))
IF entry<>INT(entry) THEN entry.error
IF entry<-32768 OR entry>32767 THEN entry.error
IF fld%=1 THEN LET last.r%=entry
IF fld%=2 THEN LET last.c%=entry
IF fld%=3 THEN LET th%=entry
ON fld% GOSUB check.lw,check.lw,check.th
IF lw.ok%=0 THEN entry.error
IF act%=1 AND th.ok%=1 THEN proceed
LET fld%=nxt.fld%
EDIT FIELD fld%
GOTO get.size
entry.error:
BEEP
EDIT FIELD fld%
GOTO get.size
```

This block first ensures that the number you've entered can be stored as an integer, then proceeds to a more specific range-checking subroutine, depending upon which field number you enter.

If lw.ok%=0, the value was out of range, and the program jumps to the entry-error routine. Act%=1 and th.ok%=1 indicate that you just pressed the PROCEED button and that the thickness-setting is within range. In this case, the program exits from the dialog loop and begins drawing the maze. Otherwise, the program automatically selects the next field for editing.

The next block ends the program if you press the QUIT button.

```
quit:
WINDOW CLOSE 1
END
```

The following lines comprise a subroutine to ensure that the maze specifications are within range:

```
check.lw:
LET lw.ok%=1    :REM length & width flag
LET th.ok%=1    :REM thickness flag
IF last.r%<1 OR last.r%>max.rows% THEN lw.ok%=0
IF last.c%<1 OR last.c%>max.columns% THEN lw.ok%=0
IF last.c%*last.r%>cXr.lim% THEN lw.ok%=0
LET v%=image.w%\(last.c%*2+1)
LET h%=image.l%\(last.r%*2+1)
LET limit%=-(v%<=h%)*v%-(h%<v%)*h%
IF th%<1 OR th%>limit% THEN th.ok%=0
BUTTON 1,lw.ok%*th.ok%
```

```
LOCATE 8,12
PRINT USING "** dots)";limit%;
RETURN
check.th:
LET lw.ok%=1
IF th%<1 OR th%>limit% THEN lw.ok%=0
BUTTON 1,lw.ok%
RETURN
```

Each time the length or width is changed, the program must alter the displayed limit for wall-size. This limit, limit%, ensures that the maze will not exceed the window size in either dimension (width or length).

Test Point

This is a good place to stop to check your work. But first, add this line at the end of the program listing (borrowed from the next block to be presented).

```
proceed:
```

Now close the listing window and run the program. You should see the screen presented in Figure 1-5. Try all of the options available in the dialog box.

Now stop the program (COMMAND-.), and open the listing window. Continue adding to the listing beginning at the end of the last line entered, "proceed:".

Creating the Maze

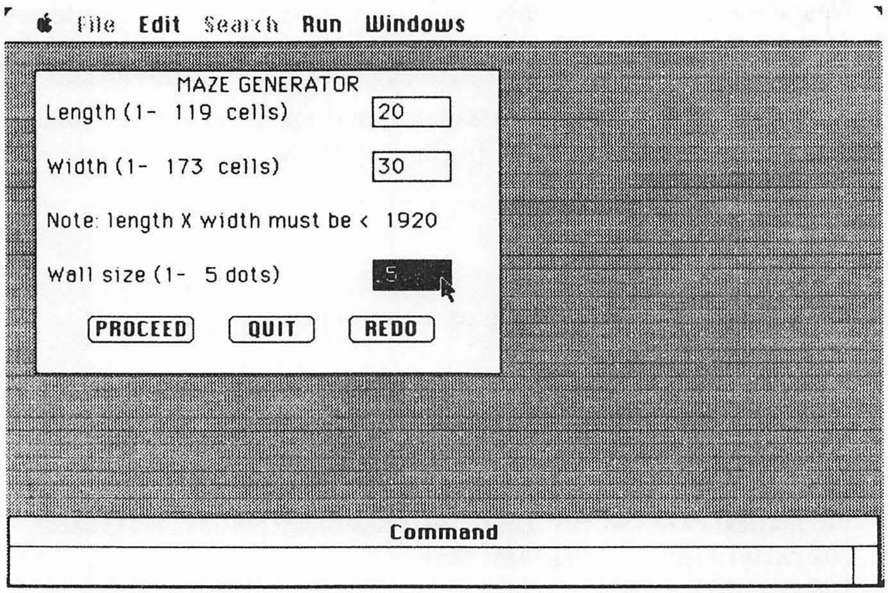
The next block sets up the maze as a two-dimensional array.

```
proceed:
WINDOW CLOSE 1
LET cell.size%=th%*2
LET last.pe%=2/3*last.r%*last.c%
DIM m%(last.r%+1,last.c%+1),pe.row%(last.pe%),pe.col%(last.pe%),vu%(4)
FOR r%=0 TO last.r%+1 STEP last.r%+1
FOR c%=0 TO last.c%+1
LET m%(r%,c%)=boundary.cell%
NEXT c%
NEXT r%
FOR c%=0 TO last.c%+1 STEP last.c%+1
FOR r%=0 TO last.r%+1
LET m%(r%,c%)=boundary.cell%
NEXT r%
NEXT c%
```

The variable last.pe% is the largest number of planned expansion (PE) cells possible for a given maze size. Array m%(,) stores the door codes for each room of the maze. Pe.row%( ) and pe.col%( ) store the row and column location of each PE cell. Vu%( ) stores the view in all four directions from the newest living quarters (LQ) cell.

Initially, the maze array m%(,) contains all 0's, except for the maze boundary cells along the top, right, bottom, and left edges of the maze. These cells get the value −2.

The next block opens window 1 for the maze and window 2 for a dialog box.



MAZE GENERATOR
Length (1- 119 cells) 20
Width (1- 173 cells) 30
Note: length X width must be < 1920
Wall size (1- 5 dots) 5

Figure 1-5. The complete maze specification dialog box

```
WINDOW 1,,(wnd.x%,wnd.y%)-(wnd.x%+wnd.w%,wnd.y%+wnd.l%),4
WINDOW 2,,(5.5*72,2.5*72)-(7*72,4*72),3
BUTTON 1,1,"CANCEL",(23,46)-(85,64)
WINDOW OUTPUT 1
PICTURE ON
CALL SHOWPEN
CALL PENSIZE(th%,th%)
IF th%>5 THEN CALL PENPAT(VARPTR(gray%(0)))
IF th%<5 THEN CALL PENPAT(VARPTR(black%(0)))
```

This dialog box contains a single CANCEL button that erases the maze window and restarts the maze specification dialog box. For wall thicknesses of six or greater, a gray pattern is used; for thinner walls a solid black pattern is used.

The following lines draw the floorplan of the maze with walls but no doors:

```
FOR r%=1 TO last.r%+1
CALL MOVETO(image.xy%,image.xy%+(r%-1)*cell.size%)
CALL LINETO(image.xy%+last.c%*cell.size%,image.xy%+(r%-1)*cell.size%)
IF DIALOG(0)=1 THEN stop.it:
NEXT r%
FOR c%=1 TO last.c%+1
CALL MOVETO(image.xy%+(c%-1)*cell.size%,image.xy%)
CALL LINETO(image.xy%+(c%-1)*cell.size%,image.xy%+last.r%*cell.size%)
IF DIALOG(0)=1 THEN stop.it
NEXT c%
```

The program draws all the horizontal walls first; then all the vertical walls.

Now the program randomly selects a room to be the first LQ cell.

```
CALL PENPAT(VARPTR(white%(0)))
RANDOMIZE TIMER
LET r1%=INT(RND*last.r%)+1
LET c1%=INT(RND*last.c%)+1
LET m%(r1%,c1%)=first.cell%
LET r%=r1%
LET c%=c1%
LET n%=0
GOSUB get.view
GOSUB mark.pe
```

Each time a room is added to the LQ space, the program calls the get.view subroutine to look for adjacent unused space (US) cells. Any US cells discovered are added to the list of PE cells by the mark.pe subroutine.

The following lines comprise a loop (repeated section of code) that continues adding rooms to the LQ space until no more PE cells are left:

```
WHILE n%>0
IF DIALOG(0)=1 THEN stop.it
LET pe.ptr%=INT(RND*n%)+1
LET r%=pe.row%(pe.ptr%)
LET c%=pe.col%(pe.ptr%)
GOSUB get.view
select.wall:
LET wd%=INT(RND*4)+1
IF vu%(wd%)<=0 THEN select.wall
LET pe.row%(pe.ptr%)=pe.row%(n%)
LET pe.col%(pe.ptr%)=pe.col%(n%)
LET n%=n%-1
LET m%(r%,c%)=exp2%(wd%-1)
LET opp.r%=r%+dr%(wd%)
LET opp.c%=c%+dc%(wd%)
LET m%(opp.r%,opp.c%)=m%(opp.r%,opp.c%) OR exp2%((wd%+1) MOD 4)
GOSUB erase.wall
GOSUB get.view
GOSUB mark.pe
WEND
```

The variable pe.ptr% randomly selects a cell from the PE list. The program gets the view from that cell; by definition, at least one of the PE cell's walls must be adjacent to the LQ space. The program randomly selects walls until it finds one that does lead to the LQ area. That wall is opened, and the PE cell is added to the LQ space.

The process repeats until the PE list is empty (n%=0).

Next the program randomly selects an entrance on the top and an exit on the bottom of the maze:

```
LET r%=1
LET c%=INT(RND*last.c%)+1
LET wd%=4
LET m%(r%,c%)=m%(r%,c%) OR exp2%(wd%-1)
GOSUB erase.wall
```

```
LET r%=last.r%
LET c%=INT(RND*last.c%)+1
LET wd%=2
LET m%(r%,c%)=m%(r%,c%) OR exp2%(wd%-1)
GOSUB erase.wall
```

When the maze is complete (or if you press CANCEL during maze construction), the following lines give you an opportunity to copy the maze to the Clipboard, ending the program, or to start a new maze:

```
stop.it:
ERASE m%,pe.row%,pe.col%,vu%
PICTURE OFF
LET maze$=PICTURE$
WINDOW OUTPUT 2
BUTTON CLOSE 1
CLS
PRINT "Copy picture"
PRINT "to clipboard?"
BUTTON 1,1,"YES",(30,40)-(80,62)
BUTTON 2,1,"NO",(30,74)-(80,96)
WHILE DIALOG(0)<>1
WEND
ON DIALOG(1) GOTO copy.maze,continue
```

The following lines copy the maze to the Clipboard:

```
copy.maze:
BUTTON CLOSE 1
BUTTON CLOSE 2
CLS
PRINT "One moment..."
OPEN "clip:picture" FOR OUTPUT AS 1
PRINT #1,maze$
CLOSE 1
CLS
PRINT "The clipboard"
PRINT "holds a copy"
PRINT "of the maze."
BUTTON 2,1,"OK",(30,74)-(80,96)
WHILE DIALOG(0)<>1
WEND
END
```

When you press the OK button, the program ends. To make a permanent copy of the maze, you can paste it into the Scrapbook or into a MacPaint document.

The following lines let you start a new maze:

```
continue:
WINDOW CLOSE 1
WINDOW CLOSE 2
GOTO spec.maze
```

Finally, here are three subroutines used during maze construction. The first gets the view from a selected cell:

```
get.view:
FOR d%=1 TO 4
LET vu%(d%)=m%(r%+dr%(d%),c%+dc%(d%))
NEXT d%
RETURN
```

On entry to this subroutine, r% and c% are the row and column addresses of the selected cell. On returning from the subroutine, array elements VU(1)-VU(4) hold the door codes of the four adjacent cells (east, south, west, north).

The second subroutine adds a cell to the PE list:

```
mark.pe:
FOR d%=1 TO 4
IF vu%(d%)<>empty.cell% THEN skip
LET n%=n%+1
LET pe.row%(n%)=r%+dr%(d%)
LET pe.col%(n%)=c%+dc%(d%)
LET m%(r%+dr%(d%),c%+dc%(d%))=pe.cell%
skip:
NEXT d%
RETURN
```

On entry, vu%( ) contains the four views from the selected cell. The program randomly selects views until it finds a US cell. It adds this cell to the PE list and puts the PE cell-code into the corresponding array.

The last subroutine erases a wall from the maze, creating a door between two rooms:

```
erase.wall:
LET x1%=image.xy%+(c%-1)*cell.size%
LET y1%=image.xy%+(r%-1)*cell.size%
ON wd% GOTO wall.1,wall.2,wall.3,wall.4
wall.1:
CALL MOVETO(x1%+cell.size%,y1%+th%)
CALL LINETO(x1%+cell.size%,y1%+cell.size%-th%)
GOTO erase.done
wall.2:
CALL MOVETO(x1%+th%,y1%+cell.size%)
CALL LINETO(x1%+cell.size%-th%,y1%+cell.size%)
GOTO erase.done
wall.3:
CALL MOVETO(x1%,y1%+th%)
CALL LINETO(x1%,y1%+cell.size%-th%)
GOTO erase.done
wall.4:
CALL MOVETO(x1%+th%,y1%)
CALL LINETO(x1%+cell.size%-th%,y1%)
erase.done:
RETURN
```

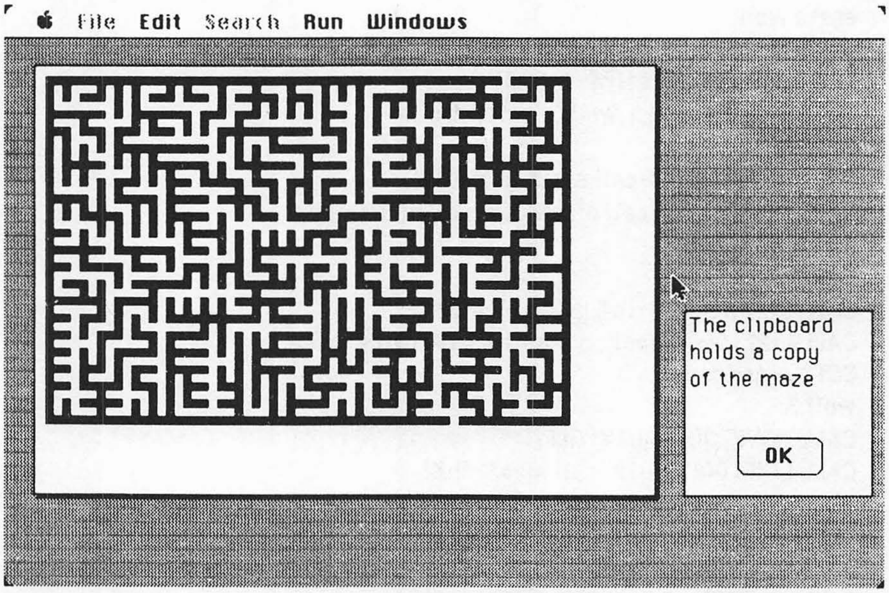
—Using the Program

Run the complete program. Specify a maze length of 6, width of 10, and thickness of 15. Then press the PROCEED button. The program should draw a maze of the specified size in a large rectangular window, while a CANCEL button will appear in a smaller window to the right, as you can see in Figure 1-6.

When the maze is complete, the program will ask whether you want to save the maze on the Clipboard. If you do, press the YES button. The program will confirm that the maze has been saved and will end, making BASIC's command window active.

To make a permanent copy of the maze, open the Scrapbook and paste the contents of the Clipboard onto a blank page (COMMAND-V). Another alternative is to open a MacPaint document and paste the Clipboard contents.

If you choose not to save a maze on the Clipboard, the program returns you to the maze specification window. Start another maze or press the QUIT button to end.



The clipboard holds a copy of the maze

Figure 1-6. A 20 × 30 maze

Important Note: If you have a 128K Macintosh, you must change the first line of the program as follows:

```
LET A=FRE(-1): CLEAR A/2,A/2
```

You will also need to restrict the size of the maze to 11 × 11.

This chapter has been adapted from "Making Mazes" by George Stewart, appearing in the November 1982 issue of *Popular Computing* magazine. Copyright 1982 Byte Publications, Inc. Used with the permission of Byte Publications, Inc.

Chapter 2

Hidden Words—Part 1

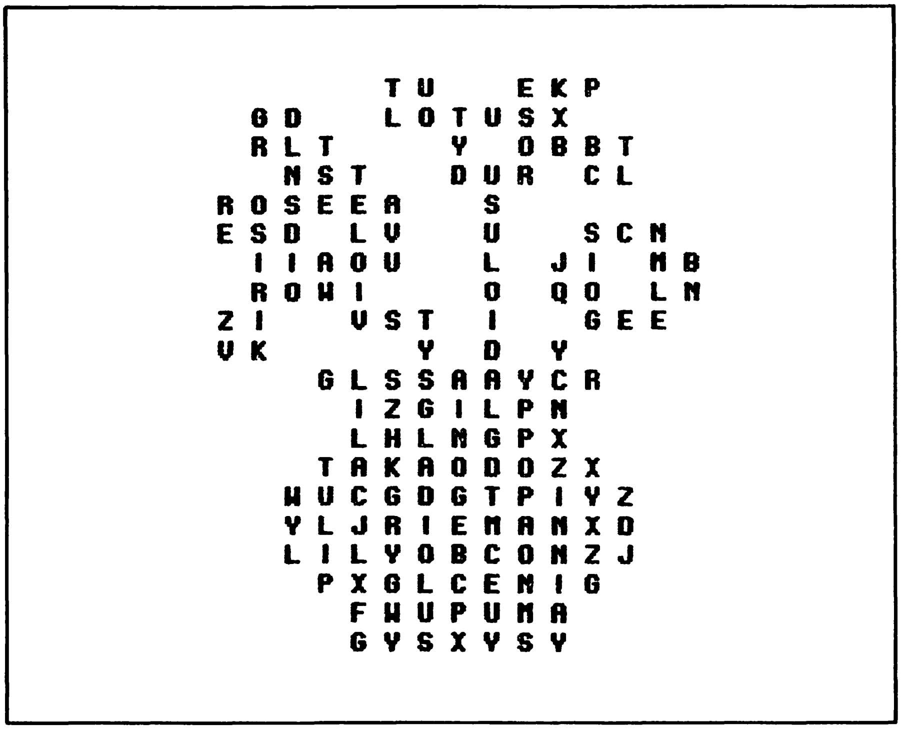
This program generates hidden word puzzles as challenging and entertaining as the best you'll find in newspapers or game magazines. The completed puzzles are truly personalized: you design the puzzle shape, specify the puzzle vocabulary, and determine the directions in which words may be placed.

Figure 2-1 shows a sample puzzle created with this program. The puzzle solution is given in Figure 2-2.

Because the program is long, it is presented in two parts in this chapter and the next. Several test points are provided during the presentation so you can check your progress. However, you won't be able to produce finished puzzles until you've entered the entire program.

—Overview

The program starts by asking you to provide a word list and to specify the puzzle's dimensions (the grid size). Then you are given the opportunity to define a shape inside the grid. The completed grid consists of



```
          T U     E K P
  G D     L O T U S X
  R L T       Y   O B B T
    N S T     D U R   C L
R O S E E A     S
E S D   L V     U     S C N
  I I A O U     L   J I   N B
  R O W I       O   Q O   L N
Z I     V S T   I     G E E
V K         Y   D   Y
      G L S S A A Y C R
        I Z G I L P N
        L H L N G P X
      T A K A O D O Z X
    W U C G D G T P I Y Z
    Y L J R I E N A N X D
    L I L Y O B C O N Z J
      P X G L C E N I G
        F W U P U N A
        G Y S X Y S Y
```

Figure 2-1. Seventeen flower names are hidden in this puzzle

blank cells (the white space in a finished puzzle) and empty cells to be filled with letters as words are placed into the puzzle.

Given these specifications, the program makes a puzzle. The program begins by shuffling a list of all the grid cells and selecting a letter cell from the shuffled list. The program chooses the longest available word first and tries to fit it in one of eight possible directions (east, southeast, south, southwest, west, northwest, north, and northeast).

If the program is unable to make the word fit, it tries the next-longest word in the list. If none of the words fit, the program skips to the next cell and tries to fit the longest available word.

After trying all the cells once, the program makes another pass through the list to find places where two words can start at the same cell. When it has completed the second pass, the program randomly fills in the remaining letter cells. The program then prints the puzzle on the display or printer.

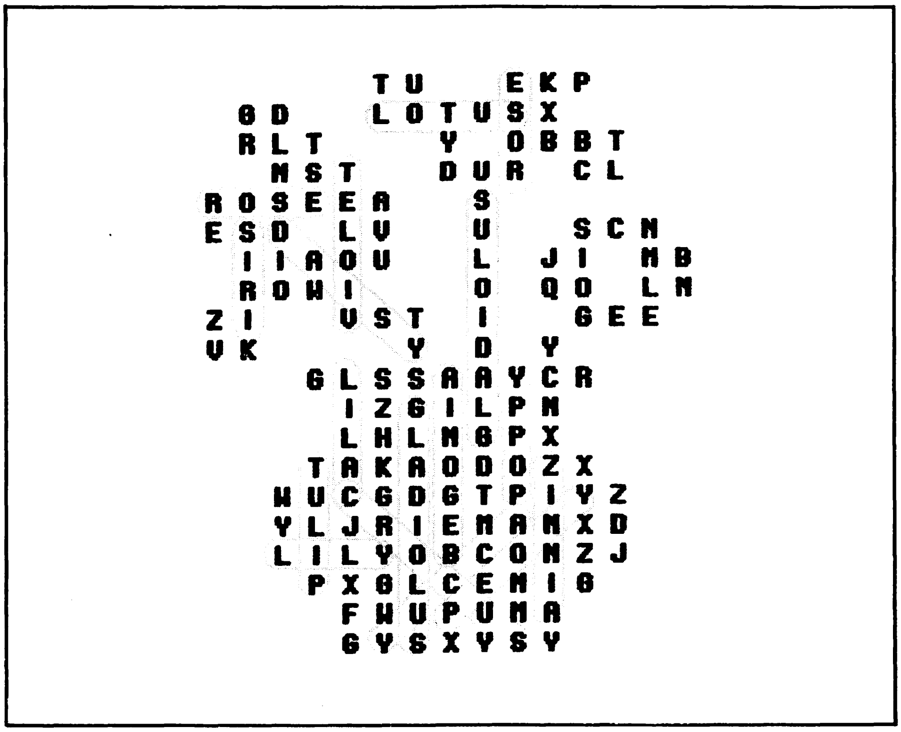


Figure 2-2. Solution to the flower puzzle

After the first puzzle has been completed, you can select any of the following commands in the Puzzle menu on the menu bar (see Figure 2-3):

Change grid shape—edit the puzzle shape

Change grid size—resize the puzzle grid

Change word list—change or replace the word list

Make puzzle—place words into the puzzle grid

Print puzzle—show the completed puzzle

Print solution—show the hidden-word locations

Save grid—save the completed puzzle in a disk file

Save word list—save the word list in a disk file

Quit

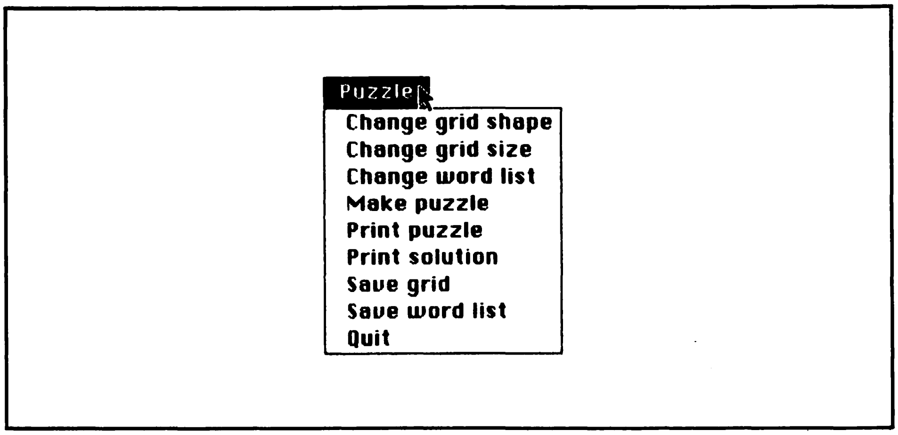


Figure 2-3. The main puzzle generation menu

The menu commands make it possible to change individual puzzle parameters without having to change all of them. For instance, after creating a puzzle, you may wish to modify its shape; you can do this without affecting the word list.

—The Program

The first block of the program loads the labels for the Puzzle menu into the array menu.label$.

```
READ last.option%
DIM menu.label$(last.option%)
FOR j%=0 TO last.option%
READ menu.label$(j%)
NEXT j%
DATA 9,Puzzle
DATA Change grid shape,Change grid size,Change word list
DATA Make puzzle,Print puzzle,Print solution
DATA Save grid,Save word list,Quit
```

The next lines load the data for the X-cursor, which appears while you are editing a shape:

```
DIM cursor%(33)
FOR j%=0 TO 33
READ cursor%(j%)
```

```
NEXT j%
DATA 0,0,0
DATA &H0808,&H0410,&H0220,&H0140,&H0080
DATA &H0140,&H0220,&H0410,&H0808
DATA 0,0,0,0
DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
DATA 8,9
```

The first 16 numbers in the DATA statements define a 16 × 16 cursor pattern, the next 16 numbers define a 16 × 16 cursor mask, and the last two define the *hot spot*, or origin, of the cursor. (Cursor definition is explained in detail in the Microsoft BASIC Interpreter reference manual, page 298.) Figure 2-4 shows the worksheet used to derive the cursor-pattern numbers.

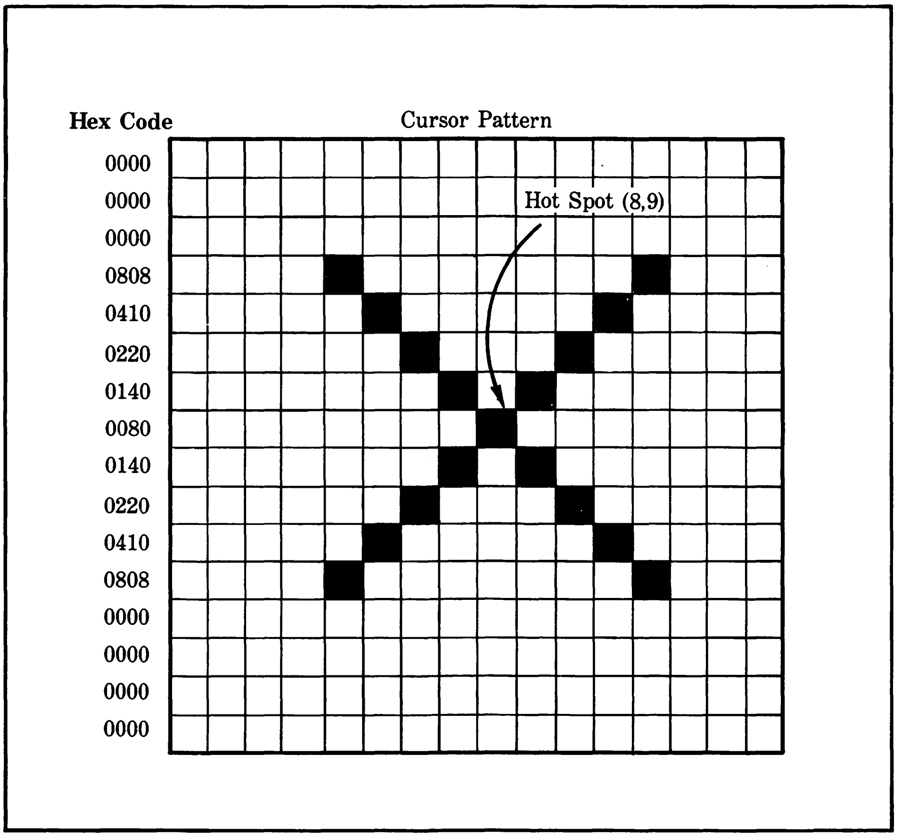


Figure 2-4. Definition worksheet for the X-cursor

The program creates two windows—one for use in dialogs and the other for output. The following lines set up the parameters that control window location and size:

```
LET w1.x%=.1*72     :REM window #1 left side
LET w1.y%=.35*72    :REM top
LET w1.w%=2.5*72   :REM width
LET w1.l%=3.5*72   :REM length
LET w1.x1%=w1.w%+w1.x%   :REM right side
LET w1.y1%=w1.l%+w1.y%   :REM bottom
LET w2.x%=2.7*72    :REM window #2 left side
LET w2.y%=.35*72   :REM and so forth...
LET w2.w%=4.3*72
LET w2.l%=4.3*72
LET w2.x2%=w2.w%+w2.x%
LET w2.y2%=w2.l%+w2.y%
LET border%=6
LET m2.w%=w2.w%-border%*2
LET m2.l%=w2.l%-border%*2
LET l.side%=3
LET l.space%=4
LET l.tot%=l.side%+l.space%
LET max.c%=(m2.w%+l.side%)\l.tot%
LET max.r%=(m2.l%+l.side%)\l.tot%
```

Refer to Figure 2-5 for an illustration of the variables used to create the two windows. The number 72 occurs frequently because it represents the number of pixels (points on the display) per inch. For example, 2.5*72 (2.5 times 72) represents the number of pixels in 2.5 inches.

The next block of lines initializes constant values:

```
LET max.wds%=100: REM arbitrary upper limit
LET nu$="": REM no spaces inside quotes
LET hole$=" ": REM one space inside quotes
LET ltr.cell$="*"
LET no.more$="/"
LET not.used%=-1
LET yes%=-1
LET no%=0
LET zone%=2
READ max.dir%
DIM dev$(3),ri%(max.dir%),ci%(max.dir%)
LET dev$(1)="SCRN:"
LET dev$(2)="LPT1:DIRECT"
LET dev$(3)="CLIP:TEXT"
FOR j%=1 TO max.dir%
READ ri%(j%),ci%(j%)
NEXT j%
DATA 8
DATA 0,1, 1,1, 1,0, 1,-1, 0,-1, -1,-1, -1,0, -1,1
```

The variable max.dir% contains the number of path directions that will be used in hiding words. Arrays ri%( ) and ci%( ) contain the row and column increments that produce each path direction. For instance

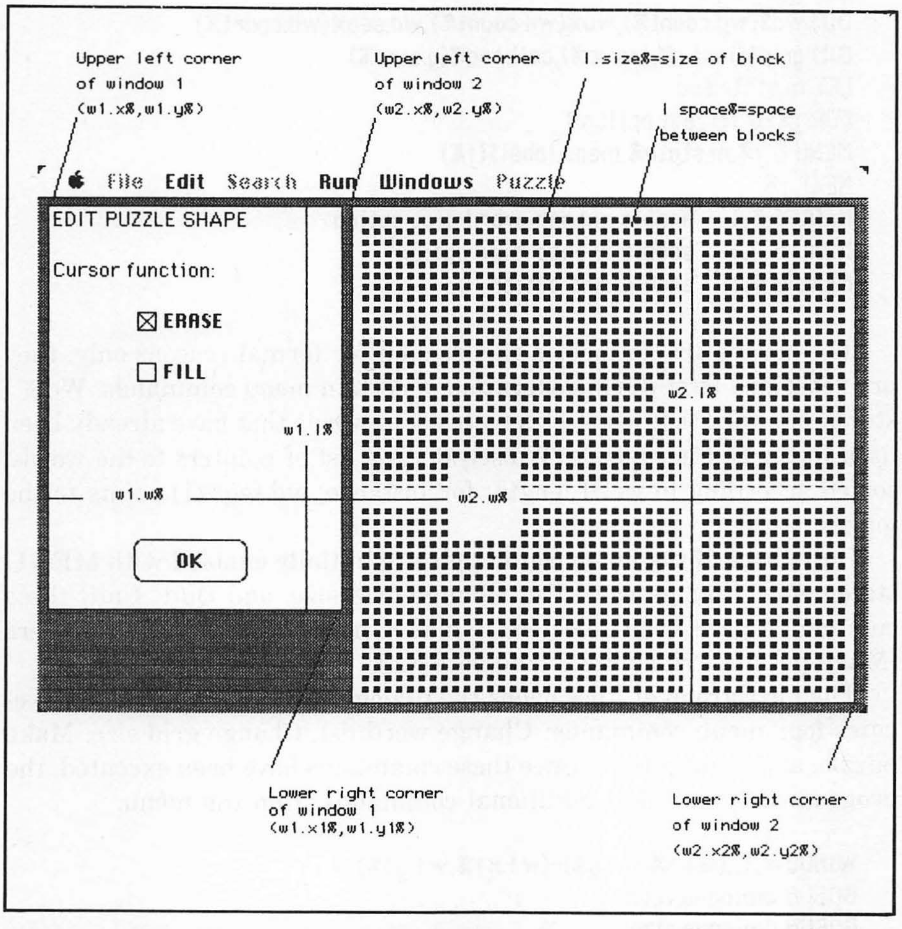

Upper left corner of window 1 (w1.x%,w1.y%)
Upper left corner of window 2 (w2.x%,w2.y%)
l.size%=size of block
l space%=space between blocks
File Edit Search Run Windows Puzzle
EDIT PUZZLE SHAPE
Cursor function:
☒ ERASE
☐ FILL
OK
w1.l%
w1.w%
w2.l%
w2.w%
Lower right corner of window 1 (w1.x1%,w1.y1%)
Lower right corner of window 2 (w2.x2%,w2.y2%)

Figure 2-5. Explanation of the window-parameter variables

direction 1 is specified as ri%(1)=0 and ci%(1)=1, indicating that the row position is unchanged while the column position is incremented by 1. This produces an easterly movement. The eight possible directions are given as eight pairs in the last DATA statement.

To simplify the puzzles, reduce the number of directions to four (east, southeast, south, and northeast) by changing the two DATA statements to DATA 4 and DATA 0,1,1,1,1,0,−1,1.

The next lines initialize the menu and certain control variables:

```
RANDOMIZE TIMER
LET wd.count%=0
LET last.r%=0
LET last.c%=0
LET g.size%=0
DIM wd$(wd.count%),wu%(wd.count%),wd.seq%(wd.count%)
DIM grid$(last.r%,last.c%),cell.seq%(g.size%)
LET m.state%=0
FOR j%=0 TO last.option%
MENU 6,j%,m.state%,menu.label$(j%)
NEXT j%
MENU 6,3,1   'enable change-word-list option
MENU 6,2,1   'enable change-size option
MENU 6,9,1   'enable quit option
```

The arrays are defined in this section for formal reasons only; they are redefined later during execution of certain menu commands. Wd$( ) stores the word list; wu%( ) keeps track of words that have already been used in the current puzzle. Wd.seq%( ) is a list of pointers to the words, sorted according to word length; for instance, wd.seq%(1) points to the longest word.

Only three of the menu commands are initially enabled with MENU statements: Change word list, Change grid size, and Quit. Until these commands have been selected, it makes no sense to select the others (Make puzzle, Print puzzle, and so forth).

The next group of lines opens the dialog box and automatically executes four menu commands: Change word list, Change grid size, Make puzzle, and Print puzzle. Once these commands have been executed, the program lets you select additional commands from the menu.

```
WINDOW 1,,(w1.x%,w1.y%)-(w1.x1%,w1.y1%),3
GOSUB dialogue.vocab
GOSUB dialogue.size
```

```
MENU 6,4,1   'enable make-puzzle option
GOSUB make.pzl
IF c.flag%=no% THEN  GOSUB prt.pzl
get.selection:
MENU 6,0,1
WHILE MENU(0)<>6
WEND
MENU 6,0,0
LET selection%=MENU(1)
IF selection%=0 THEN get.selection
WINDOW CLOSE 2
IF selection%=last.option% THEN END
IF selection%>3 THEN skip.gs
ON selection% GOSUB dialogue.shape,dialogue.size,dialogue.vocab
GOTO get.selection
skip.gs:
ON selection%-3 GOSUB make.pzl,prt.pzl,prt.sol,save.t,save.vocab
GOTO get.selection
```

Test Point 1

This gets you to the first test point. Before testing the program, you must enter temporary "dummy" lines to satisfy subroutine calls in the program. Add these lines at the end of the current program:

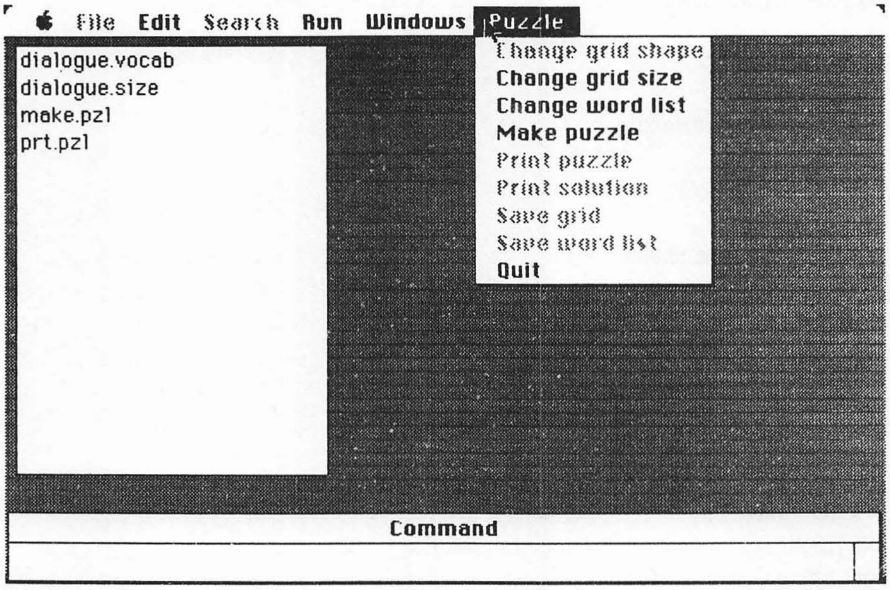
```
dialogue.vocab::
WINDOW 1
PRINT "dialogue.vocab"
RETURN
dialogue.shape:
WINDOW 1
PRINT "dialogue.shape"
RETURN
dialogue.size:
WINDOW 1
PRINT "dialogue.size"
RETURN
make.pzl:
WINDOW 1
PRINT "make.pzl"
RETURN
prt.pzl:
WINDOW 1
```

```
PRINT "prt.pzl"
RETURN
prt.sol:
WINDOW 1
PRINT"prt.sol"
RETURN
save.t:
WINDOW 1
PRINT "save.t"
RETURN
save.vocab:
WINDOW 1
PRINT "save.vocab"
RETURN
```

Now close the listing window and run the program. Your screen should resemble that shown in Figure 2-6.

Each time you select a command from the Puzzle menu, a new line appears in the window, confirming the proper operation of the dummy subroutines.

When you have tested all the active menu items, select Quit from the



```
dialogue.vocab
dialogue.size
make.pzl
prt.pzl
```

Figure 2-6. Screen at test point 1

menu to stop the program. Open the listing window. Delete the following lines from the listing:

```
dialogue.vocab:
WINDOW 1
PRINT "dialogue.vocab"
RETURN
```

The Word List Dialog Box

Now continue adding these lines at the end of the listing:

```
dialogue.vocab:
WINDOW 1
CLS
PRINT "SET UP WORD LIST"
BUTTON 1,1,"Key in new words",(2,32)-(w1.x1%-6,48),3
BUTTON 2,1,"Load new words (disk)",(2,64)-(w1.x1%-6,80),3
BUTTON 3,0,"Edit word list",(2,96)-(w1.x1%-6,112),3
IF wd.count%>0 THEN BUTTON 3,1
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON CLOSE 3
MENU 6,5,0    'disable print puzzle option
MENU 6,6,0    'disable print solution option
ON btn% GOTO key.vocab,disk.vocab,edit.vocab
```

These lines create a dialog box that lets you select three options relating to the word list: enter words from the keyboard, load words from a disk file, or edit the existing word list (if there is one).

The following lines handle the keyboard entry of new words:

```
key.vocab:
CLS
PRINT "KEY IN NEW WORDS"
PRINT
PRINT "How many words?"
PRINT TAB(3);"( 1 -";max.wds%;")";
EDIT FIELD 1,"",(120,48)-(156,63)
```

```
key.loop:
LET event%=DIALOG(0)
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%<>2 AND event%<>6 THEN key.loop:
LET entry=VAL(EDIT$(1))
IF entry<>INT(entry) THEN key.err
IF  entry <&H8000 OR entry>&H7FFF THEN key.err
LET wd.count%=entry
IF wd.count%<1 OR wd.count%>max.wds% THEN key.err
ERASE wd$,wu%,wd.seq%
DIM wd$(wd.count%),wu%(wd.count%),wd.seq%(wd.count%)
EDIT FIELD CLOSE 1
GOTO edit.vocab
key.err:
BEEP
GOTO key.vocab
```

Once you have set the size of the word list, the program creates an array to hold the words and then jumps to the Edit word list option.

The next group of lines handles the input of words from a file stored on disk.

```
disk.vocab:
CLS
PRINT "LOAD NEW WORDS (DISK)"
LET vocab.file$=FILES$(1,"TEXT")   :REM dialog box to select a file
IF vocab.file$=nu$ THEN dialogue.vocab  :REM if cancelled try again
LET wd.count%=0
OPEN vocab.file$ FOR INPUT AS 1
WHILE NOT EOF(1)
LINE INPUT#1, w$
LET wd.count%=wd.count%+1
WEND
CLOSE 1
ERASE wd$,wu%,wd.seq%
DIM wd$(wd.count%),wu%(wd.count%),wd.seq%(wd.count%)
OPEN vocab.file$ FOR INPUT AS 1
FOR j%=1 TO wd.count%
LINE INPUT#1,w$
LET wd$(j%)=UCASE$(w$)
NEXT j%
CLOSE 1
```

The program prompts you to name the file that contains your word list. This should be a file created with the Save word list option. However, it may be any text file that contains one word per line. The UCASE$ function translates the words into uppercase as they are loaded into the array wd$( ).

After the words are loaded from disk, the program executes the Edit word list option:

```
edit.vocab:
CLS
PRINT "EDIT WORD LIST"
PRINT
PRINT "Vocabulary size=";wd.count%
BUTTON 1,1,"BACK",(52,144)-(122,159)
BUTTON 2,1,"FORWARD",(52,176)-(122,191)
BUTTON 3,1,"OK",(52,208)-(122,223)
LET wd.ptr%=1
edit.loop:
LOCATE 5,1
PRINT "Enter word #";wd.ptr%;":"
EDIT FIELD 1,WD$(wd.ptr%),(6,96)-(w1.w%-6,111)
edit.here:
LET event%=DIALOG(0)
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=1 THEN edit.btn
IF event%=2 THEN edit.here
IF event%=6 THEN edit.fld
GOTO edit.loop
```

The program lets you change words and scan forward or backward through the list. Whenever you enter a new word (or click the OK button to stop editing), the program checks the field you entered:

```
edit.fld:
LET wd$(wd.ptr%)=UCASE$(EDIT$(1))
LET wd.ptr%=wd.ptr% MOD wd.count%+1
GOTO edit.loop

edit.btn:
LET wd$(wd.ptr%)=UCASE$(EDIT$(1))
ON DIALOG(1) GOTO edit.back,edit.fwd,done.vocab
```

```
edit.back:
IF wd.ptr%>1 THEN LET wd.ptr%=wd.ptr%-1 ELSE LET wd.ptr%=wd.count%
GOTO edit.loop
edit.fwd:
LET wd.ptr%=wd.ptr% MOD wd.count%+1
GOTO edit.loop
done.vocab:
WINDOW CLOSE 1
MENU 6,8,1    'enable save-word-list option
RETURN
```

Notice that words typed into a field are automatically translated to uppercase with the UCASE$ function.

Test Point 2

You are now ready for the second test point. Run the program. It should automatically begin executing the Change word list command. Try the keyboard entry option. You should see screens similar to those shown in Figures 2-7*a*, *b*, and *c*. Selecting the Edit word list option results in a screen similar to Figure 2-7*c*.

Test the disk input option. When the program prompts you to name a file, press the CANCEL button instead (see Figure 2-7*d*). You can fully test this option later on.

The Grid Size Dialog Box

Now stop the program (select Quit from the menu) and open the listing window. Find these lines and delete them:

```
dialogue.size:
WINDOW 1
PRINT "dialogue.size"
RETURN
```

Now add these lines, which create a grid size dialog box:

```
dialogue.size:
WINDOW 1
CLS: ON ERROR GOTO 0
PRINT "SET PUZZLE SIZE"
BUTTON 1,1,"Key in new grid size",(2,32)-(w1.x1%-6,48),3
BUTTON 2,1,"Load new grid (disk)",(2,64)-(w1.x1%-6,80),3
BUTTON 3,0,"Edit current grid size",(2,96)-(w1.x1%-6,112),3
IF g.size%>0 THEN BUTTON 3,1
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON CLOSE 3
MENU 6,5,0   'disable print puzzle option
MENU 6,6,0   'disable print solution option
ON btn% GOTO key.grid,disk.grid,dialogue.shape
```

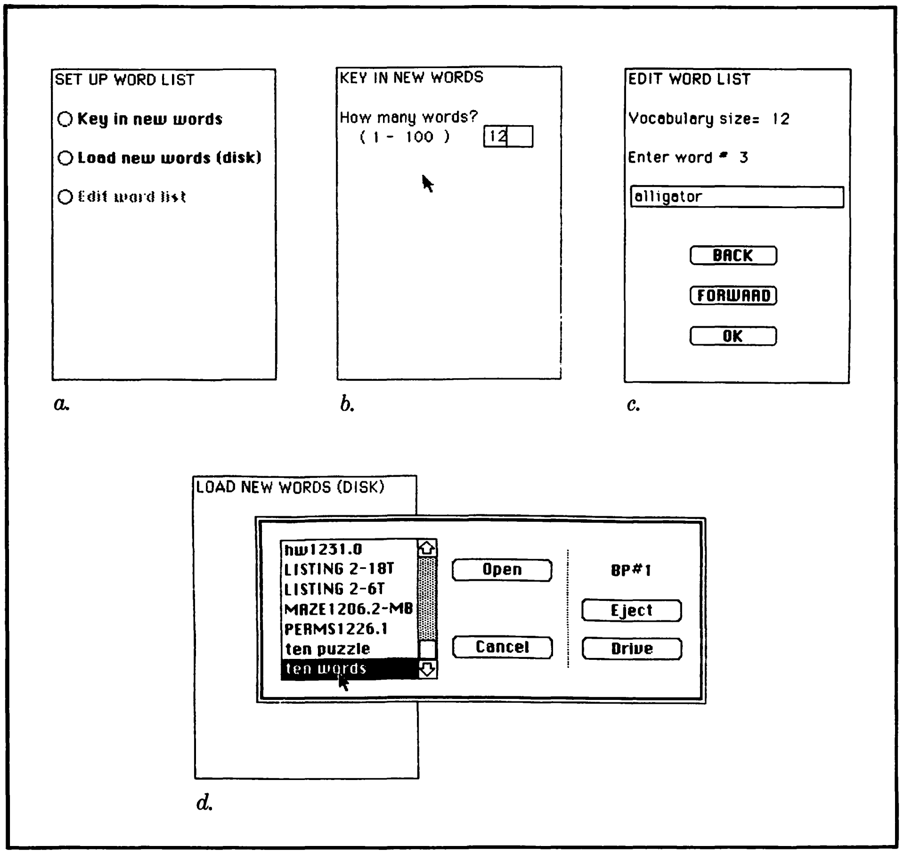


Figure 2-7. Test point 2: dialog boxes for setting up word list (*a*), specifying list size (*b*), entering and editing words (*c*), and loading a word list from disk (*d*)

These lines give you the option of resetting the grid size, loading a grid from disk, or leaving the existing grid as is (if a grid has been set up).

Here is the block that prompts you to specify the grid size from the keyboard:

```
key.grid:
CLS
PRINT "KEY IN NEW GRID SIZE"
PRINT
PRINT "How many rows?"
PRINT TAB(3); "( 1-";max.r%;")"
PRINT
PRINT "How many columns?"
PRINT TAB(3); "( 1-";max.c%;")";
EDIT FIELD 2,"",(120,96)-(156,111)
EDIT FIELD 1,"",(120,48)-(156,63)
BUTTON 1,0,"OK",(52,186)-(122,213)
LET fld%=1
LET nxt.fld%=1
LET r.ok%=no%
LET c.ok%=no%
```

The next series of lines gets your input for the number of rows and columns in the grid:

```
grid.loop:
BUTTON 1,c.ok%*r.ok%
LET event%=DIALOG(0)
IF event%=1 THEN GOTO check.fld
IF event%=2 THEN LET nxt.fld%=DIALOG(2): GOTO check.fld
IF event%=6 THEN LET nxt.fld%=(fld% MOD 2)+1: GOTO check.fld
GOTO grid.loop
check.fld:
LET entry=VAL(EDIT$(fld%))
IF entry<>INT(entry) THEN fld.err
IF  entry<-32768 OR entry>32767 THEN fld.err
ON fld% GOTO check.row,check.col
```

The check.fld routine ensures that each value you enter is within integer range, and then the program executes the appropriate routine to check for value row and column specifications:

```
check.row:
LET last.r%=entry
LET r.ok%=(last.r%>=1 AND last.r%<=max.r%)
IF r.ok%=no% THEN fld.err
GOTO fld.ok
check.col:
LET last.c%=entry
LET c.ok%=(last.c%>=1 AND last.c%<=max.c%)
IF c.ok%=no% THEN fld.err
fld.ok:
IF event%=1 THEN grid.ok
LET fld%=nxt.fld%
EDIT FIELD fld%
GOTO grid.loop
fld.err:
BEEP
EDIT FIELD fld%
GOTO grid.loop
grid.ok:
EDIT FIELD CLOSE 1
EDIT FIELD CLOSE 2
BUTTON CLOSE 1
GOSUB grid.arrays
MENU 6,1,1    'enable change-shape option
MENU 6,7,1    'enable save-shape option
GOTO dialogue.shape
```

The program will not let you continue until you have entered valid data for the row and column size.

Here are the lines that load a grid from disk:

```
disk.grid:
CLS
PRINT "LOAD NEW GRID (DISK)"
LET grid.file$=FILES$(1,"TEXT")
IF grid.file$=nu$ THEN dialogue.size
ON ERROR GOTO grid.file.err
OPEN grid.file$ FOR INPUT AS 1
INPUT#1,last.r%,last.c%
GOSUB grid.arrays
FOR r%=1 TO last.r%
FOR c%=1 TO last.c%
INPUT#1,grid$(r%,c%)
```

```
NEXT c%,r%
CLOSE 1
ON ERROR GOTO 0
WINDOW CLOSE 1
MENU 6,1,1   'enable change-shape option
MENU 6,7,1   'enable save-shape option
GOTO dialogue.shape
```

The program prompts you to specify the name of a previously saved puzzle (using an option presented later). The following lines handle errors that may occur during the loading of the grid file:

```
grid.file.err:
CLOSE 1
LET errcode%=ERR
IF errcode%<>6 AND errcode%<>13 AND errcode%<>62 THEN unknown.err
BEEP
PRINT "Invalid data in"
PRINT grid.file$
BUTTON 1,1,"OK",(52,220)-(102,240),1
WHILE DIALOG(0)<>1
WEND
RESUME dialogue.size
unknown.err:
ON ERROR GOTO 0
grid.arrays:
LET g.size%=last.r%*last.c%
ERASE grid$,cell.seq%
DIM grid$(last.r%,last.c%),cell.seq%(g.size%)
RETURN
```

If you specify a non-puzzle file, the program will recognize that the format is incorrect and will return you to the puzzle size dialog box.

The grid.arrays subroutine is used by the keyboard and disk input routines to set up an array to hold the grid values.

Make Puzzle Command

Locate the following lines and delete them:

```
make.pzl:
WINDOW 1
PRINT "make.pzl"
RETURN
```

Here are the lines that handle the Make puzzle command:

```
make.pzl:
WINDOW 1
CLS
PRINT "NEW PUZZLE STATUS"
PRINT
BUTTON 1,1,"CANCEL",(52,220)-(116,240),1
LET c.flag%=no%
DIALOG ON
ON DIALOG GOSUB rq.cancel
GOSUB erase.grid
GOSUB sort.words
IF c.flag%=yes% THEN cancel.pzl
GOSUB shuffle
IF c.flag%=yes% THEN cancel.pzl
GOSUB auto.fill
IF c.flag%=yes% THEN cancel.pzl
GOSUB random.fill
IF c.flag%=yes% THEN cancel.pzl
DIALOG OFF
BEEP
PRINT "Puzzle is ready"
BUTTON 1,1,"OK",(52,220)-(102,240),1
WHILE DIALOG(0)<>1
WEND
WINDOW CLOSE 1
MENU 6,5,1    'enable print-puzzle option
MENU 6,6,1    'enable print-solution option
RETURN
cancel.pzl:
DIALOG OFF
WINDOW CLOSE 1
RETURN
rq.cancel:
IF DIALOG(0)=1 THEN LET c.flag%=yes%
RETURN
```

A series of subroutine calls actually produces the puzzle. To allow testing of the program, type in these temporary lines:

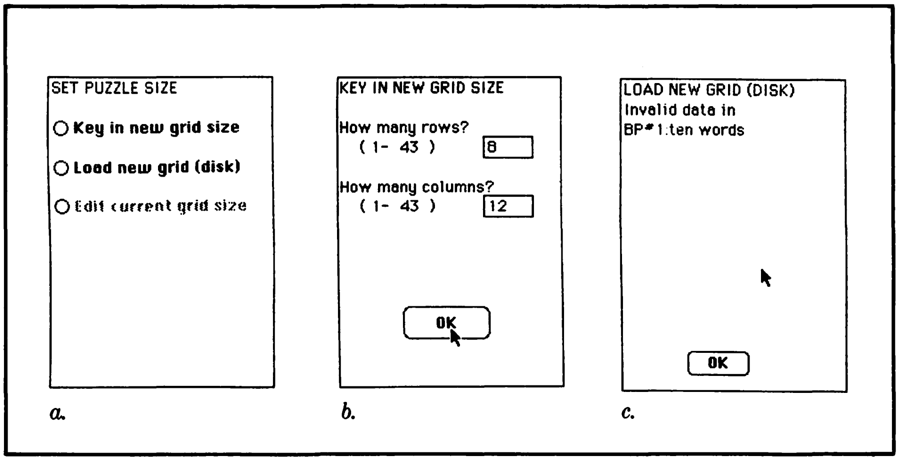
```
erase.grid:
PRINT "erase.grid"
RETURN
```

```
sort.words:
PRINT "sort.words"
RETURN
shuffle:
PRINT "shuffle"
RETURN
auto.fill:
PRINT "auto.fill"
RETURN
random.fill:
PRINT "random.fill"
FOR j%=1 TO 8000: NEXT j%
RETURN
```

Test Point 3

Now you can test the Change grid size command and the Make puzzle command. Run the program. After prompting you to enter the word list, the program will display the Change grid size dialog boxes shown in Figure 2-8*a*, *b*, and *c*. Try the keyboard option first.

Reselect the Change grid size command from the menu, and try the Load new grid option. Of course, you don't yet have a puzzle file to load, but try loading some other text file; the program should tell you that the


```
SET PUZZLE SIZE

O Key in new grid size

O Load new grid (disk)

O Edit current grid size
```
a.

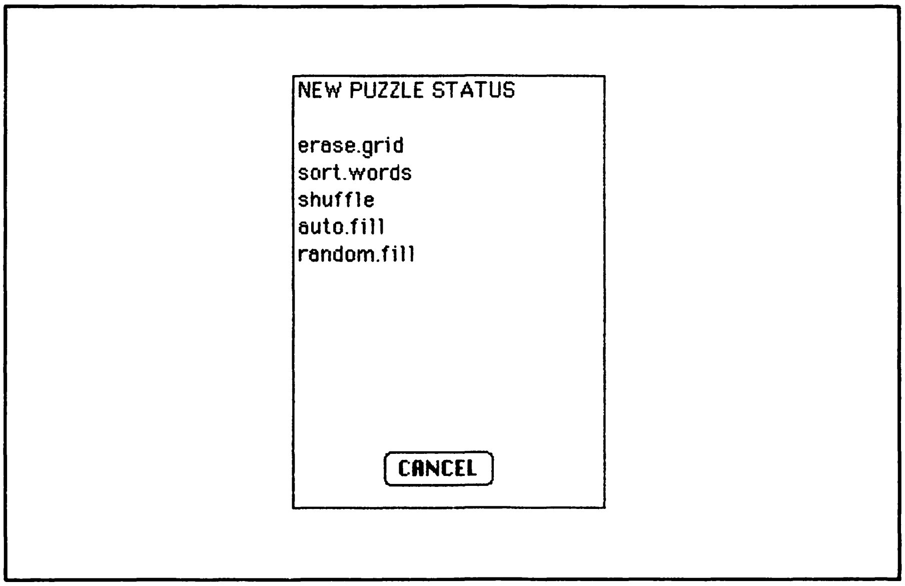
```
KEY IN NEW GRID SIZE

How many rows?
  ( 1- 43 )        8

How many columns?
  ( 1- 43 )        12
```

b.

```
LOAD NEW GRID (DISK)
Invalid data in
BP*1:ten words
```

c.

Figures 2-8. Test point 3: dialog boxes for setting the puzzle size (*a*), specifying a new size (*b*), and handling a disk error (*c*)


NEW PUZZLE STATUS

erase.grid
sort.words
shuffle
auto.fill
random.fill

Figure 2-9. Test point 3: Status indicator during the Make puzzle procedure

file format is incorrect and should display the dialog box that is shown in Figure 2-8*c*.

After you specify the grid size or load a grid from disk, the program will execute the Make puzzle command, during which time you should see a dialog box like that shown in Figure 2-9.

This completes part 1 of the program. Go back through the available options, making sure that everything works as shown in Figures 2-7, 2-8, and 2-9. Then continue with the next chapter.

This chapter has been adapted from "Hidden Words" by George Stewart, appearing in the December 1983 issue of *Popular Computing* magazine. Copyright 1983 Byte Publications, Inc. Used with the permission of Byte Publications, Inc.

Chapter 3

Hidden Words—Part 2

In the last chapter you completed the Hidden Words program up through the Change grid size procedure. This chapter presents the rest of the program: the Change grid shape procedure and the logic for filling the puzzle grid with words. Test points are provided to help you check your work as you go along.

—The Program

Load the program from the last chapter into Microsoft BASIC. Open the listing window and delete these lines:

```
dialogue.shape:
WINDOW 1
PRINT "dialogue.shape"
RETURN
```

Editing the Grid Shape

Now add these lines, which set up the shape dialog box and the shape editing window:

```
dialogue.shape:
LET aw%=last.c%*1.tot%-1.side%+2*border%
LET al%=last.r%*1.tot%-1.side%+2*border%
WINDOW 2,,(w2.x%,w2.y%)-(w2.x%+aw%,w2.y%+al%),3
WINDOW 1
MENU 6,5,0   'disable print puzzle option
MENU 6,6,0   'disable print solution option
CLS
PRINT "EDIT PUZZLE SHAPE"
PRINT
PRINT "Cursor function:"
BUTTON 1,0,"ERASE",(52,64)-(122,79),2
BUTTON 2,0,"FILL",(52,96)-(122,111),2
BUTTON 3,1,"OK",(52,208)-(122,235),1
```

The shape editing window is just large enough to hold the number of rows and columns you requested. Figure 3-1 shows the initial appearance of the shape editing window formed by an 11 × 13 grid.

The next lines activate a dialog event trap and draw the current grid shape in window 2:

```
LET shape.done%=no%
DIALOG ON
ON DIALOG GOSUB shape.interrupt
WINDOW OUTPUT 2
LET color%=1
FOR r%=1 TO last.r%
FOR c%=1 TO last.c%
IF shape.done%=yes% THEN LET r%=last.r%: LET c%=last.c%: GOTO skip
IF grid$(r%,c%)<>hole$ THEN GOSUB set.reset
skip:
NEXT c%,r%
LET color%=0
WINDOW OUTPUT 1
BUTTON 1,2
BUTTON 2,1
```

The event trap lets you change windows to draw in window 2 or to change the settings shown in the dialog box (window 1).

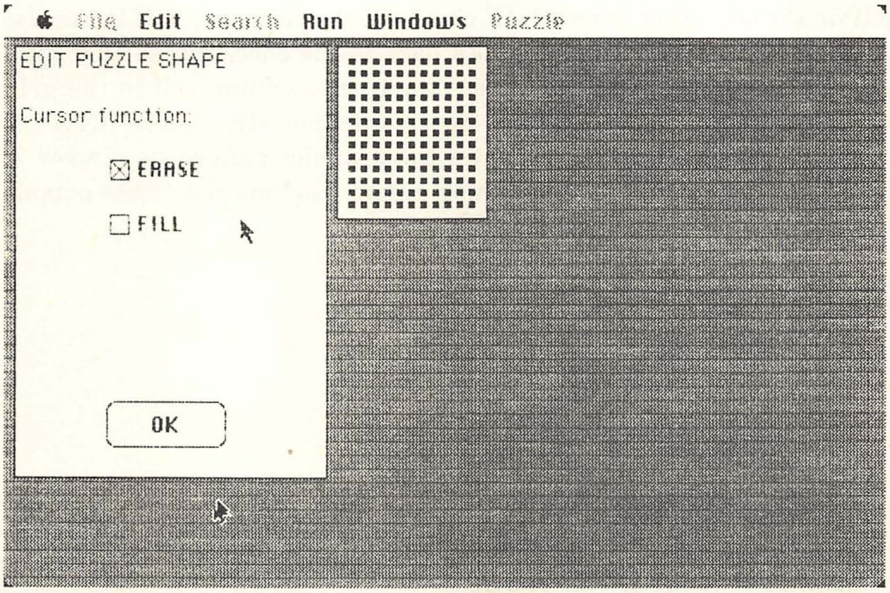


Figure 3-1. Test point 1, the Change grid shape dialog screen

After the windows are drawn, the program begins checking for mouse activity (clicking or dragging the mouse) in the active window.

The following lines take effect only while the mouse is in the shape editing window.

```
check.mouse:
LET mouse.status%=ABS(MOUSE(0))
WHILE mouse.status%<>1
IF shape.done%=yes% THEN done
LET mouse.status%=ABS(MOUSE(0))
WEND
LET mouse.x%=MOUSE(1)
LET mouse.y%=MOUSE(2)
LET c%=(mouse.x%-border%+1.tot%)\1.tot%
LET r%=(mouse.y%-border%+1.tot%)\1.tot%
IF c%<1 OR c%>last.c% OR r%<1 OR r%>last.r% THEN check.mouse
IF color%=0 THEN LET grid$(r%,c%)=hole$ ELSE LET grid$(r%,c%)=ltr.cell$
GOSUB set.reset
GOTO check.mouse
```

The program repeatedly executes the WHILE/WEND loop until the mouse is clicked. The program then determines whether the mouse is within the bounds of the grid (IF c%<1 OR c%>last.c%...). If the mouse is out of bounds, the program goes back to the check.mouse loop. Otherwise, the program fills or erases the corresponding cell in the grid array, depending on the current cursor function. (IF color%=0...)

The set.reset subroutine updates the graphics pattern in window 2.

The following lines handle dialog events (clicking the mouse outside the active window or pressing a button):

```
shape.interrupt:
LET event%=DIALOG(0)
IF event%=3 THEN change.windows
IF event%<>1 THEN RETURN
LET btn%=DIALOG(1)
ON btn% GOTO set.color,set.color,request.end
change.windows:
LET rq.w%=DIALOG(3)
WINDOW rq.w%
IF rq.w%=1 THEN CALL INITCURSOR
IF rq.w%=2 THEN CALL SETCURSOR(VARPTR(cursor%(0)))
RETURN
set.color:
WINDOW OUTPUT 1
BUTTON btn%,2
BUTTON 3-btn%,1
LET color%=btn%-1
WINDOW OUTPUT 2
RETURN
request.end:
LET shape.done%=yes%
RETURN
```

A dialog event of 3 indicates you pressed the mouse in an inactive window. In that case, the program goes to the change.windows routine. A dialog event of 1 indicates you pressed a button; since only window 1 has buttons, the program tests for buttons 1, 2, or 3 (FILL, ERASE, OK). Any other dialog event is ignored (IF event%<>1 THEN RETURN).

Note that the change.windows routine also changes the cursor; if you have selected window 1, the default cursor (INITCURSOR) is used. If you have selected window 2, the X-cursor (SETCURSOR...) is used.

The set.color routine handles the selection of the FILL and ERASE buttons. The routine request.end handles the selection of the OK button by setting a flag that will be noticed by the routine in progress when the dialog event occurs.

The following lines set or erase grid blocks and end the Change grid shape procedure:

```
set.reset:
LET char.x%=(c%-1)*l.tot%+border%
LET char.y%=(r%-1)*l.tot%+border%
LINE (char.x%,char.y%)-STEP (l.side%,l.side%),color%,bf
RETURN
done:
CALL INITCURSOR
DIALOG OFF
WINDOW CLOSE 2
WINDOW CLOSE 1
RETURN
```

Depending on the value of color% (zero or non-zero), the LINE statement either erases or fills a grid block.

The routine done, executed when you press the OK button, restores the default cursor (the pointer), terminates dialog event trapping, and closes both windows.

Test Point 1

First, save the program in its current state in a disk file. Now you can test the Change grid shape routine. Run the program. Enter a short word list. Specify a grid size of 13 rows × 11 columns. You should see the screen pictured in Figure 3-1.

Move to the graphics window and click the mouse. The cursor should change to an X. Now press the button on top of each block you want to erase. To restore a block (fill it in again), go back to the cursor-function window and select FILL. Try to create the pattern shown in Figure 3-2. Remember that to change functions, you must press the mouse button two separate times (not a double-click): once to activate the inactive window, and a second time to specify the desired cursor function.

Press OK when you are done; you should see the new puzzle status window and a notice that the puzzle is ready (it really isn't; we still have to add the puzzle fill routines).

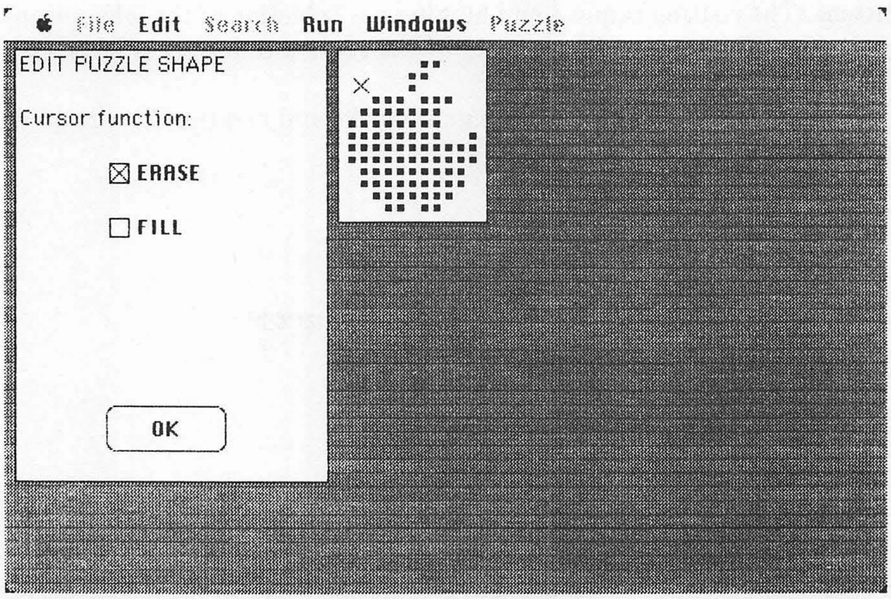


Figure 3-2. Test point 1, the Change grid shape dialog screen showing the X-cursor and a design in the graphics window

Select the Change grid shape command from the Puzzle menu. You should be able to edit the shape you created previously.

If the computer should operate abnormally (screen image becomes garbled or other unusual behavior), requiring you to turn the computer off and on again, you may have typed in the cursor definition numbers incorrectly. Carefully recheck the DATA statements that you entered at the beginning of Chapter 2.

Puzzle Fill Logic

Now we come to the subroutines that actually fill-in the hidden word puzzle. First locate the following lines and delete them:

```
erase.grid:
PRINT "erase.grid"
RETURN
```

Erase Grid Subroutine

Now type in the erase.grid subroutine as follows:

```
erase.grid:
PRINT "Erasing the puzzle grid..."
FOR j%=1 TO last.r%
FOR k%=1 TO last.c%
IF grid$(j%,k%)<>hole$ THEN LET grid$(j%,k%)=ltr.cell$
IF c.flag%=yes% THEN LET j%=last.r%: LET k%=last.c%
NEXT k%,j%
RETURN
```

This routine leaves holes unchanged, but changes everything else to a letter cell (IF grid$(j%,k%)<>hole$. . .).

After each cell is checked, the program checks the cancel flag (c.flag%). Recall that during the Make puzzle procedure, you can press a CANCEL button (see Figure 2-9). If you do so, the c.flag% will be set, causing the subroutine erase.grid to terminate early.

Word Sort Subroutine

Locate and delete the following lines:

```
sort.words:
PRINT "sort.words"
RETURN
```

Now type in the sort.words subroutine, which sorts the words according to length, so the longest can be tried first in each potential path.

```
sort.words:
PRINT "Sorting the words..."
FOR j%=1 TO wd.count%
LET wd.seq%(j%)=j%
NEXT j%
LET lw%=wd.count%
bubble.sort:
IF lw%=1 THEN sorted
LET sort.ok%=yes%
```

```
FOR j%=1 TO lw%-1
LET l2%=LEN(wd$(wd.seq%(j%+1)))
LET l1%=LEN(wd$(wd.seq%(j%)))
IF l2%>l1% THEN SWAP wd.seq%(j%),wd.seq%(j%+1): sort.ok%=no%
IF c.flag%=yes% THEN LET sort.ok%=yes%: LET lw%=1
NEXT j%
IF sort.ok%=yes% THEN sorted
LET lw%=lw%-1
GOTO bubble.sort
sorted:
FOR j%=1 TO wd.count%
LET wu%(j%)=not.used%
IF c.flag%=yes% THEN LET j%=wd.count%
NEXT j%
RETURN
```

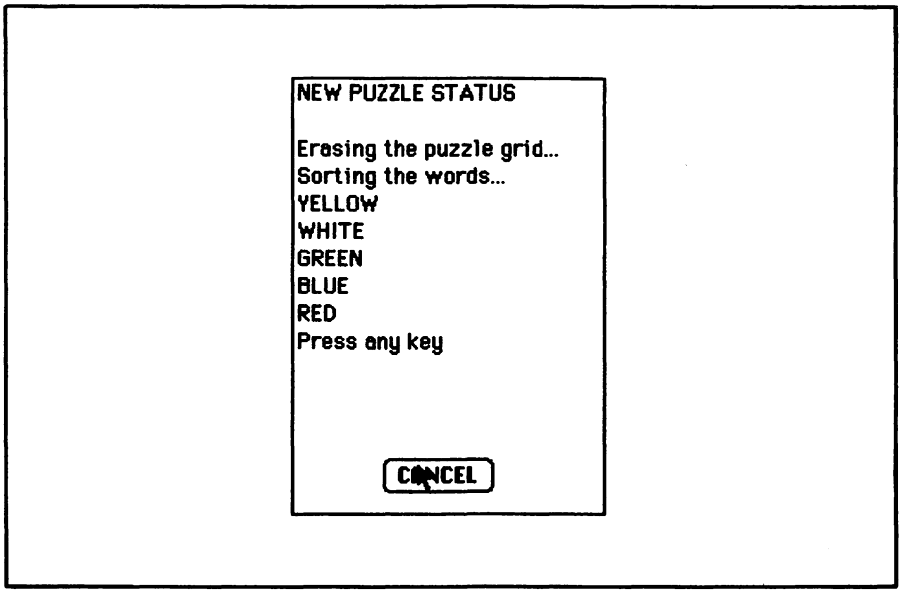
These lines perform a bubble sort, going through the list and comparing each word with its successor. If the successor is longer, the words are swapped. The routine goes through the list repeatedly until no more swaps can be made.

Test Point 2

Add the following lines to the sort.words subroutine just *before* the final RETURN statement; these lines will allow you to test the sort.words subroutine:

```
FOR j%=1 TO wd.count%
PRINT wd$(wd.seq%(j%))
NEXT j%
PRINT "Press any key"
WHILE INKEY$=nu$
WEND
```

Now run the program. Type in these words: **red**, **white**, **blue**, **green**, **yellow**. Complete the grid size and grid shape dialogs with arbitrarily chosen data. The program should then enter the Make puzzle procedure, erasing the grid and sorting the words. After the words are sorted, they should be displayed in window 1 as shown in Figure 3-3.



```
NEW PUZZLE STATUS

Erasing the puzzle grid...
Sorting the words...
YELLOW
WHITE
GREEN
BLUE
RED
Press any key
```

Figure 3-3. Test point 2, word list in order of word length

Before continuing, locate the following test lines and delete them:

```
FOR j%=1 TO wd.count%
PRINT wd$(wd.seq%(j%))
NEXT j%
PRINT "Press any key"
WHILE INKEY$=nu$
WEND
```

Grid Shuffle Subroutine

First locate and delete the following lines:

```
shuffle:
PRINT "shuffle"
RETURN
```

Now type in the following subroutine, which shuffles the grid cells so the program will attempt to fill them in random order.

```
shuffle:
PRINT "Shuffling the cells..."
FOR j%=1 TO g.size%
LET cell.seq%(j%)=0
NEXT j%
FOR j%=1 TO g.size%
find.unused:
LET g.ptr%=INT(RND*g.size%)+1
IF cell.seq%(g.ptr%)<>0 THEN find.unused
LET cell.seq%(g.ptr%)=j%
IF c.flag%=yes% THEN LET j%=g.size%
NEXT j%
RETURN
```

Auto Fill Subroutine

Locate and delete these lines:

```
auto.fill:
PRINT "auto.fill"
RETURN
```

The following lines control the puzzle fill-in process:

```
auto.fill:
LOCATE 7,1
PRINT "Filling in the puzzle..."
PRINT "Pass #"
PRINT "Words used="
PRINT "Cells checked="
LET dir%=INT(RND*max.dir%)+1
LET wds.left%=wd.count%
LET pass.num%=1
af.loop:
LOCATE 8,7
PRINT USING "#";pass.num%
GOSUB next.pass
PRINT
IF pass.num%=2 OR wds.left%=0 OR c.flag%=yes% THEN af.done
LET pass.num%=2
GOTO af.loop
af.done:
RETURN
```

The program makes two passes through the list of grid cells, as explained later on. When both passes are complete, the program is finished filling in words. Later, any unfilled letter cells will be filled at random.

The next lines perform pass 1 and 2 through the grid cells.

```
next.pass:
LET g.ptr%=1
np.loop:
GOSUB cell.check
LOCATE 9,11
PRINT USING "**";wd.count%-wds.left%
LOCATE 10,13
PRINT USING "***";g.ptr%
IF wds.left%=0 OR g.ptr%=g.size% OR c.flag%=yes% THEN np.done
LET g.ptr%=g.ptr%+1
GOTO np.loop
np.done:
RETURN
```

The variable g.ptr% points to the cell currently being examined; for instance, g.ptr%=1 indicates that the first cell (in the shuffled sequence) is under examination.

The following subroutine checks to see whether a word can be entered into the puzzle starting with the current cell:

```
cell.check:
LET wd.ptr%=1
LET word.fit%=no%
LET cell.num%=cell.seq%(g.ptr%)
LET row%=(cell.num%-1)\last.c%+1
LET col%=(cell.num%-1) MOD last.c%+1
LET t$=grid$(row%,col%)
IF pass.num%=1 THEN LET skip.it%=(t$=hole$)
IF pass.num%=2 THEN LET skip.it%=(t$=hole$) OR (t$=ltr.cell$)
IF skip.it%=yes% THEN cc.done
cc.loop:
GOSUB word.check
IF word.fit%=yes% OR wd.ptr%=wds.left% OR c.flag%=yes% THEN cc.done
LET wd.ptr%=wd.ptr%+1
GOTO cc.loop
cc.done:
RETURN
```

During pass 1, all letter cells are considered. During pass 2, only filled-in cells are considered. This allows the program to start two words at the same cell.

To check a cell, the program tries every unused word to see if it fits in one of the available directions.

Here are the lines that check whether a given word can be entered starting with the current cell:

```
word.check:
LET wd.num%=wd.seq%(wd.ptr%)
LET try.wd$=wd$(wd.num%)
LET wl%=LEN(try.wd$)
LET dir.count%=1
wc.loop:
GOSUB dir.check
IF word.fit%=yes% THEN LET dir%=dir% MOD max.dir%+1: GOTO wc.done
IF dir.count%=max.dir% THEN wc.done
LET dir.count%=dir.count%+1
LET dir%=dir% MOD max.dir%+1
GOTO wc.loop
wc.done:
RETURN
```

For each word examined, the program tries all possible directions before giving up on that word.

The following lines determine whether the word try.wd$ can be placed in the grid starting at row%, col% in the direction specified by DIR%:

```
dir.check:
LET f.row%=row%+(wl%-1)*ri%(dir%)
LET f.col%=col%+(wl%-1)*ci%(dir%)
LET r.ok%=(f.row%>=1) AND (f.row%<=last.r%)
LET c.ok%=(f.col%>=1) AND (f.col%<=last.c%)
IF NOT (r.ok% AND c.ok%) THEN dc.done
LET word.fit%=yes%
LET pr%=row%
LET pc%=col%
FOR l%=1 TO wl%
LET t$=grid$(pr%,pc%)
LET word.fit%=(t$=ltr.cell$) OR (t$=MID$(try.wd$,l%,1))
IF word.fit%=no% THEN LET l%=wl%: GOTO nxt
LET pr%=pr%+ri%(dir%)
LET pc%=pc%+ci%(dir%)
nxt:
NEXT l%
```

The program traces the proposed path in the grid, comparing each letter of the word with the corresponding cell in the grid path.

If no conflicts are found, word.fit% is set to yes% at the end of this routine.

The next block of lines handles the result of the word-fit test:

```
IF word.fit%=no% THEN dc.done
LET pr%=row%
LET pc%=col%
FOR l%=1 TO wl%
LET grid$(pr%,pc%)=MID$(try.wd$,l%,1)
LET pr%=pr%+ri%(dir%)
LET pc%=pc%+ci%(dir%)
NEXT l%
IF wd.ptr%>wds.left% THEN cut.word
FOR j%=wd.ptr% TO wds.left%-1
LET wd.seq%(j%)=wd.seq%(j%+1)
NEXT j%
cut.word:
LET wds.left%=wds.left%-1
LET wu%(wd.num%)=(dir%-1)*g.size%+cell.num%-1
dc.done:
RETURN
```

If the word fits, these lines embed it one letter at a time into the grid. In this case, the word that was used is removed from the sorted word list.

Random Fill Subroutine

Once the auto fill subroutine has completed both passes through the grid, the remaining letter cells are filled with randomly chosen letters.

Before adding the lines for the random fill routine, locate and delete the following lines:

```
random.fill:
PRINT "random.fill"
FOR j%=1 TO 8000: NEXT j%
RETURN
```

Now type in the random fill subroutine:

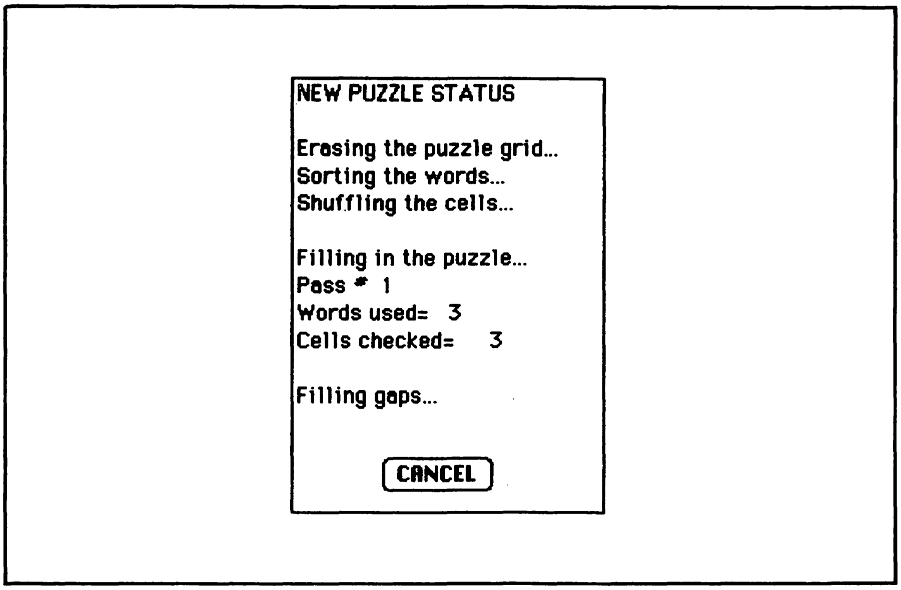
```
random.fill:
LOCATE 12,1
PRINT "Filling gaps..."
FOR row%=1 TO last.r%
FOR col%=1 TO last.c%
IF grid$(row%,col%)<>ltr.cell$ THEN nxt.fill
LET grid$(row%,col%)=CHR$(INT(RND*26)+65)
nxt.fill:
IF c.flag%=yes% THEN LET col%=last.c%: LET row%=last.r%
NEXT col%,row%
RETURN
```

Test Point 3

To test the puzzle fill logic, add these lines to the random fill subroutine immediately *before* the RETURN statement:

```
CLS
FOR row%=1 TO last.r%
FOR col%=1 TO last.c%
PRINT grid$(row%,col%);
NEXT col%
PRINT
NEXT row%
PRINT "Press any key"
WHILE INKEY$=nu$
WEND
```

Run the program. Specify a short word list (man,bites,dog) and a 5 × 5 grid size. Define a simple shape so the words will easily fit.



NEW PUZZLE STATUS

Erasing the puzzle grid...
Sorting the words...
Shuffling the cells...

Filling in the puzzle...
Pass # 1
Words used= 3
Cells checked= 3

Filling gaps...

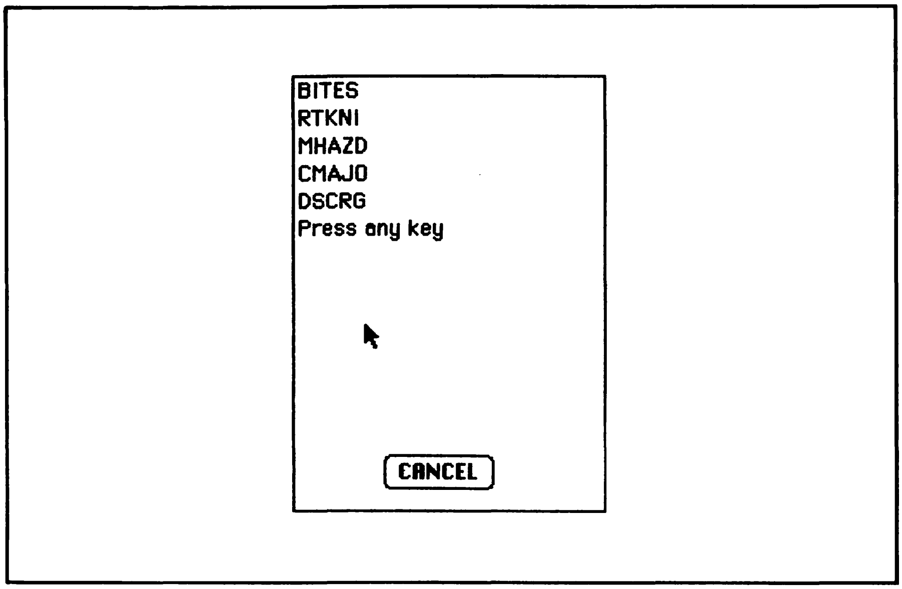
Figure 3-4. Test point 3, puzzle fill in progress

During the auto fill process, your screen should show the new puzzle status similar to Figure 3-4.

When the puzzle is complete, the program should print a copy of the puzzle in window 1 similar to that shown in Figure 3-5.

Before continuing, delete the following lines from the listing:

```
CLS
FOR row%=1 TO last.r%
FOR col%=1 TO last.c%
PRINT grid$(row%,col%);
NEXT col%
PRINT
NEXT row%
PRINT "Press any key"
WHILE INKEY$=nu$
WEND
```



```
BITES
RTKNI
MHAZD
CMAJO
DSCRG
Press any key
```

Figure 3-5. Test point 3, puzzle printout (hidden words are: man, bites, dog)

Save Grid Subroutine

Now we'll present the lines that save a grid pattern (its size and shape). But first locate and delete the following dummy lines:

```
save.t:
WINDOW 1
PRINT "save.t"
RETURN
```

Now enter the save grid routine:

```
save.t:
WINDOW 1
CLS
PRINT "SAVE PUZZLE GRID"
grid.file$=FILES$(0)
IF grid.file$=nu$ THEN st.done
OPEN grid.file$ FOR OUTPUT AS 1
```

```
WRITE#1,last.r%,last.c%
FOR r%=1 TO last.r%
FOR c%=1 TO last.c%
WRITE#1,grid$(r%,c%)
NEXT c%,r%
CLOSE 1
st.done:
WINDOW CLOSE 1
RETURN
```

Save Word List Subroutine

Locate and delete the following lines:

```
save.vocab:
WINDOW 1
PRINT "save.vocab"
RETURN
```

Now enter these lines, which let you save the word list in a disk file:

```
save.vocab:
WINDOW 1
CLS
PRINT "SAVE WORD LIST"
vocab.file$=FILES$(0)
IF vocab.file$=nu$ THEN sv.done
OPEN vocab.file$ FOR OUTPUT AS 1
FOR j%=1 TO wd.count%
IF wd$(j%)=nu$ THEN skip.null
PRINT#1, wd$(j%)
skip.null:
NEXT j%
CLOSE#1
sv.done:
WINDOW CLOSE 1
RETURN
```

Any null entries that are in your word list are not saved (IF wd$(j%)=nu$...). The words are saved one per line in a text file. The file may be reloaded by the Hidden Words program or by a word processing program.

Print Subroutines

Locate and delete the following lines:

```
prt.pzl:
WINDOW 1
PRINT "prt.pzl"
RETURN
```

The following subroutine prints a copy of the completed puzzle on the screen, printer, or Clipboard:

```
prt.pzl:
GOSUB select.device
CALL TEXTFONT(4)
CALL TEXTSIZE(9)
CALL TEXTFACE(1)
FOR tr%=1 TO last.r%
FOR tc%=1 TO last.c%
PRINT#1,, grid$(tr%,tc%);
NEXT tc%
PRINT#1,
NEXT tr%
CLOSE 1
CALL TEXTSIZE(12)
CALL TEXTFONT(3)
CALL TEXTFACE(0)
RETURN
```

The puzzle must be printed in a monospace font; that is, one in which every letter uses the same amount of space on a line; otherwise the columns will not line up correctly and the shape will be distorted. For this reason, font number 4 (Monaco) was used. Text size 9 and text face 1 were selected for appearance's sake. **Note:** the monospaced font is used only for output to the screen.

Locate and delete the following lines:

```
prt.sol:
WINDOW 1
PRINT"prt.sol"
RETURN
```

The next lines print a copy of the puzzle solution on the screen, printer, or Clipboard.

```
prt.sol:
GOSUB select.device
CALL TEXTSIZE(9)
CALL TEXTFONT(4)
CALL TEXTFACE(1)
PRINT#1, "The hidden words are: "
PRINT#1, "Word (row:col:direction)"
FOR j%=1 TO wd.count%
IF wu%(j%)=not.used% THEN nxt.sol
LET dir%=wu%(j%)\g.size%+1
LET cell.num%=wu%(j%)-(dir%-1)*g.size%+1
LET row%=(cell.num%-1)\last.c%+1
LET col%=(cell.num%-1) MOD last.c%+1
PRINT#1, USING "& (## :## :##)";wd$(j%),row%,col%,dir%
nxt.sol:
NEXT j%
CLOSE 1
CALL TEXTSIZE(12)
CALL TEXTFONT(3)
CALL TEXTFACE(0)
RETURN
```

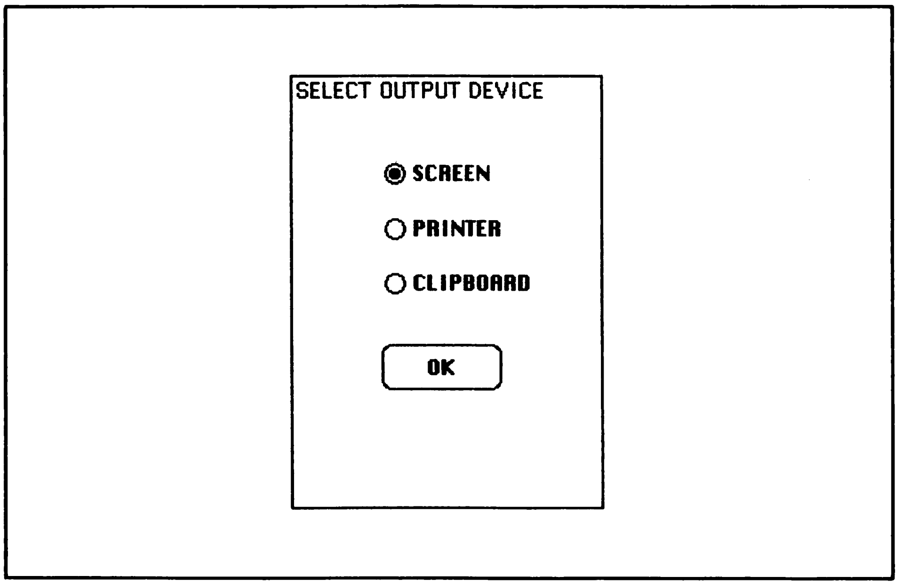
The final subroutine of the program lets you select the output device for printing:

```
select.device:
WINDOW 1
CLS
PRINT "SELECT OUTPUT DEVICE"
LET device%=1
BUTTON 1,2,"SCREEN",(52,48)-(122,63),3
BUTTON 2, 1,"PRINTER",(52,80)-(122,95),3
BUTTON 3,1,"CLIPBOARD",(52,112)-(142,127),3
BUTTON 4,1,"OK",(52,156)-(122,183),1
sd.loop:
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
```

```
IF btn%=4 THEN dev.ok
LET device%=btn%
BUTTON btn%,2
BUTTON btn% MOD 3+1,1
BUTTON (btn%+1)MOD 3+1,1
GOTO sd.loop
dev.ok:
WINDOW CLOSE 1
IF device%=1 THEN WINDOW 2,,(w1.x%,w1.y%)-(w2.x2%,w2.y2%),3
WIDTH  dev$(device%),255,ZONE%
OPEN dev$(device%) FOR OUTPUT AS 1
RETURN
```

—Testing and Using the Program

Figure 3-6 shows the select output device dialog box. Run the program and try each of the three devices. After using the Clipboard, end the program (Quit command), exit from BASIC to the Finder, and examine the Clipboard to see if it holds your puzzle (or puzzle solution). You can-

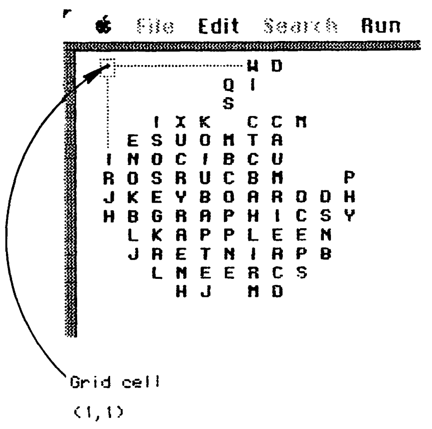

SELECT OUTPUT DEVICE

(●) SCREEN

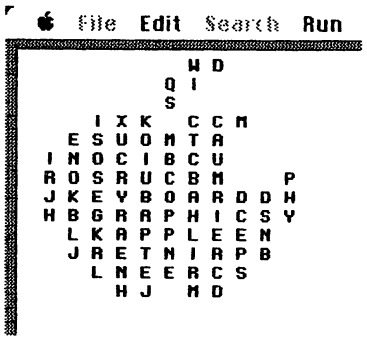
○ PRINTER

○ CLIPBOARD

Figure 3-6. The select output device dialog box



```
            W D
          Q I
          S
    I X K   C C M
  E S U O M T A
I N O C I B C U
R O S R U C B M     P
J K E Y B O A R O D H
H B G R A P H I C S Y
  L K A P P L E E N
  J A E T N I A P B
    L M E E R C S
      H J   M D
```



```
            W D
          Q I
          S
    I X K   C C M
  E S U O M T A
I N O C I B C U
R O S R U C B M     P
J K E Y B O A R O D H
H B G R A P H I C S Y
  L K A P P L E E N
  J A E T N I A P B
    L M E E R C S
      H J   M D
```

Figure 3-7. Sample puzzle output to the screen (note the position of grid cell (1,1) in the upper left corner)

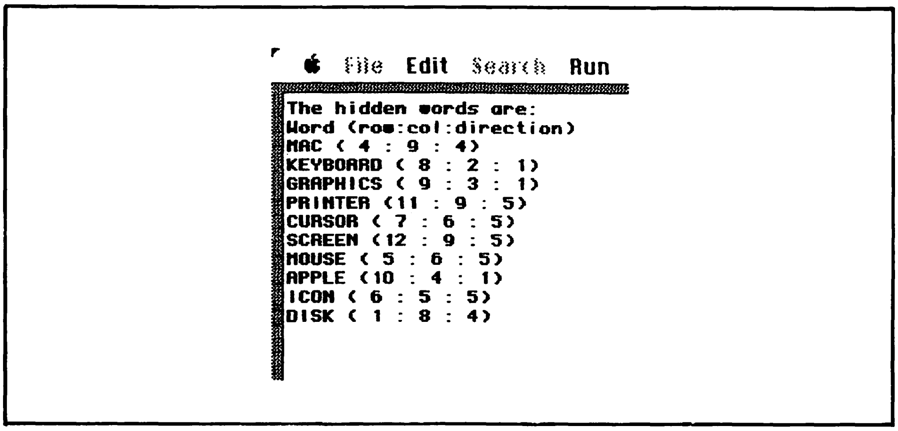
not save both the solution and the puzzle on the Clipboard. Typically, a printed copy of the puzzle solution will be sufficient.

Once the puzzle is saved on the Clipboard, you paste it to the Scrapbook or into MacWrite or MacPaint (the puzzle may be too large for pasting into MacPaint).

Remember, the puzzle pattern won't look right unless the current text font is monospaced. For instance, in the Clipboard, the puzzle will not look right. You must copy the puzzle to Macwrite and reset the font to Monaco 9.

Figures 3-7 and 3-8 show a sample puzzle and puzzle solution using the screen for output. Notice that the solution lists three numbers after each word. The first and second numbers identify the row and column where the word starts. The upper left-hand corner of the grid is row 1, column 1; often this position will be blank in the puzzle shape, but you must still use it as the reference point for interpreting the puzzle solution (see the labeled position in Figure 3-7).

The third number after each word is the path direction. Numbers run from 1 to 8, corresponding to east, southeast, and so forth, in a clockwise rotation. (If you changed the direction count (max.dir%) or direction increments (arrays ri%( ) and ci%( )) from what was supplied in Chapter 2, the path numbers and directions will differ accordingly.)



```
The hidden words are:
Word (row:col:direction)
MAC ( 4 : 9 : 4)
KEYBOARD ( 8 : 2 : 1)
GRAPHICS ( 9 : 3 : 1)
PRINTER (11 : 9 : 5)
CURSOR ( 7 : 6 : 5)
SCREEN (12 : 9 : 5)
MOUSE ( 5 : 6 : 5)
APPLE (10 : 4 : 1)
ICON ( 6 : 5 : 5)
DISK ( 1 : 8 : 4)
```

Figure 3-8. Sample puzzle solution output to the screen

Note: The maximum puzzle size for use with a 128K Macintosh is 20 × 20 (or any size such that rows × columns <= 400).

—Editing Grid Shapes

Editing a grid shape is similar to using the pencil tool in MacPaint. The program provides a window filled with little blocks. Each block represents a letter cell. Using the X-shaped cursor, you selectively erase blocks until your shape is fully defined. Hold down the mouse button to erase; release it to move around the window without erasing. If you erase too many blocks, change the cursor function to FILL and replace the blocks. Again, you hold down the mouse button to fill and release it to move around the window without filling.

You can create free-form designs or more carefully planned pictures. To plan a shape, draw the desired outline on graph paper using no more then 43 squares in any direction. Then fill in all those squares that are halfway or fully inside the outline. For each colored-in graph square, fill in a block on the grid; for each blank graph square, erase a block on the grid. Finally, use this "digitized" shape as your guide for creating the grid shape with the Hidden Words program.

Finally, if your word list is long and the shape is irregular or small, be prepared for a substantial delay while the program attempts to fill-in all the words.

This chapter is adapted from "Hidden Words" by George Stewart, appearing in the December 1983 issue of *Popular Computing* magazine. Copyright 1983 Byte Publications, Inc. Used with the permission of Byte Publications, Inc.

Chapter 4

The Matchmaker

The Matchmaker program enables you to create an endless succession of personalized logic puzzles. You've probably seen this type of puzzle before. Given a list of characters, a list of attributes, and a set of clues, you are to match each character with its attribute. By specifying the two lists, you are able to determine the subject and difficulty of the resultant puzzles.

Figure 4-1 shows a puzzle produced by the Matchmaker program. Before continuing with this chapter, it will be worthwhile for you to try solving the puzzle. Even if you don't succeed, you'll gain some insight into the processes we'll be discussing.

—Overview of Program Logic

Given the two lists (characters and attributes), the program randomly formulates a clue about the various matchups.

Clues can take four forms:

- p implies q
- not p implies q
- p implies not q
- not p implies not q

where p and q are character-attribute pairs.

CLUES

If Sally's grandmother lives in Kansas then Jim lives in Texas

If Sally's grandmother moved from Kansas then Sally lives in Idaho

If Sally moved from Texas then Sally's grandmother moved from Kansas

Match each character with the corresponding attribute:

Character	**Attribute**
Sally	Idaho
Sally's grandmother	Kansas
Jim	Texas

SOLUTION

Sally lives in Idaho

Sally's grandmother lives in Texas

Jim lives in Kansas

Figure 4-1. A sample logic puzzle from the Matchmaker

For example, the proposition "If Sally's grandmother lives in Kansas, then Jim lives in Texas" is an instance of the general form *p implies q*. Recognizing the other forms can be a little harder. For example, the proposition "If Sally moved from Texas, then Sally's grandmother moved from Kansas" is an instance of the general form *not p* (Sally does not live in Texas) *implies not q* (Sally's grandmother does not live in Kansas).

The program cannot take randomly selected matchups and call them clues; it first must verify that the pairings are logically consistent and that they imply a unique solution. To do this, the program uses a truth table showing which matchups are true for each of the possible solutions.

Table 4-1 shows the truth table contents for 3 data pairs. Each column in the table signifies a distinct solution; each row signifies a particular pairing. Note that there are exactly three T's in each column; that's because there are only three valid pairings in each solution.

Table 4-1. Truth table for 3-pair puzzles

Pairs	Solution Number					
	1	2	3	4	5	6
A1	T	T	F	F	F	F
A2	F	F	T	T	F	F
A3	F	F	F	F	T	T
B1	F	F	T	F	T	F
B2	T	F	F	F	F	T
B3	F	T	F	T	F	F
C1	F	F	F	T	F	T
C2	F	T	F	F	T	F
C3	T	F	T	F	F	F

—The Program

The first program block loads the data for the Matchmaker menu:

```
READ last.option%
DIM menu.label$(last.option%)
FOR j%=0 TO last.option%
READ menu.label$(j%)
NEXT j%
DATA 5,Matchmaker
DATA Generate clues,Change data
DATA Print clues, Print solution,Quit
```

The next block sets up certain program constants:

```
LET dg.x%=.1*72
LET dg.y%=.35*72
LET dg.x1%=dg.x%+6*72
LET dg.y1%=dg.y%+4*72
LET true%=(1=1)
LET false%=(1=0)
RANDOMIZE TIMER
DIM t%(1),f%(1),ft%(1),p%(1),q%(1),g%(1)
LET min.pairs%=3
LET max.pairs%=4
DIM ba%(max.pairs%), bp%(max.pairs%), bt%(max.pairs%), a$(max.pairs%,2),
      tf$(2), dev$(2)
LET dev$(1)="SCRN:"
LET dev$(2)="LPT1:DIRECT"
LET pairs%=max.pairs%
FOR pa%=1 TO pairs%
FOR which%=1 TO 2
READ a$(pa%,which%)
NEXT which%,pa%
READ tf$(1),tf$(2)
DATA A,1,B,2,C,3,D,4,is not,is
```

The variable pairs dg.x%,dg.y% and dg.x1%,dg.y1% define the upper left and lower right corners of the window used for dialogs and output. The arrays t$, f%, and so forth are defined in this block for formal reasons only; later on they are erased and redefined to suit the requirements of the puzzle data.

The DATA statements at the end of the block provide initial values for four characters and attributes, as well as verb forms for positive and negative statements.

Generating the First Puzzle

The next block opens the dialog window, prints a title, and generates the first puzzle.

```
WINDOW 1,,(dg.x%,dg.y%)-(dg.x1%,dg.y1%),3
PRINT "THE MATCHMAKER: a logic-puzzle generator."
PRINT
PRINT "Given a series of clues, the object is to"
PRINT "Match each character with one of the attributes."
GOSUB wait.ok
GOSUB make.tables
LET m.state%=1
FOR j%=0 TO last.option%
```

```
MENU 6,j%,m.state%,menu.label$(j%)
NEXT j%
MENU 6,0,0       : REM disable menu
```

The wait.ok subroutine places a button in the lower right corner of the window and waits for you to click on it.

Figure 4-2 shows the title window.

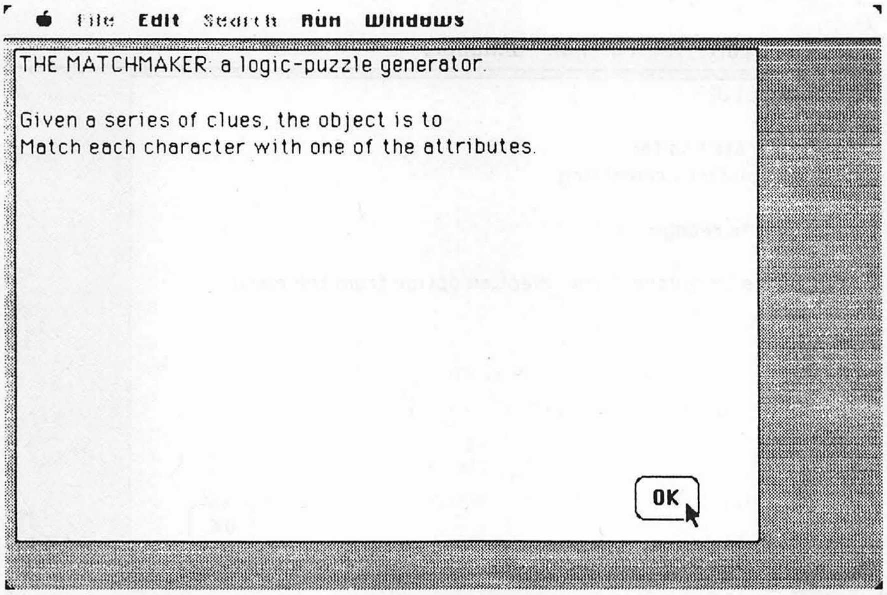
The make.tables subroutine generates the truth tables that are needed during clue generation and then produces a deterministic set of clues about the data.

Figures 4-3 and 4-4 show the screen appearance during the truth-table and clue-generation procedures.

The Menu Loop

The next block of lines lets you select an item from the Matchmaker menu on the menu bar:

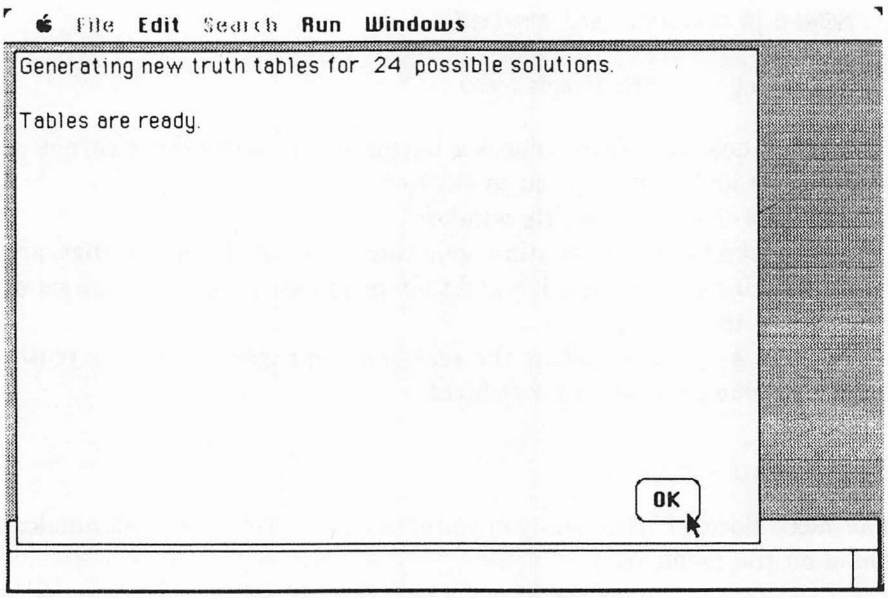
```
get.selection:
MENU 6,0,1       : REM enable menu
WHILE MENU(0)<>6
```



THE MATCHMAKER: a logic-puzzle generator.

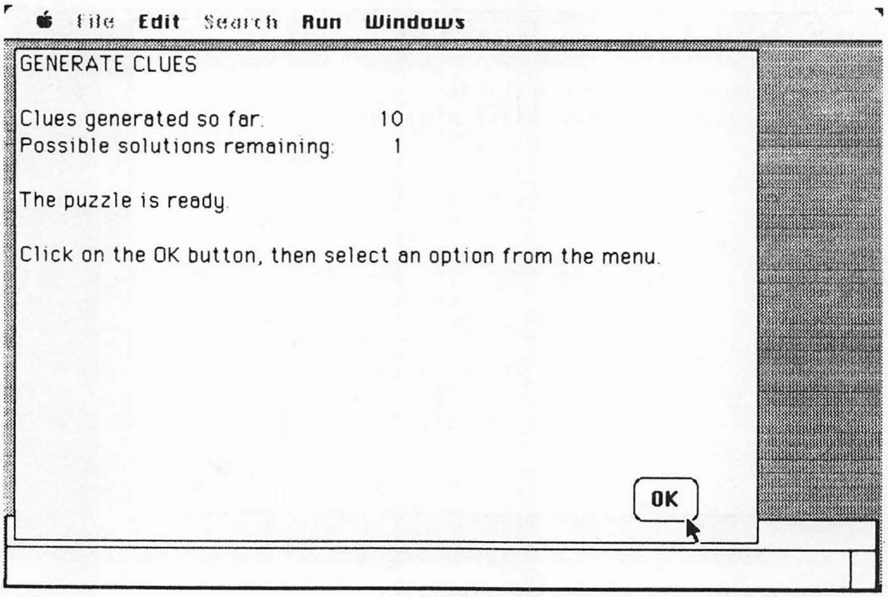
Given a series of clues, the object is to
Match each character with one of the attributes.

Figure 4-2. The initial title screen



```
Generating new truth tables for 24 possible solutions.

Tables are ready.
```

Figure 4-3. Screen during generation of the truth tables



```
GENERATE CLUES

Clues generated so far:              10
Possible solutions remaining:         1

The puzzle is ready.

Click on the OK button, then select an option from the menu.
```

Figure 4-4. Screen at the end of the clue-generation procedure

```
WEND
MENU 6,0,0       : REM disable menu
LET selection%=MENU(1)

ON selection% GOSUB generate.clues,change.data,prt.clues,prt.sol,quit
GOTO get.selection
quit:
WINDOW CLOSE 1
END
```

The program waits in the WHILE/WEND loop until you select a command. The entire menu is then disabled, and the command you selected is executed. The menu is not re-enabled until the program completes the command. Figure 4-5 shows the Matchmaker menu.

The Wait.ok Subroutine

The following lines are used by several program commands as a way of stopping the action so you can read the screen before it is erased:

```
wait.ok:
WINDOW 1
BUTTON 1,2,"OK",(360,250)-(398,276)
WHILE DIALOG(0)<>1
WEND
WINDOW CLOSE 1
RETURN
```

Setting Up the Truth Tables

When you first run the program (also when you change the number of data pairs), the truth tables must be re-defined.

```
make.tables:
WINDOW 1
CLS
ERASE t%,f%,ft%,p%,q%,g%
LET np%=1
FOR j%=1 TO pairs%
LET np%=np%*j%
NEXT j%
PRINT "Generating new truth tables for"; np%; "possible solutions."
```

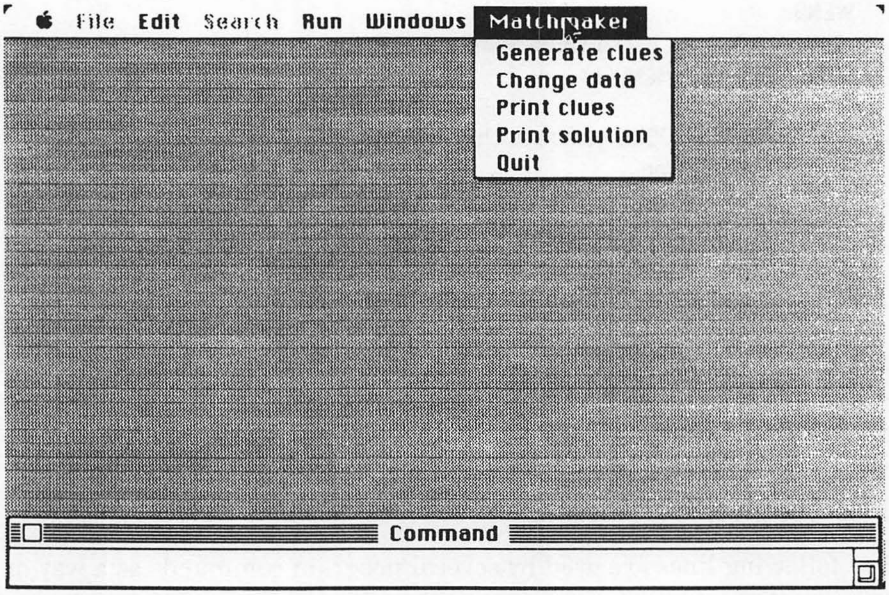


Figure 4-5. The Matchmaker menu

```
LET nc%=pairs%*pairs%
DIM t%(nc%,np%),f%(np%,np%),ft%(np%),p%(np%),q%(np%),g%(np%)
FOR j%=1 TO pairs%
LET bp%(j%)=0
LET bt%(j%)=0
LET ba%(j%)=true%
NEXT j%
```

Array t%(,) is a truth table that specifies all possible solutions for a given number of data pairs. The array f%(,) keeps track of how each clue relates to the list of possible solutions. For instance, f%(2,5) shows whether or not the second clue is consistent with the fifth solution.

Array ft%( ) keeps the same information for a single, tentative clue. When the clue has been checked for consistency with all preceding clues, the information is copied into f%(,). The arrays p%( ), g%( ), and q%( ) keep track of the matchups used in each clue.

Np% is the number of possible solutions (possible matchups) for a given number of data pairs. The variable nc% contains the number of possible combinations of items from the character and attribute lists.

Generating Possible Solutions

The next lines begin the process of generating all n! possible solutions.

```
start.perms:
LET l%=1   : REM tree-level counter
LET p%=0   : REM permutation counter
move.ptr:
LET bp%(l%)=bp%(l%)MOD pairs% + 1
IF ba%(bp%(l%))=false% THEN move.ptr
LET bt%(l%)=bt%(l%)+1
LET ba%(bp%(l%))=false%
IF l%=pairs% THEN reached.end
LET l%=l%+1
GOTO move.ptr
```

The program generates solutions by spanning a tree as shown in Figure 4-6.

Each pathway from the tree trunk to an endpoint corresponds to one

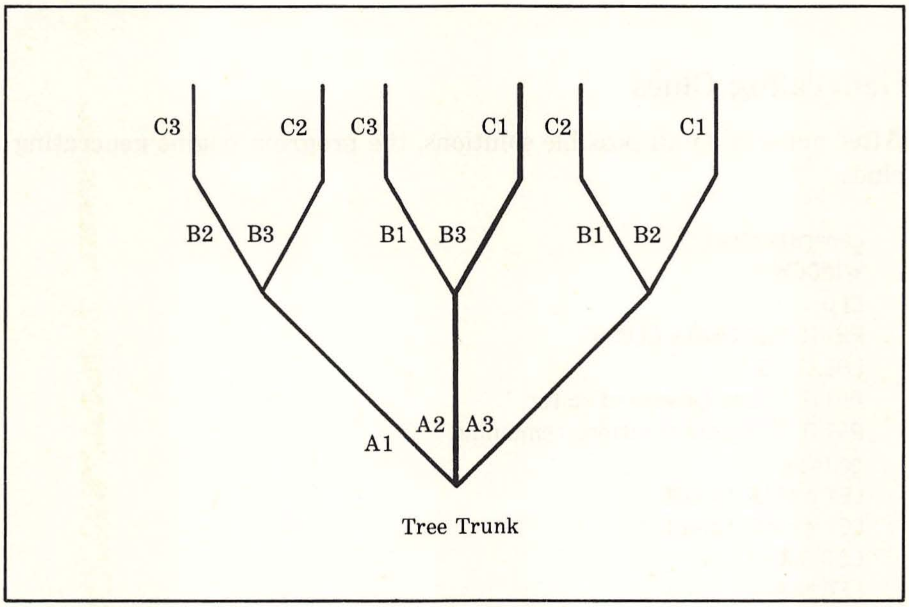


Figure 4-6. A tree diagram showing all 24 possible solutions to a 3-pair puzzle. Each path represents one solution

possible solution to the puzzle. In Figure 4-6, the highlighted path corresponds to the pairing A is 2, B is 3, C is 1.

When the program reaches an endpoint of the tree, the following lines record the pairings defined by the latest pathway:

```
reached.end:
LET p%=p%+1
FOR j%=1 TO pairs%
LET t%((j%-1)*pairs%+bp%(j%),p%)=true%
NEXT j%
back.up:
LET ba%(bp%(l%))=true%
LET bt%(l%)=0
LET l%=l%-1
IF l%=0 THEN mt.done
IF bt%(l%)=pairs%-l%+1 THEN back.up
LET ba%(bp%(l%))=true%
GOTO move.ptr
mt.done:
PRINT
PRINT "Tables are ready."
GOSUB wait.ok
```

Generating Clues

After generating all possible solutions, the program begins generating clues:

```
generate.clues:
WINDOW 1
CLS
PRINT "GENERATE CLUES"
LOCATE 3,1
PRINT "Clues generated so far: "
PRINT "Possible solutions remaining:"
gc.loop:
LET p.old%=false%
LET q.old%=false%
LET fc%=0
LET pn%=0
LOCATE 3,27
PRINT USING "**";pn%
```

```
LOCATE 4,27
PRINT USING "**";np%-fc%
FOR j%=1 TO np%
LET ft%(j%)=true%
NEXT j%
```

P.old% and q.old% are the most recent pairings; the program keeps this information to avoid giving repetitious clues. Pn% is the number of clues generated so far.

The following block randomly selects a proposition:

```
pick.pq:
LET p%=INT(RND*nc%)+1
LET q%=INT(RND*nc%)+1
LET p1%=(p%-1)\pairs%+1
LET p2%=p%-(p1%-1)*pairs%
LET q1%=(q%-1)\pairs%+1
LET q2%=q%-(q1%-1)*pairs%
IF (p1%=q1%) OR (p2%=q2%) THEN pick.pq
pick.g:
LET g%=INT(RND*np%)+1
LET pv%=t%(p%,g%)
LET qv%=t%(q%,g%)
IF (pv%=p.old%) AND (qv%=q.old%) THEN pick.g
LET pn%=pn%+1
LET g%(pn%)=g%
LET p%(pn%)=p%
LET q%(pn%)=q%
LET j%=pn%
```

The randomly chosen variables p%, q%, and g% uniquely define a proposition. For example, suppose p%, q%, and g% have the values 3, 4, and 2. Referring to Table 4-1, p%=3 refers to pairing A3; q%=4 refers to pairing B1; and g%=2 refers to solution 2. Under solution 2, A3 is false and B1 is false. Thus the proposition is "not A3 implies not B1."

The next group of lines determines the efficacy of the latest clue on the logical system of clues.

```
FOR j%=1 TO np%
LET pt%=t%(p%,j%)
LET qt%=t%(q%,j%)
LET f%(pn%,j%)=(pv%=pt%) IMP (qv%=qt%)
```

```
NEXT j%
LET fa%=0
FOR j%=1 TO np%
IF ft%(j%)=true% AND f%(pn%,j%)=false% THEN fa%=fa%+1
NEXT j%
IF fa%>0 THEN effective.clue
LET pn%=pn%-1
GOTO pick.pq
```

The variable fa% counts the number of possible solutions ruled out by the latest clue. If fa%>0, the clue is effective; otherwise, it is syllogistic (it rules out no new solutions at all). In the latter case, the program discards the clue and gets another.

In the case of effective clues, the program compares the new clue with the preceding clues to see if it is redundant (ruling out solutions that have already been ruled out) or too exclusive (ruling out all remaining solutions).

The following lines make the comparison:

```
effective.clue:
IF fa%+fc%>=np% THEN gc.loop    : REM insoluble, try again
LET fc%=fa%+fc%
FOR j%=1 TO np%
LET ft%(j%)=f%(pn%,j%) AND ft%(j%)
NEXT j%
LOCATE 3,27
PRINT USING "##";pn%
LOCATE 4,27
PRINT USING "##";np%-fc%
IF fc%=np%-1 THEN enough.clues
LET p.old%=pv%
LET q.old%=qv%
GOTO pick.pq
enough.clues:
PRINT
PRINT "The puzzle is ready."
PRINT
PRINT "Click on the OK button, then select an option from the menu."
GOTO wait.ok
```

The variable fc% counts the number of solutions that have been ruled out. When fc% is one less than the total number of solutions, the clues imply a unique solution, so the puzzle is ready. Otherwise the program goes back to generate another clue (GOTO pick.pq).

Figure 4-4 shows the screen appearance at the end of the clue selection process.

Changing Clue Data

Initially the characters are A, B, C, and D, and the attributes are 1, 2, 3, and 4. The next block allows you to change the data used in formulating the actual clues:

```
change.data:
WINDOW 1
CLS
PRINT "CHANGE PUZZLE DATA"
BUTTON 1,5-pairs%,"3 pairs",(18,32)-(90,47),3
BUTTON 2,pairs%-2,"4 pairs",(126,32)-(198,47),3
BUTTON 3,1,"OK",(234,26)-(306,53),1
LET new.pairs%=pairs%
set.pairnum:
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
IF btn%=3 THEN pairnum.ok
LET new.pairs%=btn%+2
BUTTON 1,5-new.pairs%
BUTTON 2,new.pairs%-2
GOTO set.pairnum
```

Your data can consist of three or four data pairs. Puzzles based on four data pairs are considerably harder to solve than those based on three data pairs.

Figure 4-7 shows the dialog window for specifying the number of data pairs.

The next block displays the data presently in use:

```
pairnum.ok:
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON CLOSE 3
LET changed.pairnum%=(pairs%<>new.pairs%)
```

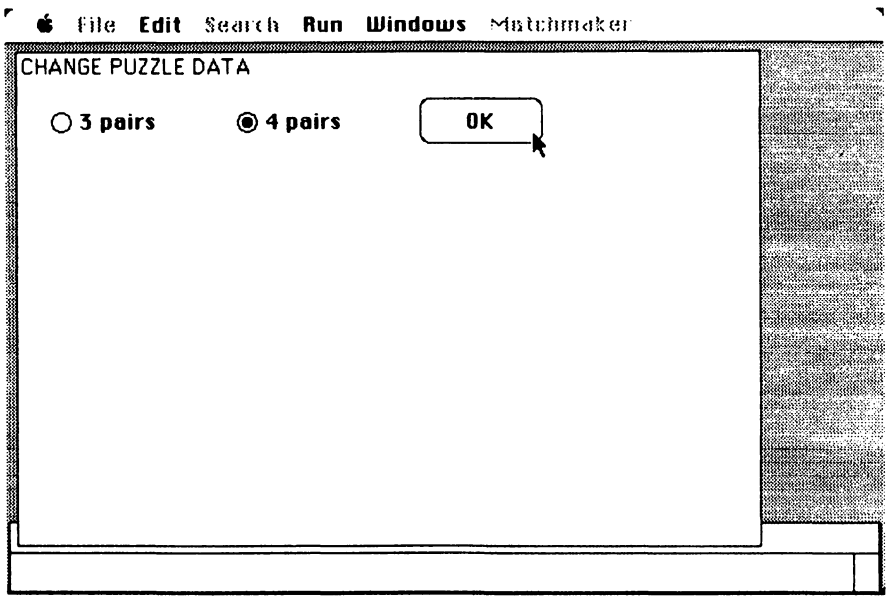
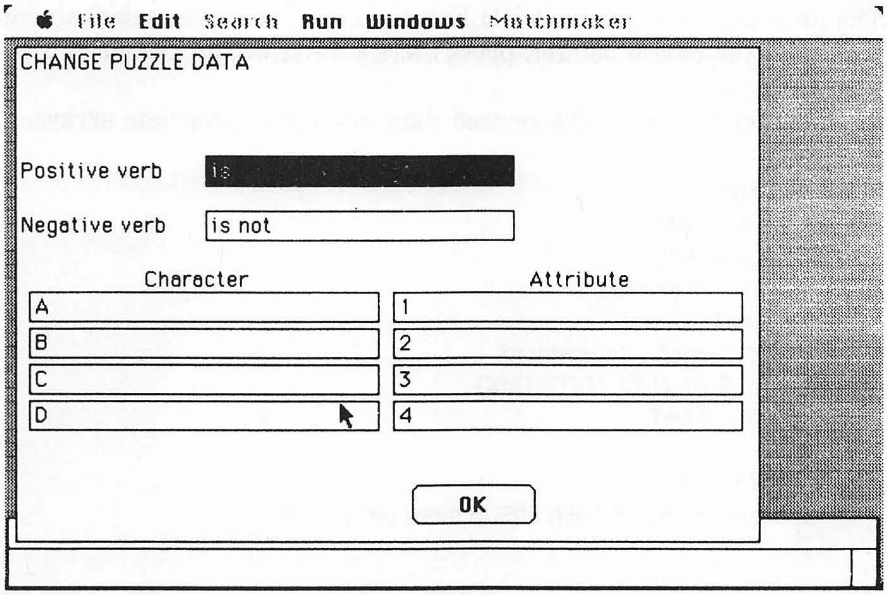


Figure 4-7. Dialog box for selecting the number of data pairs

```
LET pairs%=new.pairs%
LET n.fields%=pairs%*2+2
LOCATE 5,1
PRINT "Positive verb"
PRINT
PRINT "Negative verb"
PRINT
PRINT TAB(10)"Character"; TAB(38); "Attribute"
FOR pa%=1 TO pairs%
FOR which%=1 TO 2
LET fld%=2+(pa%-1)*2+which%
LET fld.x%=(which%-1)*212+9
LET fld.y%=(pa%-1)*21+9*16
EDIT FIELD fld%, a$(pa%,which%), (fld.x%,fld.y%)- (fld.x%+200,fld.y%+15), 1
NEXT which%,pa%
EDIT FIELD 2,tf$(1),(112,96)-(288,111)
EDIT FIELD 1,tf$(2),(112,64)-(288,79)
BUTTON 1,1,"OK",(230,dg.y1%-58)-(302,dg.y1%-35)
```



CHANGE PUZZLE DATA

Positive verb	is
Negative verb	is not

Character	Attribute
A	1
B	2
C	3
D	4

Figure 4-8. The data entry and editing screen

The program lets you move from one edit field until another until you have filled them all and pressed the OK button. Figure 4-8 shows the dialog window that lets you change the puzzle data.

The following lines wait for you to press the OK button, signifying that you have finished editing or changing the data:

```
LET fld%=1
cd.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=1 THEN check.data
IF event%=2 THEN fld%=DIALOG(2)
IF event%=6 OR event%=7 THEN fld%=fld% MOD (n.fields%)+1
EDIT FIELD fld%
GOTO cd.loop
```

The program waits in the WHILE/WEND loop until you select an edit field with the mouse pointer, press ENTER, RETURN, or TAB, or click on the OK button.

The next lines copy the revised data into the appropriate arrays.

```
check.data:
LET tf$(1)=EDIT$(2)
LET tf$(2)=EDIT$(1)
FOR pa%=1 TO pairs%
FOR which%=1 TO 2
LET fld%=2+(pa%-1)*2+which%
LET a$(pa%,which%)=EDIT$(fld%)
NEXT which%,pa%
cd.done:
WINDOW CLOSE 1
IF changed.pairnum% THEN GOSUB make.tables
RETURN
```

If you have changed the number of data pairs from the previous setting, the program must regenerate its truth tables (IF changed.pairnum%...). Otherwise the program simply returns control to the menu procedure (get.selection). This allows you to keep the same set of formal clues but change the words that are plugged into the formal structure.

Printing the Puzzle

Here are the lines that print the clues.

```
prt.clues:
WINDOW 1
CLS
PRINT "PRINT CLUES"
GOSUB select.device
PRINT#1, "CLUES"
PRINT#1,
FOR j%=1 TO pn%
LET g%=g%(j%)
LET p%=p%(j%)
LET q%=q%(j%)
LET pv%=t%(p%,g%)
LET qv%=t%(q%,g%)
LET p1%=(p%-1)\pairs%+1
LET p2%=p%-(p1%-1)*pairs%
LET q1%=(q%-1)\pairs%+1
LET q2%=q%-(q1%-1)*pairs%
PRINT#1, "If ";    a$(p1%,1); " "; tf$(ABS(pv%)+1); " "; a$(p2%,2);
PRINT#1, " then "; a$(q1%,1); " "; tf$(ABS(qv%)+1); " "; a$(q2%,2)
NEXT j%
PRINT #1,
PRINT#1, "Match each character with the corresponding attribute."
CLOSE 1
GOTO wait.ok
```

Before printing, the program prompts you to select the output device. Note that SCREEN is the Macintosh screen and PRINTER is the Imagewriter or other printer connected to the Macintosh's printer connector.

Figure 4-9 shows the screen that allows you to select the output device for the puzzle clues. Figure 4-10 shows the clues output to the screen.

Here are the lines that print the puzzle's solution:

```
prt.sol:
WINDOW 1
CLS
PRINT "PRINT SOLUTION"
GOSUB select.device
PRINT#1, "SOLUTION"
PRINT#1,
LET x%=1
WHILE ft%(x%)=false%
```

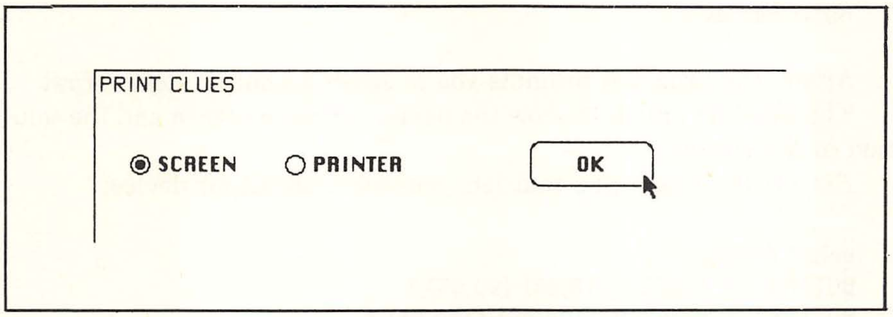
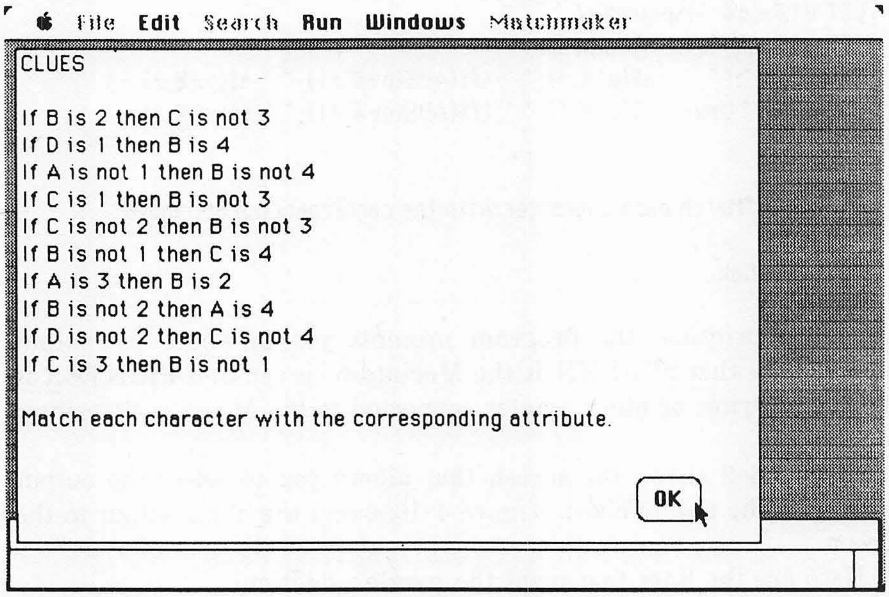

PRINT CLUES

Figure 4-9. Device selection for printing clues



```
CLUES

If B is 2 then C is not 3
If D is 1 then B is 4
If A is not 1 then B is not 4
If C is 1 then B is not 3
If C is not 2 then B is not 3
If B is not 1 then C is 4
If A is 3 then B is 2
If B is not 2 then A is 4
If D is not 2 then C is not 4
If C is 3 then B is not 1

Match each character with the corresponding attribute.
```

Figure 4-10. Clues output to the screen

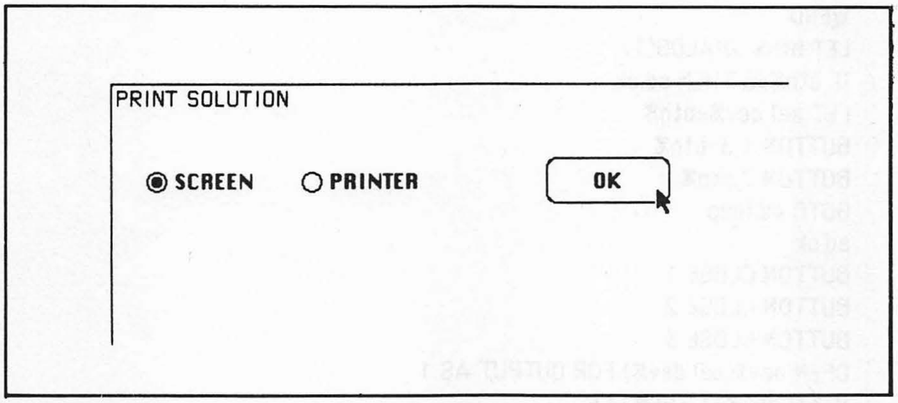
```
LET x%=x%+1
WEND
FOR pa%=1 TO nc%
IF t%(pa%,x%)=false% THEN next.pair
LET p1%=(pa%-1)\pairs%+1
LET p2%=pa%-(p1%-1)*pairs%
PRINT#1, a$(p1%,1);" "; tf$(2); " "; a$(p2%,2)
next.pair:
NEXT pa%
CLOSE 1
GOTO wait.ok
```

Again, the program prompts you to select an output device first.

Figures 4-11 and 4-12 show the device selection screen and the solution to the puzzle.

Here is the subroutine that lets you select an output device:

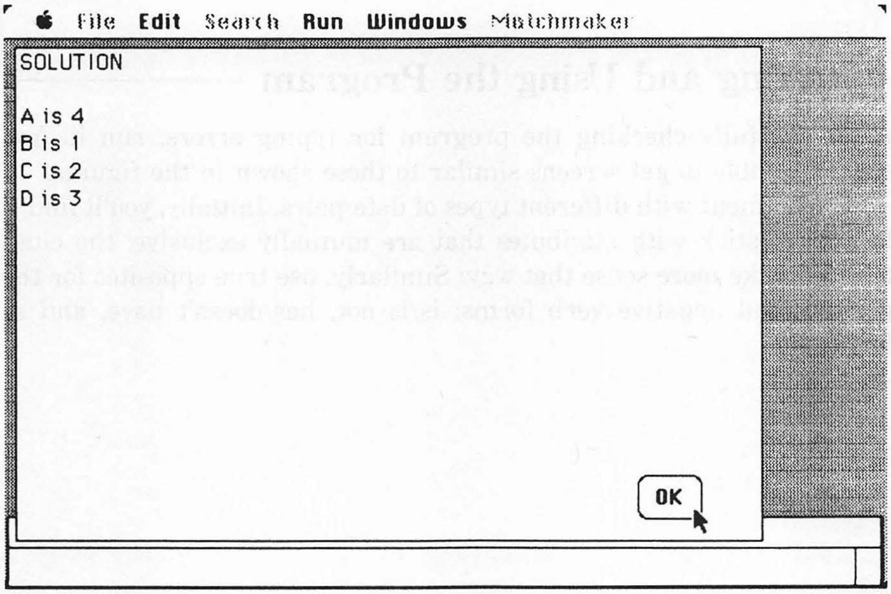
```
select.device:
BUTTON 1,2,"SCREEN",(18,48)-(90,63),3
BUTTON 2,1,"PRINTER",(108,48)-(180,63),3
```



PRINT SOLUTION

[✓ SCREEN] PRINTER

Figure 4-11. Device selection for printing the solution



SOLUTION

A is 4
B is 1
C is 2
D is 3

Figure 4-12. Solution output to the screen

```
BUTTON 3,1,"OK",(252,42)-(324,69),1
LET sel.dev%=1       : REM Initial device setting
sd.loop:
WHILE DIALOG(0)<>1
```

```
WEND
LET btn%=DIALOG(1)
IF btn%=3 THEN sd.ok
LET sel.dev%=btn%
BUTTON 1,3-btn%
BUTTON 2,btn%
GOTO sd.loop
sd.ok:
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON CLOSE 3
OPEN dev$(sel.dev%) FOR OUTPUT AS 1
IF sel.dev%=1 THEN CLS
RETURN
```

—Testing and Using the Program

After carefully checking the program for typing errors, run it. You should be able to get screens similar to those shown in the figures.

Experiment with different types of data pairs. Initially, you'll find it helpful to stick with attributes that are mutually exclusive; the clues tend to make more sense that way. Similarly, use true opposites for the positive and negative verb forms: is/is not, has/doesn't have, and so forth.

Chapter 5

Crossword Puzzle Patterns

This program generates ready-to-use crossword puzzle patterns ranging in size from 3 × 3 to 12 × 12. You determine the approximate difficulty level of the puzzle by specifying the shortest allowable word path (2, 3, 4, or 5 letters).

As each puzzle is generated, it is displayed on the screen. If you like a particular puzzle pattern, you may print it on the Imagewriter and save the entire screen in a MacPaint file. Later, you can use MacPaint to erase extraneous graphics (everything but the puzzle).

The program does not place words into the puzzle pattern. Using the paper copy, you manually insert the words you want to use. When you have completely filled all the puzzle paths, you use MacWrite to prepare a list of word clues. A completed puzzle consists of a high-quality MacPaint printout of the puzzle, plus your list of clues for words across and down.

—Anatomy of a Crossword Puzzle

A lot of care goes into the creation of a puzzle pattern. Good puzzle patterns have the following properties:

- Solid blocks are arranged in symmetric, geometric, or representational patterns.
- Every possible word path is numbered.
- Only one set of numbers is used for paths across and down.

Puzzles generated by this program satisfy all three criteria. Refer to Figure 5-1 while reading the following explanation of the puzzle-creation process.

The program starts out with an empty square puzzle grid. It then randomly fills in a certain number of the cells in quadrant I of the grid. Each time it fills in a cell in quadrant I, the corresponding cells in quadrants II through IV are also filled in. Quadrant II is equivalent to quadrant I rotated 90 degrees clockwise; quadrant III is equivalent to quadrant I rotated 180 degrees; quadrant IV is equivalent to quadrant I rotated 270 degrees.

The result of this process is a radially symmetrical pattern of empty and filled cells, such as the pattern in Figure 5-1 without the path numbers. Notice that by rotating quadrant I 90 degrees at a time, you duplicate the other three quadrants. Also note that the centerpoint of the puzzle is considered to belong to all four quadrants. Only puzzles in which the length of a side is an odd number have such a cell at the center.

Next the program must locate all the word paths. To explain this process, we need to present a couple of terms. A *head cell* is the starting location of a word path: the numbered squares of a crossword puzzle are head cells. A *block cell* is a filled-in cell that marks the boundary of a word path.

The set of potential head cells in a puzzle pattern consists of those empty cells immediately below or to the right of block cells. Note that the puzzle is surrounded by an imaginary boundary of block cells so that empty cells in the top row and left column are also potential head cells. Potential head cells are used only if the resultant path would be long enough to meet the minimum word length. (There is no maximum word length; subject to chance, word paths may occasionally go completely across or down the puzzle grid.)

Notice in Figure 5-1 that some of the numbered cells define paths across and down, while others define paths in one direction only.

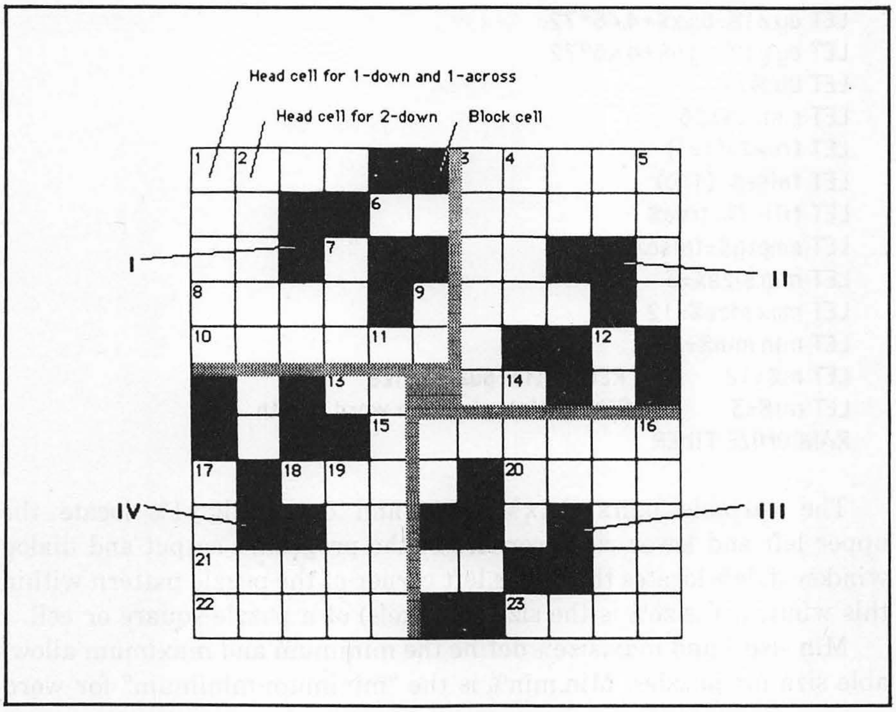


Figure 5-1. Anatomy of a crossword puzzle pattern. Note that the cell in the center belongs to all four quadrants

—The Program

The first block loads the data for the Crossword menu.

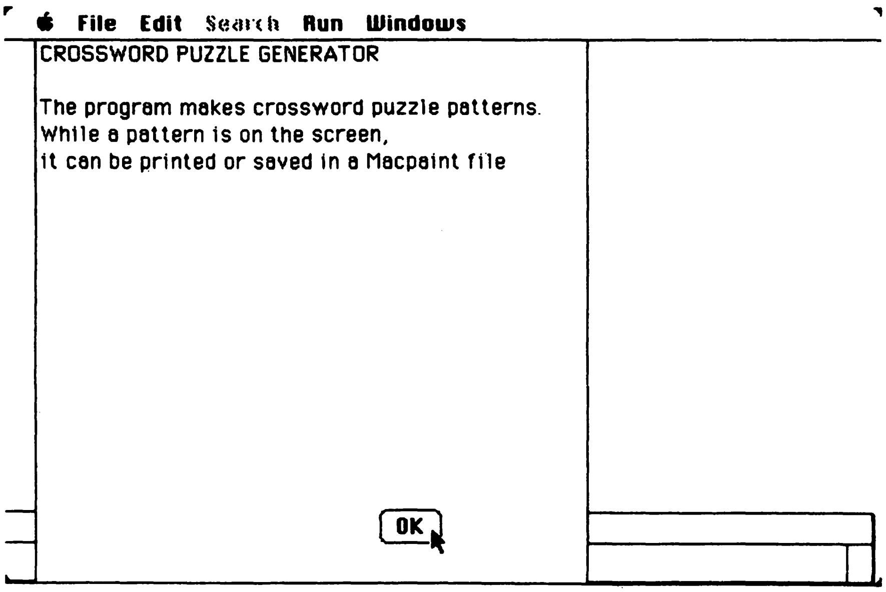
```
READ last.option%
DIM menu.label$(last.option%)
FOR j%=0 TO last.option%
READ menu.label$(J%)
NEXT j%
DATA 5,Crossword
DATA New puzzle,Print screen,Redraw screen,Save screen,Quit
```

The next block defines constants and initializes parameters.

```
LET dg.x%=.25*72
LET dg.y%=.275*72
LET dg.x1%=dg.x%+4.46*72
LET dg.y1%=dg.y%+4.46*72
LET ulc%=3
LET c.size%=26
LET true%=(1=1)
LET false%=(1=0)
LET filled%=true%
LET empty%=false%
LET min.size%=3
LET max.size%=12
LET min.min%=2
LET m%=12          : REM initial puzzle size
LET ml%=3          : REM initial minimum word length
RANDOMIZE TIMER
```

The variable pairs dg.x%,dg.y% and dg.x1%,dg.y1% locate the upper left and lower right corners of the program's output and dialog window. Ulc% locates the upper left corner of the puzzle pattern within this window. C.size% is the size (in pixels) of a puzzle square or cell.

Min.size% and max.size% define the minimum and maximum allowable size for puzzles. Min.min% is the "minimum-minimum" for word



```
CROSSWORD PUZZLE GENERATOR

The program makes crossword puzzle patterns.
While a pattern is on the screen,
it can be printed or saved in a Macpaint file
```

Figure 5-2. The title window

lengths; in other words, it is the smallest value you can specify as a minimum path length.

The next block creates the title window shown in Figure 5-2:

```
WINDOW 1,,(dg.x%,dg.y%)-(dg.x1%,dg.y1%),3
CALL TEXTSIZE(12)
CALL TEXTFACE(0)
PRINT "CROSSWORD PUZZLE GENERATOR"
PRINT
PRINT "The program makes crossword puzzle patterns."
PRINT "While a pattern is on the screen,"
PRINT "it can be printed or saved in a Macpaint file."
BUTTON 1,2,"OK",(200,278)-(236,298)
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
BUTTON CLOSE 1
```

The following lines prompt you to create the first puzzle pattern:

```
GOSUB resize.puzzle
GOSUB new.puzzle
LET m.state%=1
FOR j%=0 TO last.option%
MENU 6,j%,m.state%,menu.label$(j%)
NEXT j%
MENU 6,0,0
```

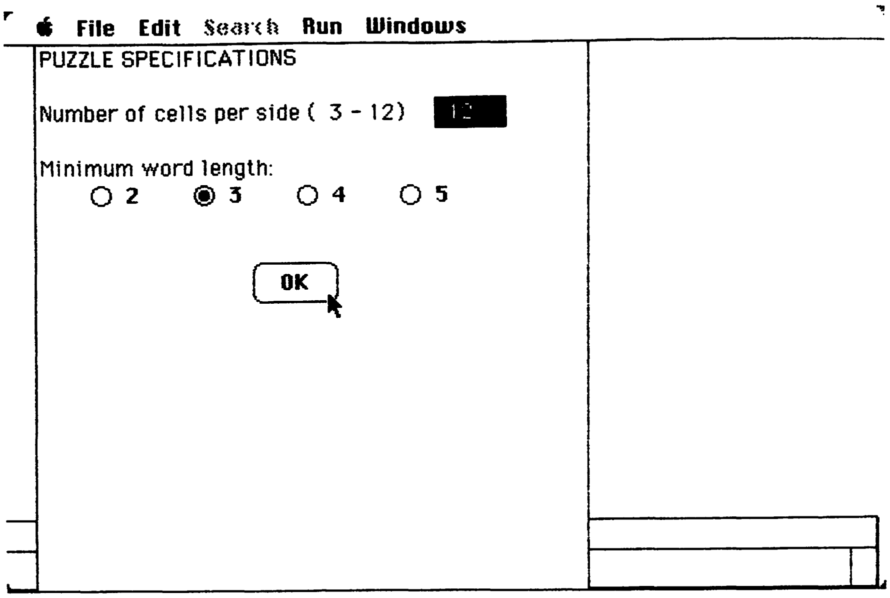
Figures 5-3 through 5-5 depict the screens you'll see when you first run the program (after you've typed it all in).

After creating the new puzzle, the program turns on the Crossword menu, pictured in Figure 5-5.

The Menu Selection Loop

Once a puzzle has been created, the program enters a menu selection loop that controls all activity until you quit the program:

```
get.selection:
MENU 6,0,1
WHILE MENU(0)<>6
WEND
```

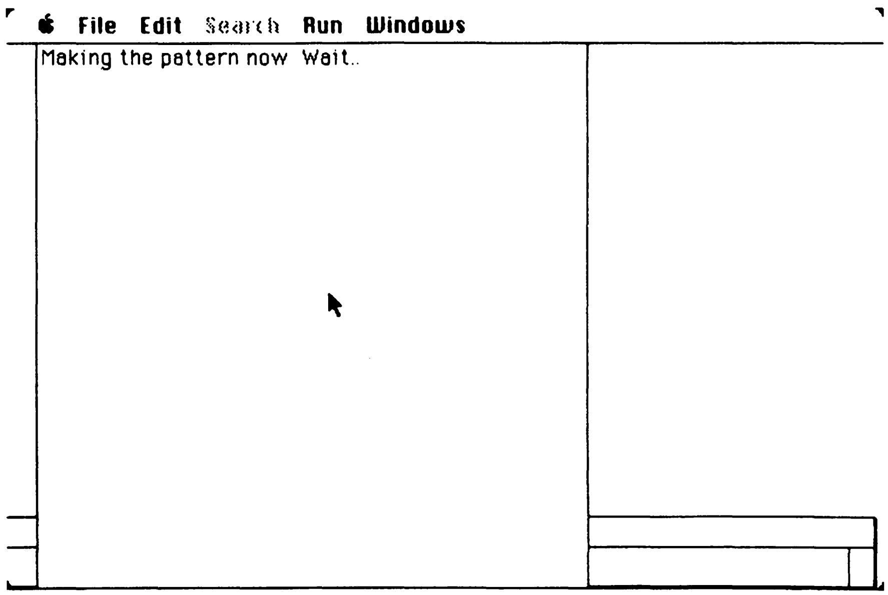


PUZZLE SPECIFICATIONS

Number of cells per side (3 - 12) 12

Minimum word length:
○ 2 [✓ 3] ○ 4 ○ 5

Figure 5-3. The puzzle specifications dialog window



Making the pattern now Wait..

Figure 5-4. The puzzle-creation "wait" window

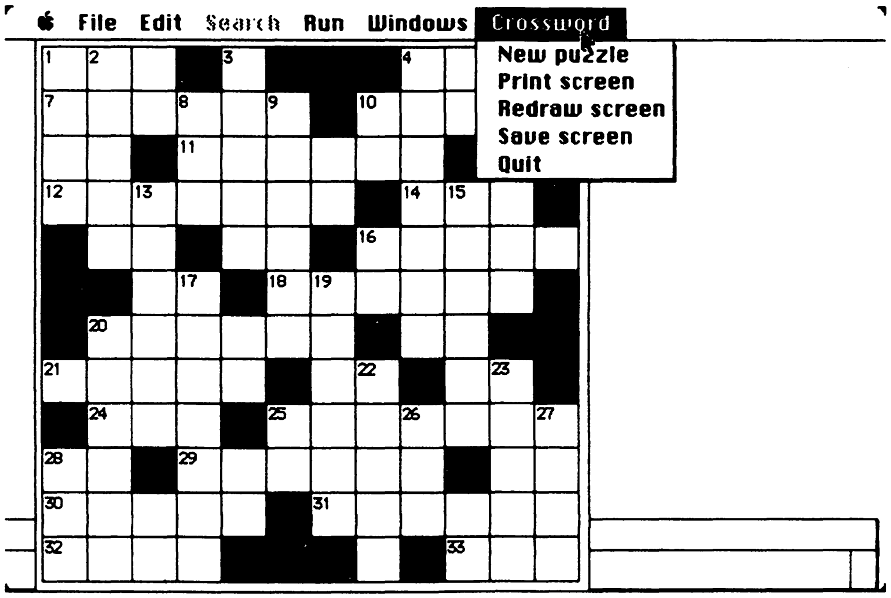


Figure 5-5. A completed puzzle showing the Crossword menu

```
MENU 6,0,0
LET selection%=MENU(1)
ON selection% GOSUB new.puzzle, prt.screen, prt.pzl,
      save.screen, quit
GOTO get.selection
quit:
END
```

The rest of the program consists of major and auxiliary subroutines to accomplish the five menu options (make a new puzzle, print the screen to the Imagewriter, redraw the puzzle, save the screen in a MacPaint file, and quit.

Resizing the Puzzle

Each time you change the puzzle size, the following lines make the necessary changes in the arrays and other parameters:

```
resize.puzzle:
LET n%=m%\2
LET odd%=(n%*2<>m%)
LET nc%=m%*m%
LET nb%=nc%\6+1
DIM m%(m%+1,m%+1),pl%(m%,m%),r%(nc%),c%(nc%)
RETURN
```

Variable n% is the number of cells in a quadrant. If the puzzle size, m%, is an odd number, n% is actually one less than the true quadrant size. The variable odd% takes care of this anomaly; for odd m%, odd%=−1; for even m%, odd%=0.

Variable nc% is the total number of cells in the puzzle, and nb% is the number of block cells to be marked during the pattern design phase.

Making a New Puzzle

The next lines generate a new puzzle (New puzzle command):

```
new.puzzle:
GOSUB dialogue.size
IF changed.size%=false% THEN size.ok
ERASE m%,pl%,r%,c%
GOSUB resize.puzzle
size.ok:
GOSUB pattern
GOSUB prt.pzl
RETURN
```

If you change the puzzle size during the puzzle specification dialog, the program calls the resize puzzle subroutine before continuing. Once a new pattern is generated, the result is displayed on the screen.

Puzzle Specification Dialog

The following lines let you specify the puzzle size and the minimum word length:

```
dialogue.size:
WINDOW 1
LET new.m%=m%
CLS
PRINT "PUZZLE SPECIFICATIONS"
```

```
PRINT
PRINT USING "Number of cells per side (** - **)";min.size%,
      max.size%
EDIT FIELD 1,STR$(new.m%),(234,32)-(272,47)
LOCATE 5,1
PRINT "Minimum word length:"
FOR b%=1 TO 4
BUTTON b%,0,STR$(b%+1), (30+(b%-1)*60,80)-
      (60+(b%-1)*60,96),3
NEXT b%
BUTTON 5,0,"OK",(126,128)-(176,152)
LET btn%=0
```

Refer to Figure 5-3 while studying these lines. The program sets up one edit field for specifying the size. Then it creates four radio-type buttons for specifying the minimum word length. Finally, it creates an OK button for use when you have completed the specifications. Initially, all the buttons are inactive.

The next block of lines checks the current size settings.

```
change.size:
new.m%=VAL(EDIT$(1))
IF new.m%<min.size% OR new.m%>max.size% THEN size.error
LET max.min%=new.m%\3+1
FOR B%=1 TO 4
BUTTON b%,ABS((b%+1)<=max.min%)
NEXT b%
IF ml%>max.min% THEN ml%=max.min%
BUTTON ml%-1,2
BUTTON 5,1
IF btn%=5 THEN end.dialogue
```

If the size specification is out of range, the size.error routine is executed. Otherwise, the size setting is used to determine an upper limit for the minimum word length, referred to as the "maximum-minimum" or max.min%. The maximum-minimum equals the puzzle size divided by 3, rounded up to the nearest integer.

Those buttons corresponding to paths up to the maximum-minimum are made active (BUTTON b%, ABS...). If the current maximum-minimum setting, ml%, exceeds the newly calculated value, the current setting is reset to the new value (IF ml%>max.min%...).

The following lines constitute a loop that is repeated until you change the puzzle size (edit field) or select one of the buttons:

```
size.loop:
LET btn%=0
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=2 OR event%=6 THEN change.size
IF event%=1 THEN which.button
GOTO size.loop
size.error:
BUTTON 5,0
BEEP
GOTO size.loop
```

When you select a button, the following lines are executed:

```
which.button:
LET btn%=DIALOG(1)
IF btn%=5 THEN change.size
change.ml:
BUTTON ml%-1,1
BUTTON btn%,2
LET ml%=btn%+1
GOTO size.loop
end.dialogue:
LET changed.size%=(new.m%<>m%)
LET m%=new.m%
EDIT FIELD CLOSE 1
FOR b%=1 TO 5
BUTTON CLOSE b%
NEXT b%
CLS
RETURN
```

If you press button 5 (the OK button), the program goes back to check the current size setting. If the size is within range, the last line of the change.size routine will force the program to skip to the end.dialogue routine. Otherwise, change.size will wait for you to enter a valid size.

The change.ml routine resets the minimum word length and updates the button display accordingly.

The end.dialogue routine records whether the puzzle size was changed (changed.size%), closes the edit field and closes the buttons.

That ends the dialogue.size subroutine.

Creating a New Pattern

Now we present the subroutine that creates a new puzzle pattern. The first block of lines marks the randomly chosen block cells:

```
pattern:
PRINT "Making the pattern now. Wait..."
FOR j%=0 TO m%+1
FOR k%=0 TO m%+1
LET m%(j%,k%)=(j%=0) OR (j%=m%+1) OR (k%=0) OR (k%=m%+1)
IF m%(j%,k%)=empty% THEN LET pl%(j%,k%)=empty%
NEXT k%,j%
FOR j%=1 TO nb%\4+1
pick.cell:
LET r%=INT(RND*(n%-odd%))+1
LET c%=INT(RND*(n%-odd%))+1
IF m%(r%,c%)=filled% THEN pick.cell
GOSUB mark.4
NEXT j%
```

The first FOR-NEXT loop marks a perimeter of block cells around the entire puzzle. These block cells are never displayed; however, they are needed so the program can identify path boundaries along the outer edges of the puzzle grid. The second FOR-NEXT loop randomly selects a cell in quadrant I. The mark.4 subroutine marks that cell and its counterparts in quadrants II, III, and IV.

After the block cells have been marked, the following lines eliminate head cells that do not define at least one path that meets the minimum length requirement.

```
pass.2:
LET chg.cell%=false%
FOR r%=1 TO n%-odd%
FOR c%=1 TO n%-odd%
IF m%(r%,c%)=filled% THEN next.cell
LET hs%=c%-1
WHILE m%(r%,hs%)=empty%
LET hs%=hs%-1
WEND
LET he%=c%+1
WHILE m%(r%,he%)=empty%
LET he%=he%+1
WEND
LET vs%=r%-1
WHILE m%(vs%,c%)=empty%
LET vs%=vs%-1
WEND
LET ve%=r%+1
WHILE m%(ve%,c%)=empty%
LET ve%=ve%+1
WEND
IF he%-hs%>ml% OR ve%-vs%>ml% THEN next.cell
LET chg.cell%=chg.cell% OR ((he%-hs%<=ml%) AND
      (ve%-vs%<=ml%))
IF chg.cell% THEN GOSUB mark.4
next.cell:
NEXT c%,r%
IF chg.cell% THEN pass.2
```

For each empty cell in quadrant I, the program locates the head cell that “owns” that cell. First the program backs up horizontally until reaching a block cell. Then it advances until it reaches another block cell. The distance from one boundary to another is the effective path length. The process is repeated in the vertical direction.

If at least one path (horizonal or vertical) is as long as the minimum path length ml%, the program examines the next cell. If neither path is long enough, the program makes the current cell a block cell and sets a flag (chg.cell%) to record this change.

After all the cells in quadrant I have been examined, the program checks to see whether any changes were made during the refinement process; if changes were made, the refinement process must be repeated, because adding a block cell may have caused other paths to become too short.

Numbering the Paths

Now the program assigns numbers to all the head cells:

```
LET pn%=0
FOR r%=1 TO m%
FOR c%=1 TO m%
IF m%(r%,c%)=filled% THEN another.cell
IF m%(r%-1,c%)=empty% THEN h.path
LET ve%=r%+1
WHILE m%(ve%,c%)=empty%
LET ve%=ve%+1
WEND
IF ve%-r%<ml% THEN h.path
GOSUB path.info
```

Variable pn% counts the number of head cells found. The cell at row r%, column c% is by definition a head cell if it is empty and the cell above it is filled (m%(r%−1,c%)=filled%). The program first determines whether the vertical path starting at that head cell is long enough. If it is, the path location is recorded by the path.info subroutine.

Next the program checks for a horizontal path starting from the current head cell.

```
h.path:
IF m%(r%,c%-1)=empty% THEN another.cell
LET he%=c%+1
WHILE m%(r%,he%)=empty%
LET he%=he%+1
WEND
IF he%-c%<ml% THEN another.cell
IF pl%(r%,c%)=0 THEN GOSUB path.info
another.cell:
NEXT c%,r%
RETURN
```

In this case, the current cell is a head cell if the cell to the left is filled (m%(r%,c%−1)=filled%). If the cell does define a horizontal path, the program determines if that path is long enough. If it is, the horizontal path is recorded by the path.info subroutine.

When every cell has been checked in this manner, the pattern creation process is complete.

Here are a couple of auxiliary subroutines to the pattern creation subroutine:

```
mark.4:
LET m%(r%,c%)=filled%
LET m%(c%,n%+n%-odd%+1-r%)=filled%
LET m%(n%+n%-odd%+1-r%,n%+n%-odd%+1-c%)=filled%
LET m%(n%+n%-odd%+1-c%,r%)=filled%
RETURN
path.info:
LET pn%=pn%+1
LET pl%(r%,c%)=pn%
LET r%(pn%)=r%
LET c%(pn%)=c%
RETURN
```

The mark.4 subroutine marks a block cell in quadrants I through IV. The path.info subroutine records the location of path number pn%. Pl%(r%,c%) contains the number of the path that starts at location r%, c%. It is 0 if no path starts there. R%(pn%) and c%(pn%) store the row and column addresses of path pn%.

Displaying the Puzzle Pattern

The next subroutine displays the puzzle on the screen:

```
prt.pzl:
WINDOW 1
CLS
CALL TEXTSIZE(9)
CALL TEXTFACE(32)
FOR r%=1 TO m%
FOR c%=1 TO m%
LET px%=(c%-1)*c.size%+ulc%
LET py%=(r%-1)*c.size%+ulc%
IF m%(r%,c%)=empty% THEN find.head
LINE (px%,py%)-STEP(c.size%,c.size%),1,bf
GOTO try.another
find.head:
LET hn%=pl%(r%,c%)
```

```
IF hn%=0 THEN try.another
LET n$=STR$(hn%)
LET n$=RIGHT$(n$,LEN(n$)-1)
CALL MOVETO(px%+2,py%+10)
PRINT n$
try.another:
NEXT c%,r%
```

The first series of lines draws the block cells and places numbers in the upper left corner of each head cell. The following block of lines draws in the cell-divider lines:

```
FOR l%=0 TO m%
LINE (ulc%,ulc%+l%*c.size%)-
      (ulc%+m%*c.size%,ulc%+l%*c.size%)
LINE (ulc%+l%*c.size%,ulc%)-
      (ulc%+l%*c.size%,ulc%+m%*c.size%)
NEXT l%
CALL TEXTFACE(0)
CALL TEXTSIZE(12)
RETURN
```

Printing the Screen

The following lines send a copy of the screen to the Imagewriter printer:

```
prt.screen:
CALL HIDECURSOR
LCOPY
CALL SHOWCURSOR
RETURN
```

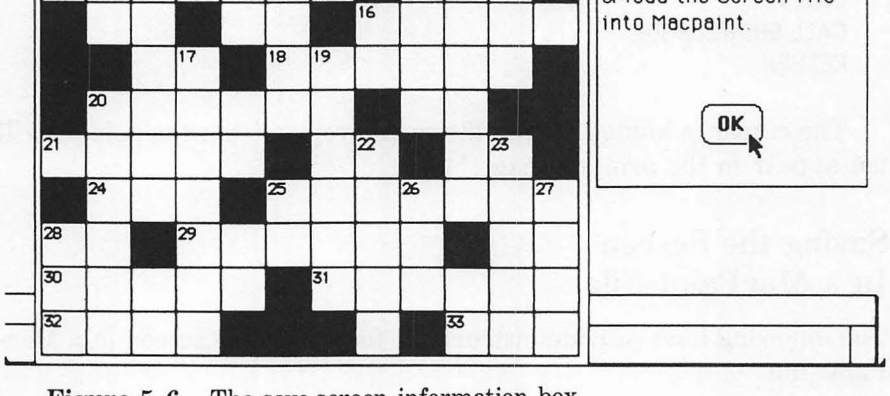
The cursor is hidden during the screen copy so that the pointer will not appear in the printed image.

Saving the Screen In a MacPaint File

The following lines provide instructions for saving the screen in a MacPaint file.

```
save.screen:
WINDOW 2,,(dg.x1%+6,dg.y%)-(dg.x1%+162,dg.y%+216),3
CLS
PRINT "SAVE SCREEN:"
PRINT
PRINT"Type command-shift-3"
PRINT "to save the screen"
PRINT "image in a Screen file."
PRINT "Then quit this program"
PRINT "& load the Screen file"
PRINT "into Macpaint."
BUTTON 1,1,"OK",(60,168)-(96,192)
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
WINDOW CLOSE 2
RETURN
```

These lines simply provide an information box (Figure 5-6) explaining that COMMAND-SHIFT-3 causes the screen contents to be saved in a screen file which may later be loaded into MacPaint.



SAVE SCREEN:

Type command-shift-3
to save the screen
image in a Screen file
Then quit this program
& load the Screen file
into Macpaint.

Figure 5-6. The save-screen information box

—Using the Program

After typing in the entire program and carefully checking the listing, try to run it. You should be able to reproduce the screens shown in Figures 5-2 through 5-6.

The Redraw screen command is provided on the Crossword menu so that you may stop the program to perform a BASIC command and later restart the program and redraw the latest puzzle pattern.

To do this, proceed as follows:

Select Stop from the Run menu. Type in the command you want. For instance, after saving a screen image, you might want to rename the file. Type the command:

```
NAME        "Screen 0" AS "Xword Puzzle"
```

Then select Continue from the Run menu, and select Redraw screen from the Crossword menu.

Note: Occasionally the puzzle pattern will consist entirely of block cells. When this happens, simply select New puzzle from the Crossword menu.

Chapter 6

Playback

This program turns your Macintosh into a game machine similar to several popular electronic games, such as Merlin and Simon. The object of the game is to play back a sequence of notes generated by the computer. Each time you repeat a sequence correctly, the computer adds a new note to the end of the sequence. You must continue to play back the sequence up to a preset sequence length in order to score a success.

Most people can recall a sequence of as many as seven notes without difficulty. Playback lets you play with sequences of as many as 99 notes.

—Operating Instructions for Playback

Figure 6-1 shows the Playback machine with all of its controls and switches identified.

The four squares in the center are play back buttons. When the computer produces a tone, one of the buttons lights up. To playback a computer-generated sequence, you press the playback buttons in the proper order.

The computer keeps track of your cumulative score in the left-hand

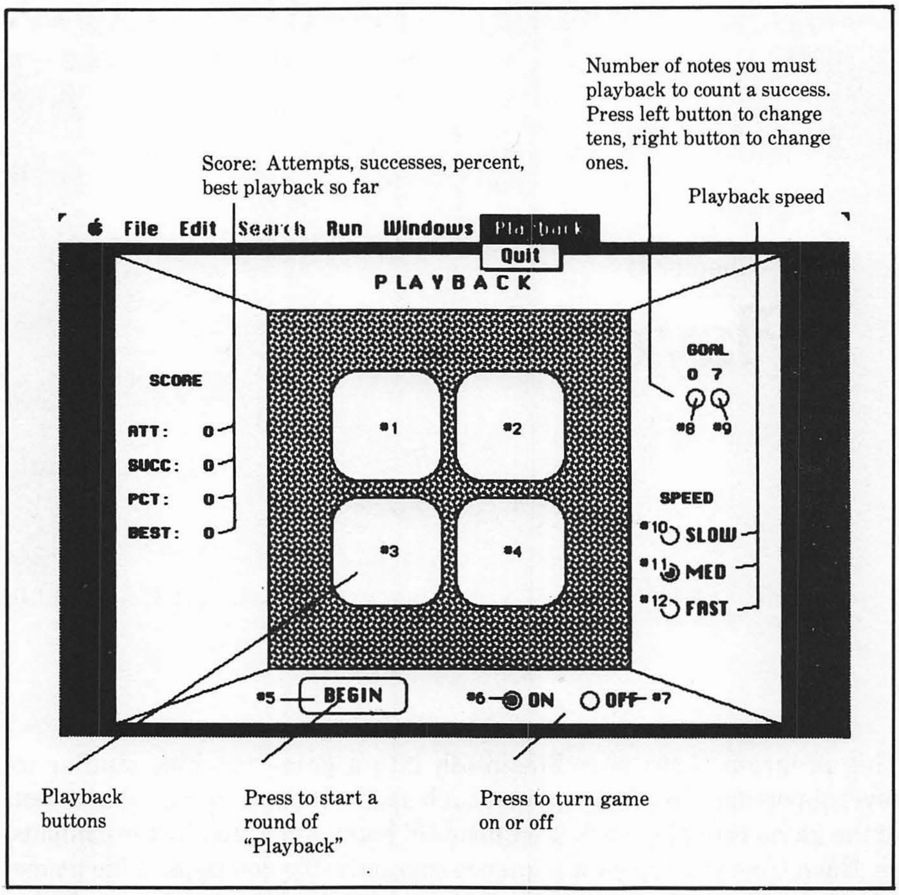

Number of notes you must playback to count a success. Press left button to change tens, right button to change ones.
Score: Attempts, successes, percent, best playback so far
Playback speed
File Edit Search Run Windows Playback
Quit
PLAYBACK
SCORE
ATT: 0
SUCC: 0
PCT: 0
BEST: 0
#1
#2
#3
#4
GOAL
0 7
#8 #9
SPEED
#10 SLOW
#11 MED
#12 FAST
#5 BEGIN
#6 ON
OFF #7
Playback buttons
Press to start a round of "Playback"
Press to turn game on or off

Figure 6-1. The Playback unit. The numbers #1 through #12 show how each button is referenced by the program

panel of the machine: attempts, successes (times you reached the goal), percentage of successes, and best (longest) play back.

On the right-hand panel are two control sections labeled GOAL and SPEED. The goal indicates the number of notes you must playback before the computer will score a success. To change the goal, click on the tens or ones button. The speed setting determines how quickly the computer will play its sequence.

On the bottom panel of the game unit are three buttons: BEGIN, ON, and OFF. Turning the machine off deactivates all the machine's functions. Turning the machine on again reactivates all functions and resets the scoring record.

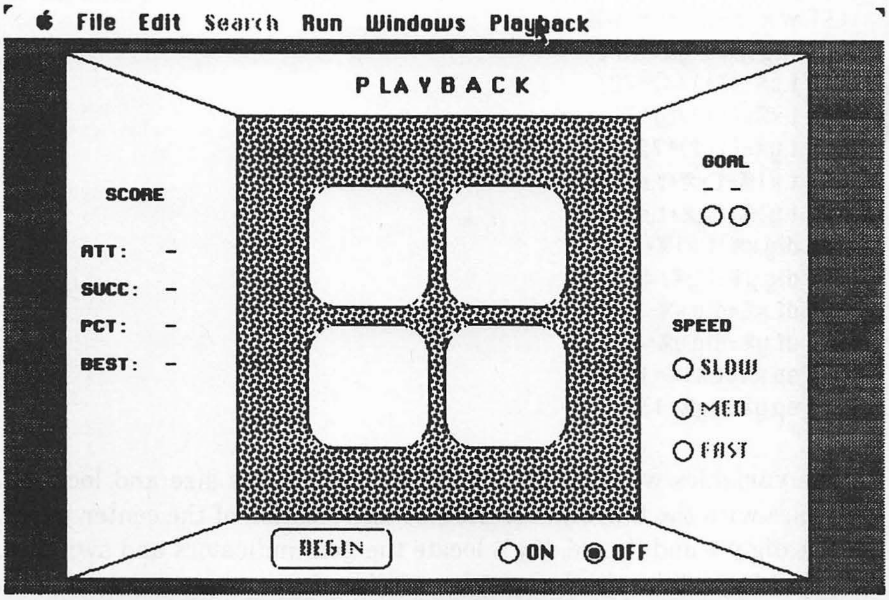


Figure 6-2. The Playback unit

Figure 6-2 shows the appearance of the unit when it is off.

To start a round of the Playback game, you press the BEGIN button. The computer will play a note and wait for you to repeat the same note. If you play it back correctly, the computer will add a note to the sequence and play it again. This process continues until you make an error or reach the preset goal.

An error is also registered if you play back the notes too slowly. The computer allows a delay of two seconds at most between your playback notes.

—The Program

The first block of lines sets up constants pertaining to the window size and location and the game unit's appearance.

```
LET w.w%=6*72
LET w.l%=(4+1/4)*72
LET w.x%=(1/2)*72
LET w.y%=(3/8)*72
LET w.x1%=w.x%+w.w%
LET w.y1%=w.y%+w.l%
LET t.s%=(3+1/4)*72
LET t.x%=(1+3/8)*72
LET t.y%=(1/2)*72
LET t.x1%=t.x%+t.s%
LET t.y1%=t.y%+t.s%
LET dig.x%=t.x1%+38
LET dig.y%=t.y%+46
LET df.x%=dig.x%-3
LET df.y%=dig.y%+4
LET sp.x%=t.x1%+18
LET sp.y%=t.y%+130
```

The variables with the w. prefix set the window size and location. Variables with the t. prefix set the size and location of the center panel. Dig.x%,dig.y% and dg.x%,df.y% locate the goal indicators and switches. Sp.x% and sp.y% locate the speed controls.

The next section of the program initializes certain other constants and arrays.

```
RANDOMIZE TIMER
LET ms%=99   :REM must be <100
DIM cq%(ms%), speed$(3), speed%(3), rr%(3, 4), fq%(4),
      gray%(3), top%(3)
LET top%(0)=t.y%
LET top%(1)=t.x%
LET top%(2)=t.y1%
LET top%(3)=t.x1%
LET b.s%=72
LET oval%=b.s%\2
LET b.zone%=b.s%+(1/8)*72
LET yes%=(1=1)
LET no%=(1=0)
LET loops.persecond=5041   :REM integer FOR/NEXT cycles per
      second
FOR j%=1 TO 3
READ speed$(j%), seconds%
speed%(j%)=18.2\seconds%   :REM convert to SOUND duration
NEXT j%
DATA SLOW, 4, MED, 6, FAST, 8
FOR j%=1 TO 4
READ fq%(j%)
```

```
NEXT j%
DATA 440, 550, 660, 880
LET nc%=4
FOR j%=0 TO 3
READ pattern%
LET gray%(j%)=pattern%
NEXT j%
DATA &HB130, &H031B, &HD8C0, &H0C8D
```

Variable ms% is the longest allowable sequence that may be set as a goal. You may change 99 to any positive whole number less than 100.

Array cq%( ) stores the current sequence. Speed$( ) stores the labels SLOW, MED, and FAST for the game unit. Speed%( ) stores the duration assigned to each speed. Rr%(,) stores the coordinates of the four playback buttons. Array fq%( ) stores the frequencies assigned to the playback buttons. Gray%( ) holds the codes that produce the speckled background pattern of the center panel. Top%( ) holds the coordinates of the center panel. Parameters b.s%, oval%, and b.zone% determine the button sizes and shapes.

Setting Up the Screen

The next block of lines sets up the window and creates the game unit on the screen.

```
WINDOW 1,, (w.x%, w.y%)-(w.x1%, w.y1%), 3
LET game.on%=yes%
GOSUB reset.params
GOSUB machine.outline
GOSUB put.labels
GOSUB calc.playregions
GOSUB create.playbuttons
GOSUB create.controlbuttons
GOSUB create.lengthbuttons
GOSUB put.length
GOSUB create.speedbuttons
GOSUB put.scores
MENU 6, 0, 1, "Playback"
MENU 6, 1, 1, "Quit  "
ON MENU GOSUB menu.activity
MENU ON
LET ok%=yes%
GOSUB respond
```

Most of the subroutine references are self-explanatory, and all will be presented in detail as we go along.

These lines also activate the Playback menu shown in Figure 6-1.

The Idle Loop

The next block provides an idle loop (a sequence of lines that is repeated until some action is requested).

```
loop:
DIALOG ON
WHILE DIALOG(0)<>1
WEND
DIALOG OFF
LET btn%=DIALOG(1)
IF btn%>9 THEN GOSUB change.speed: GOTO loop
IF btn%>7 THEN GOSUB change.length: GOTO loop
IF btn%=6 OR btn%=7 THEN GOSUB switch.game: GOTO loop
IF btn%=5 THEN GOSUB begin: GOTO loop
SOUND fq%(btn%), speed%(speed%)
GOTO loop
```

The button numbers correspond to those shown in Figure 6-1. For instance, buttons 10, 11, and 12 are the speed selectors; hence, when btn%>9, the program executes the change.speed subroutine.

If the button number corresponds to a playback button (btn%<5), the computer generates the sound assigned to that button. This feature lets you practice hitting the buttons before beginning a game.

Menu Selections

Here's the routine to handle selections from the Playback menu:

```
menu.activity:
IF MENU(0)<>6  THEN RETURN
IF MENU(1)<>1 THEN RETURN
LET ok%=no%
GOSUB respond
WINDOW CLOSE 1
END
```

The only menu option is Quit. If you select that option, the subroutine responds with a quick sign-off couplet and then the program ends.

Resetting the Game Parameters

The next block of lines resets the scores, speed settings, and the goal to their initial values. These lines are executed when you start the program and each time you turn on the game machine.

```
reset.params:
LET d.level%=7
LET score%(1)=0   :REM tries
LET score%(2)=0   :REM successes
LET score%(3)=0   :REM percent
LET score%(4)=0   :REM highest goal reached
LET speed%=2
RETURN
```

D.level% is the initial goal setting. Speed% is the initial speed setting, corresponding to MED. Depending on your preference, you may change d.level% to any value from 1 to ms%, and speed% to any value from 1 to 3.

Drawing the Game Unit

The following lines draw the outlines of the game unit:

```
machine.outline:
CALL FILLRECT(VARPTR(top%(0)), VARPTR(gray%(0)))
LINE (t.x%, t.y%)-(t.x1%, t.y1%), , b
LINE (0, 0)-(t.x%, t.y%)   :REM ulc
LINE (w.w%, 0)-(t.x1%, t.y%)   :REM urc
LINE (0, w.1%)-(t.x%, t.y1%)   :REM llc
LINE (w.w%, w.1%)-(t.x1%, t.y1%)   :REM lrc
RETURN
```

FILLRECT is one of the Macintosh's built-in subroutines. It fills the rectangle specified by array top%( ) with the pattern specified by the array gray%( ). In this case, FILLRECT provides the speckled pattern of the center panel.

The first LINE statement draws the outline of the center panel. The remaining LINE statements draw the diagonal contour lines to the upper left corner, upper right corner, lower left corner, and lower right corner.

Setting Up the Game Buttons

The next lines calculate the coordinates for the location of the four playback buttons:

```
calc.playregions:
FOR b%=1 TO 4
LET b.x%=((b%-1) MOD 2 )*b.zone%+(1+15/16)*72
LET b.y%=((b%-1)\2)*b.zone%+(1+1/16)*72
LET b.x%=((b%-1) MOD 2 )*b.zone%+(1+15/16)*72
LET b.y%=((b%-1)\2)*b.zone%+(1+1/16)*72
LET rr%(0, b%)=b.y%    :REM top
LET rr%(1, b%)=b.x%    :REM left
LET rr%(2, b%)=b.y%+b.s%    :REM bottom
LET rr%(3, b%)=b.x%+b.s%    :REM right
NEXT b%
RETURN
```

While the computer is playing its sequence, the buttons are treated as rounded rectangles rather than as true dialog buttons. The following lines draw four rounded rectangles in the playback button regions:

```
create.playregions:
FOR b%=1 TO 4
CALL FRAMEROUNDRECT(VARPTR(RR%(0, b%)), oval%, oval%)
NEXT b%
RETURN
```

While the computer accepts your attempts at playing back a sequence, the buttons are true dialog buttons. The following lines draw the dialog buttons:

```
create.playbuttons:
FOR b%=1 TO 4
BUTTON b%, 1, "", (rr%(1, b%), rr%(0, b%))-(rr%(3, b%), rr%(2,
      b%)), 1
NEXT b%
RETURN
```

These lines create the BEGIN, ON, and OFF buttons:

```
create.controlbuttons:
BUTTON 5, ABS(game.on%), "BEGIN", (t.x%+20,
      w.l%-30)-(t.x%+90, w.l%-6)
```

```
BUTTON 6, 1+ABS(game.on%), "ON", (t.x%+150,
      w.1%-24)-(t.x%+189, w.1%-6), 3
BUTTON 7, 1+ABS(NOT(game.on%)), "OFF", (t.x%+200,
      w.1%-24)-(t.x%+240, w.1%-6), 3
RETURN
```

Here are the lines that create the two length buttons which appear under the goal indicator:

```
create.lengthbuttons:
FOR b%=0 TO 1
LET b.x%=df.x%+b%*16
BUTTON b%+8, ABS(game.on%), "", (b.x%, df.y%)-(b.x%+16,
      df.y%+16), 3
NEXT b%
RETURN
```

Button 8 changes the tens digit, and button 9 changes the ones digit of the current goal setting.

The next block of lines creates the speed selection buttons.

```
create.speedbuttons:
FOR b%=0 TO 2
LET b.y%=sp.y%+b%*24
LET b.y1%=b.y%+34
LET b.stat%=(ABS(speed%=b%+1)+1)*ABS(game.on%)
BUTTON b%+10, b.stat%, speed$(b%+1), (SP.X%, b.y%)-(SP.X%+72,
      b.y1%), 3
NEXT b%
RETURN
```

Control Button Subroutines

Whenever you press one of the goal buttons, the following lines calculate a new goal value, d.level%:

```
change.length:
LET digit%=btn% MOD 2
ON digit%+1 GOSUB tens, ones
IF d.level%=0 THEN LET d.level%=1
GOSUB put.length
RETURN
```

```
ones:
LET ones%=d.level% MOD 10
LET tens%=d.level%-ones%
LET ones%=(ones%+1) MOD 10
LET d.level%=(tens% + ones%) MOD (ms%+1)
RETURN
tens:
LET d.level%=d.level%+10
IF d.level%>ms% THEN d.level%=d.level% MOD 10
RETURN
```

The ones and tens buttons operate independently of one another. Pressing either button changes only the corresponding tens or ones digit.

If you attempt to exceed the limit value, ms%, the tens digit is set to 0. If you attempt to set a goal of 0, a goal of 1 is used instead.

Whenever you request a change in playback speed, the following lines reset the speed buttons accordingly:

```
change.speed:
BUTTON speed%+9, 1
LET speed%=btn%-9
BUTTON btn%, 2
RETURN
```

Pressing the ON or OFF button activates the following lines:

```
switch.game:
IF game.on% AND (btn%=6) THEN RETURN
LET game.on%=(btn%=6)
IF game.on% THEN GOSUB reset.params
FOR b%=1 TO 5
BUTTON b%, ABS(game.on%)
NEXT b%
BUTTON 6, 1+ABS(game.on%)
BUTTON 7, 1+ABS(NOT(game.on%))
BUTTON 8, ABS(game.on%)
BUTTON 9, ABS(game.on%)
FOR b%=10 TO 12
LET b.stat%=(ABS(speed%=b%-9)+1)*ABS(game.on%)
BUTTON b%, b.stat%
NEXT b%
```

```
GOSUB put.length
GOSUB put.scores
LET ok%=game.on%
GOSUB respond
RETURN
```

If the game is already on and you press the ON button, the action is ignored (IF game.on% AND btn%=6...).

Otherwise, game.on% is reset according to whether you pressed ON or OFF. If the new game is on, the original game parameters are restored. Then the program resets the inactive/active/selected status of all 12 buttons according to the value of game.on%.

The respond subroutine sounds a welcome or sign-off couplet according to whether you have just turned the game on or off.

Applying the Labels

These next lines apply the text labels to the game machine:

```
put.labels:
CALL TEXTFONT(3)
CALL TEXTSIZE(12)
CALL TEXTFACE(1)
CALL MOVETO(168, 23)
PRINT "P LA Y B A C K";
CALL TEXTFONT(4)
CALL TEXTSIZE(9)
CALL MOVETO(t.x1%+38, t.y%+30)
PRINT "GOAL"
CALL MOVETO(t.x1%+20, t.y%+126)
PRINT "SPEED"
LOCATE 8, 4
PRINT "SCORE"
LOCATE 11, 2
PRINT "ATT:"
LOCATE 13, 2
PRINT "SUCC:"
LOCATE 15, 2
PRINT "PCT:"
LOCATE 17, 2
PRINT "BEST:"
RETURN
```

After each round, the following lines are executed to update the scores:

```
put.scores:
CALL TEXTFONT(4)
CALL TEXTSIZE(9)
FOR j%=1 TO 4
LOCATE 9+2*j%, 7
IF NOT game.on% THEN PRINT "  -" ELSE PRINT USING
      "###";score%(j%)
NEXT j%
RETURN
```

If the game is off, hyphens are printed in the numeric fields of the score panel. Otherwise, the appropriate numbers are printed.

Each time the goal is changed, the following lines print the new setting:

```
put.length:
CALL TEXTFONT(4)
CALL TEXTSIZE(9)
IF NOT game.on% THEN blank.digits
LET tens$=RIGHT$(STR$(d.level%\10), 1)
LET ones$=RIGHT$(STR$(d.level% MOD 10), 1)
GOTO show.digits
blank.digits:
LET tens$="-"
LET ones$="-"
show.digits:
CALL MOVETO(dig.x%, dig.y%)
PRINT tens$
CALL MOVETO(dig.x%+16, dig.y%)
PRINT ones$
RETURN
```

Again, if the game is off, hyphens are printed instead of digits.

Audible Response Subroutine

The following lines sound a descending couplet when you turn the game off or when you make an error during a playback attempt. The same lines sound an ascending sequence of notes when you turn on the game or reach the goal:

```
respond:
IF ok% THEN good
SOUND 300, 2   :REM "wrong answer" sound
SOUND 150, 2
RETURN
good:
FOR s%=1 TO 4
CALL INVERTROUNDRECT(VARPTR(RR%(0, s%)), oval%, oval%)
SOUND fq%(s%), 1   :REM "right answer" sound
CALL INVERTROUNDRECT(VARPTR(RR%(0, s%)), oval%, oval%)
NEXT s%
RETURN
```

Along with the sound indicating a success, the computer blinks the corresponding playback buttons (CALL INVERTROUNDRECT...).

Test Point

To test your work so far, add these temporary lines to the end of the listing:

```
begin:
RETURN
```

After carefully checking each block for typographical errors and omissions, close the listing window and run the program. If you have typed everything correctly, you should see a screen similar to Figure 6-1.

Try all 12 buttons. Pressing a playback button should produce a sound. Pressing any other button (except BEGIN) should produce the appropriate result. Try changing the speed and notice the effect on the sounds produced by the playback buttons.

Try using the goal selector buttons. You should be able to specify every value from 1 to ms% (which we set at 99).

After you have confirmed that everything is working properly so far, delete the following lines:

```
begin:
RETURN
```

The Playback Subroutine

Now we'll add the lines that handle the playback function. These lines are activated when you press the BEGIN button:

```
begin:
LET cp%=0
add.to:
LET cp%=cp%+1
LET c%=INT(RND*nc%)+1
LET cq%(cp%)=c%
GOSUB play.sequence
DIALOG ON
LET ok%=yes%
GOSUB playback
IF ok%=no% OR cp%=d.level% THEN end.round
LET delay%=loops.persecond*.375
GOSUB pause
GOTO add.to
```

The variable cp% indicates the current length of the playback sequence. The add.to routine randomly selects one of the four playback sounds c% and adds it to the current sequence stored in cq%( ).

Then the program calls a subroutine to play the notes of the sequence. The DIALOG ON statement allows you to use the playback buttons to play back the sequence.

To receive your playback attempt, the program calls the playback subroutine. Upon return from this subroutine, the variable ok% indicates whether you made an error or not. If you made an error (OK%=no%) or reached the goal (cp%=d.level%), the program ends the round. Otherwise, it pauses briefly and then produces a new, longer sequence (GOTO add.to).

The following lines are executed at the end of a round:

```
end.round:
LET delay%=loops.persecond*.25
GOSUB pause
GOSUB respond   :REM make appropriate sounds
LET score%(1)=score%(1)+1
LET score%(2)=score%(2)-ok%
LET score%(3)=INT(score%(2)/score%(1)*100+.5)
```

```
IF cp%>score%(4) THEN score%(4)=cp%
GOSUB put.scores
RETURN
```

The computer pauses briefly, makes an appropriate sound (GOSUB respond), and then updates the scores. The new scores are printed in the scoring panel (GOSUB put.scores). After that, the program ends the playback subroutine and returns to the idle loop.

Auxiliary Playback Subroutines

The next block of lines plays the sequence stored in cq%( ):

```
play.sequence:
GOSUB create.playregions
FOR j%=1 TO cp%
CALL INVERTROUNDRECT(VARPTR(RR%(0, cq%(j%))), oval%,
      oval%)
SOUND FQ%(CQ%(J%)), speed%(speed%)
LET delay%=speed%(speed%)*200
GOSUB pause
CALL INVERTROUNDRECT(VARPTR(RR%(0, cq%(j%))), oval%,
      oval%)
NEXT j%
RETURN
```

First, rounded rectangles are drawn over the dialog buttons (GOSUB create.playregions). Then the program plays each note in the current sequence, blinking the corresponding rounded rectangle. In the SOUND statement, FQ%(CQ%(J%)) specifies the frequency of each note and speed%(speed%) specifies the duration.

The following lines are executed when it's your turn to play back the sequence:

```
playback:
LET j%=0
ON TIMER(2) GOSUB activity.check
pb.loop:
LET time.left%=yes%
TIMER ON
WHILE DIALOG(0)<>1 AND time.left%
```

```
WEND
TIMER OFF
DIALOG OFF
IF time.left%=no% THEN pb.error
LET btn%=DIALOG(1)
IF btn%>4 THEN pb.error
SOUND fq%(btn%), speed%(speed%)
LET j%=j%+1
IF btn%<>cq%(j%) THEN pb.error
IF j%<cp% THEN pb.loop ELSE pb.done
pb.error:
LET ok%=no%
LET cp%=cp%-1    :REM didn't get last character
pb.done:
RETURN
activity.check:
LET time.left%=no%
RETURN
```

Variable j% keeps track of the number of notes you've played back. The ON TIMER statement gives you two seconds to press the next playback button; if you wait longer than that, the activity.check subroutine will record that fact, causing the program to record an error.

Pb.loop is a repeated sequence of lines that lets you play back the sequence of notes. The loop ends when you complete the sequence, press the wrong button, or wait more than two seconds before playing the next note of a sequence.

Pause Subroutine

The last subroutine of our program provides a simple pause. The length of the pause is set by the value of delay%. Delay%=5041 produces a one-second pause.

```
pause:
FOR xx%=1 TO delay%
NEXT xx%
RETURN
```

—Testing and Using the Program

Carefully check all blocks added since the test point. Close the listing window and run the program. Set the goal to 3. Press the BEGIN but-

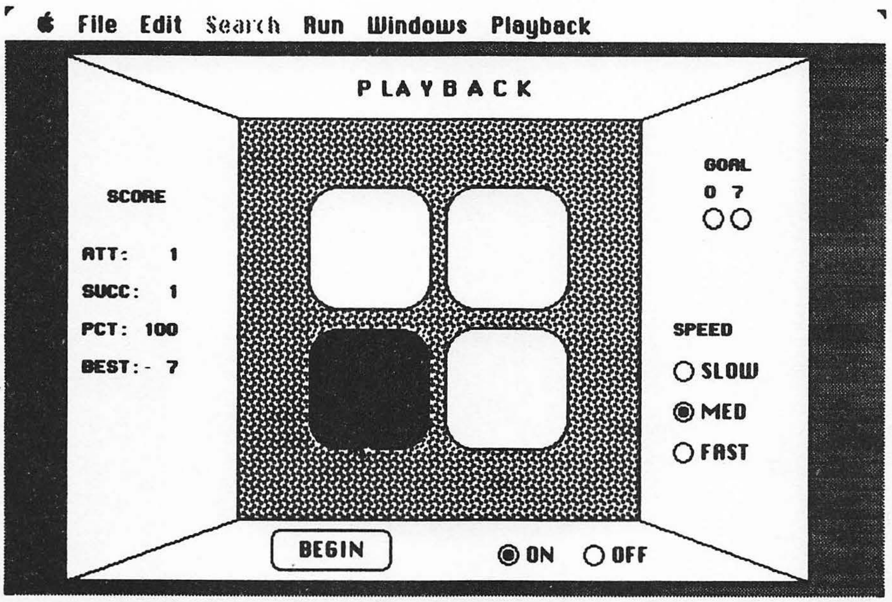


Figure 6-3. The Playback unit during a playback sequence

ton to start a game. The computer should play a single note and highlight the corresponding playback button. Play the same button by clicking the mouse on it. The computer should play that note again, followed by a new note. Continue until you repeat a sequence of three. The computer should sound the response indicating a success and update the scores to ATT=1, SUCC=1, PCT=100, and BEST=3.

Press BEGIN to play another round. This time, make an error. The computer should sound the error response and update your scores appropriately.

Figure 6-3 shows the game unit during a playback sequence.

Now begin increasing the goal, and see how long a sequence you can play back!

Chapter 7

Electronic Billiards

The Billiard Practice program turns your Macintosh display into an electronic billiard table. You can use it for practicing and experimenting with different kinds of angle shots and to play simplified games of billiards.

The table is designed to match the appearance and proportions of a real billiard table. As in billiards (not pool), the table has no holes. The object of billiards is to hit the object ball with the cue ball after first striking one or more rails.

There are a few peculiarities of our electronic billiard table. First of all, there is no table friction to slow down a ball. Once started, a ball rolls until it hits another ball or until you stop the ball by clicking the mouse. Another difference from the real world is that when the cue ball hits the object ball, the cue ball stops and the object ball rolls away from the point of impact. If the object ball hits the cue ball, the motion is again reversed. In true billiards, two object balls are used; our version has only one.

Figure 7-1 shows the billiard table and billiard control panel.

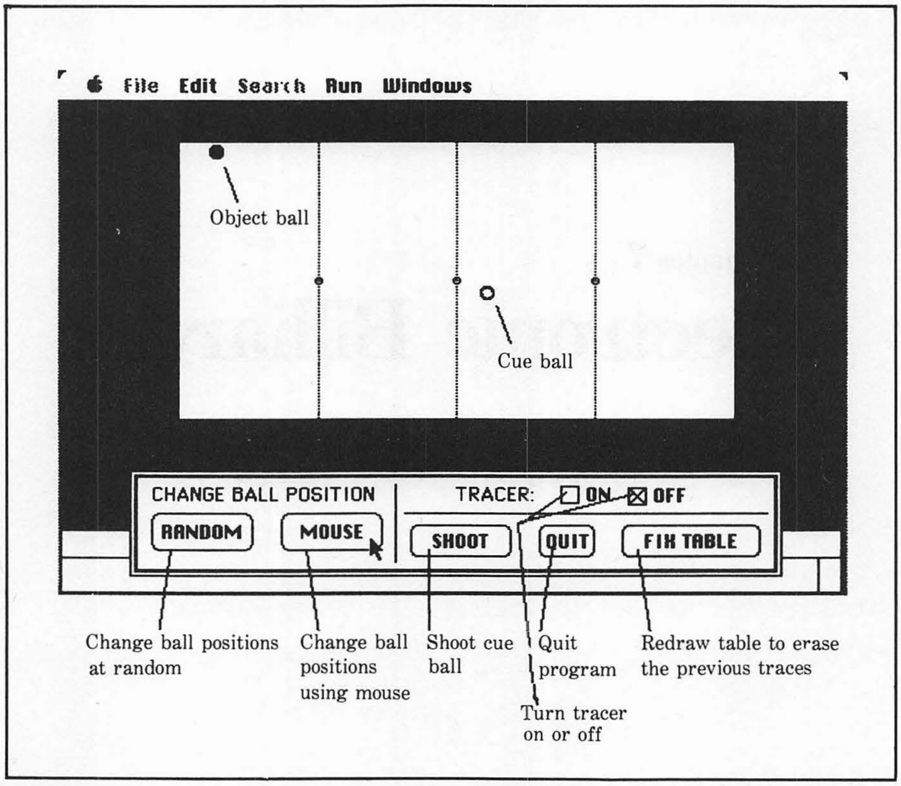

File Edit Search Run Windows
Object ball
Cue ball
CHANGE BALL POSITION
RANDOM
MOUSE
TRACER: ON OFF
SHOOT
QUIT
FIX TABLE
Change ball positions at random
Change ball positions using mouse
Shoot cue ball
Quit program
Redraw table to erase the previous traces
Turn tracer on or off

Figure 7-1. The electronic billiards table and control panel

—Brief Operating Instructions

The program will position the balls at random on the table (RANDOM button) or you can rearrange them manually (MOUSE button). When you're ready to shoot, press the SHOOT button. Then use the mouse to point to the destination—which can be any spot on the table. Click the mouse button to start the cue ball rolling in the specified direction.

When the cue ball strikes a rail, it bounces off of the rail at the angle of deflection. When the cue ball strikes the object ball, the object ball bounces off at the angle of collision. Figure 7-2 illustrates both angles.

Our electronic billiards has an optional tracer feature that shows the path of motion for each ball. You may find this helpful in sharpening your game.

Figures 7-3 through 7-6 illustrate a typical sequence in using electronic billiards.

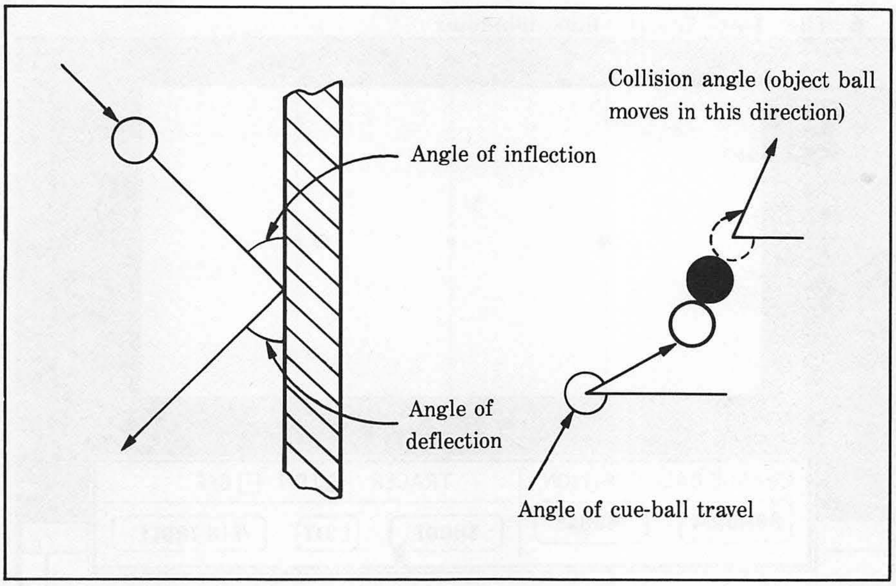
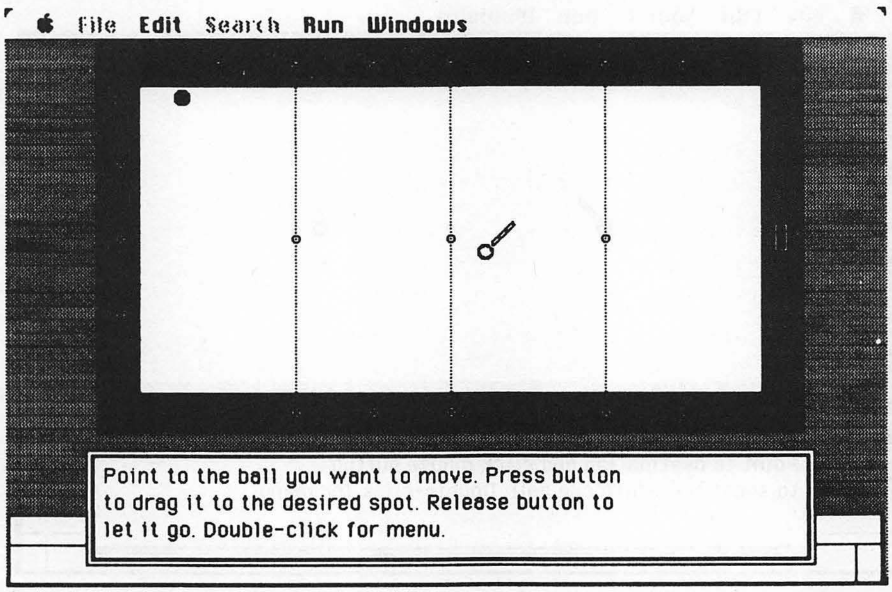


Figure 7-2. Angle of inflection/deflection and angle of collision



Point to the ball you want to move. Press button
to drag it to the desired spot. Release button to
let it go. Double-click for menu.

Figure 7-3. Screen appearance during manual repositioning of the balls

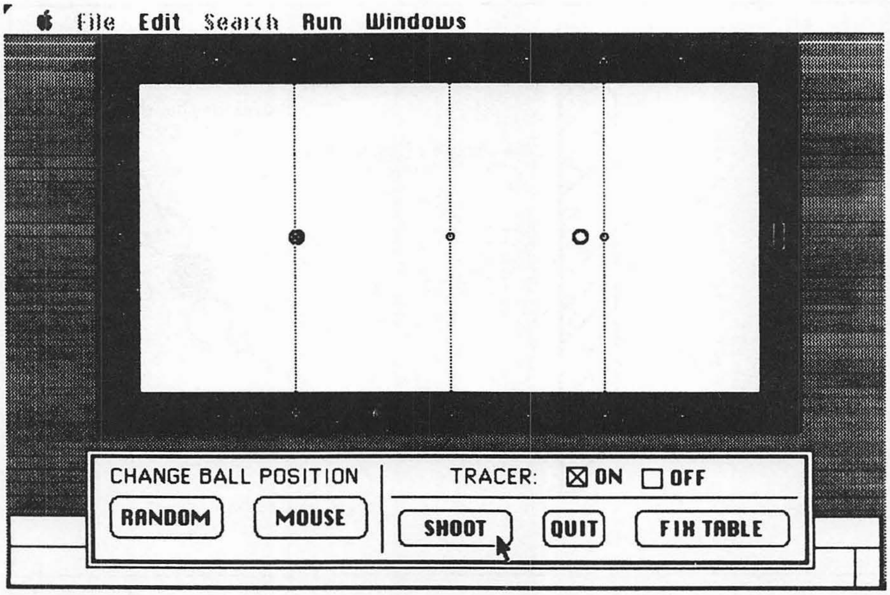
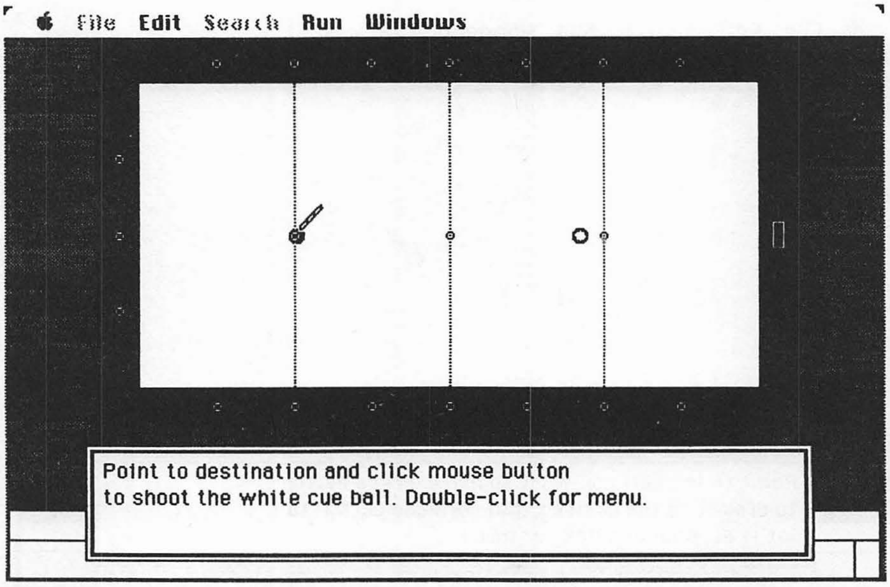
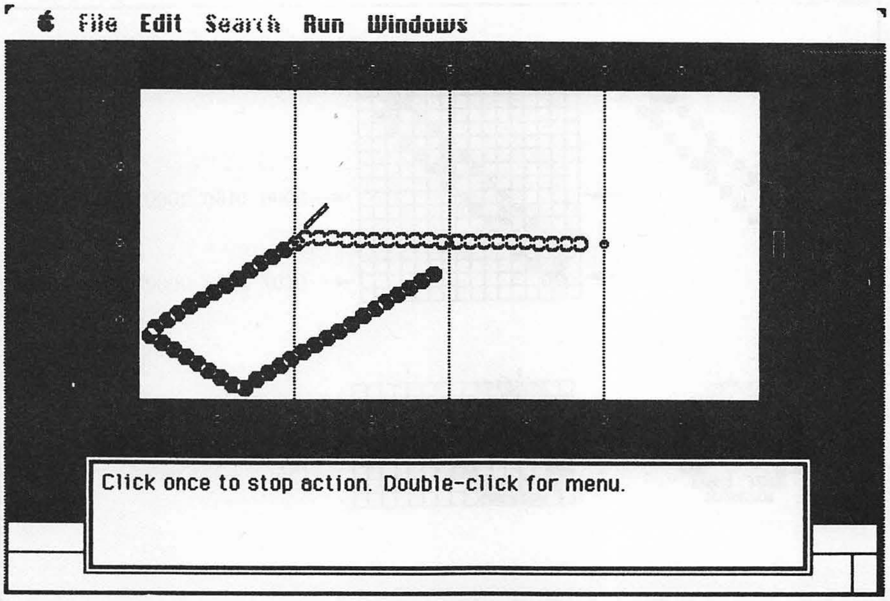


Figure 7-4. The balls have been repositioned and the tracer function is on



Point to destination and click mouse button
to shoot the white cue ball. Double-click for menu.

Figure 7-5. Screen appearance while specifying the target for the cue ball



Click once to stop action. Double-click for menu.

Figure 7-6. Paths of the white cue ball before impact and black object ball after impact

—The Program

The cursor and ball data are derived from the plans shown in Figure 7-7. For each object shown, one row of dots is represented as a hexadecimal number. (For further details, read about PUT and SETCURSOR in the Microsoft BASIC interpreter manual.)

The first block contains graphics data for the cue-stick cursor that is used when you are repositioning the balls or shooting the cue ball.

```
DATA &H0004, &H000C, &H0016, &H0024, &H0048, &H00D0,
     &H0160
DATA &H0240, &H0480, &H0D00, &H1600, &H2400, &H4800,
     &H5000
DATA &H6000, &H0000
DATA &H000E, &H001E, &H003F, &H007E, &H00FC, &H01F8,
     &H03F0
```

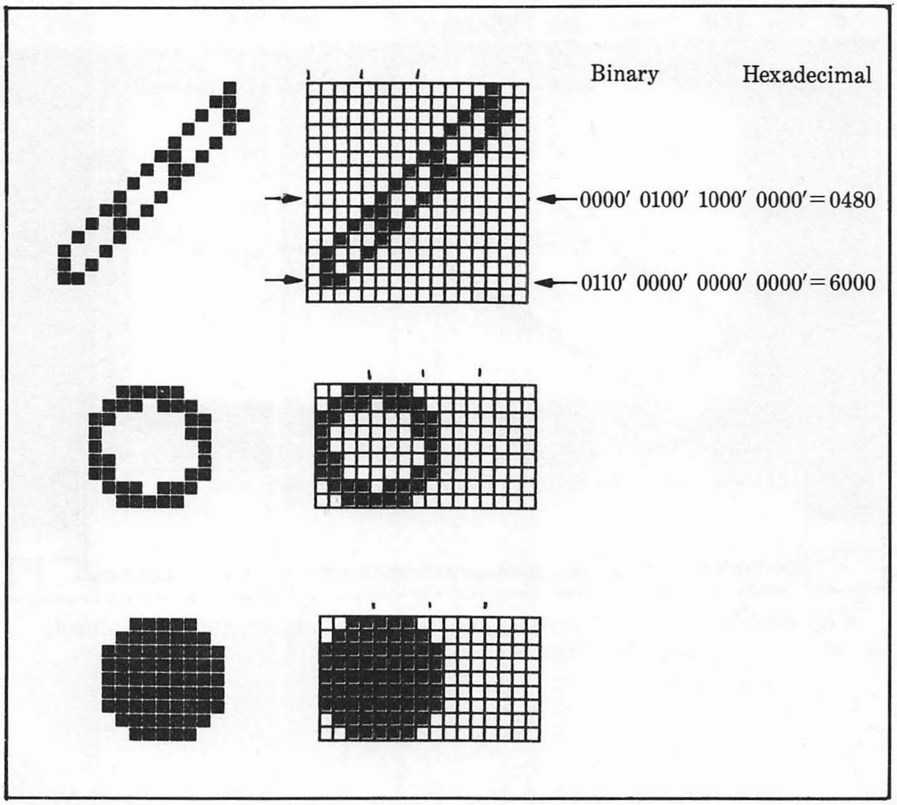


Figure 7-7. The cue-stick cursor and billiard ball patterns

```
DATA &H07E0, &H0FC0, &H1F80, &H3F00, &H7E00, &HFC00,
     &HF800
DATA &HF000, &HE000
DATA 11, 0
```

The next block contains the graphics data for the cue and object balls:

```
DATA 9, 9
DATA &H3E00, &H7700, &HC180, &H8180, &H8080, &HC080,
     &HC180
DATA &H7700, &H3E00
DATA 9, 9
DATA &H3E00, &H7F00, &HFF80, &HFF80, &HFF80, &HFF80,
      &HFF80
DATA &H7F00, &H3E00
```

The next series of lines sets up various functions and array constants:

```
DEF FNdistance(x,y,x1,y1)=SQR((x-x1)*(x-x1)+(y-y1)*(y-y1))
RANDOMIZE TIMER
DIM pool.cursor%(33), gray%(3), black%(3), ball%(10,2),locx(2),
      locy(2)
FOR j%=0 TO 33
READ pool.cursor%(j%)
NEXT j%
FOR j%=1 TO 2
FOR code%=0 TO 10
READ ball%(code%,J%)
NEXT code%, j%
FOR j%=0 TO 3
LET black%(j%)=&HFFFF    :REM black pattern
LET gray%(j%)=&HAA55     :REM halftone pattern
NEXT j%
```

FNdistance calculates the distance between two points, x,y and x1,y1. Pool.cursor%( ) stores the cue-stick cursor data. Gray%( ) and black%( ) store graphics patterns for gray and black. Ball%(,) stores the data for the two balls.

The following lines store constants and parameters:

```
LET yes%=(1=1)
LET no%=(1=0)
LET pi=4*ATN(1)
LET qtr.circ=pi/2
LET threeqtr.circ=3*pi/2
LET ball.dia%=9
LET rail%=24
LET t.width%=5*72
LET t.length%=2.5*72
LET spot.x%=rail%
LET spot.y%=rail%
LET spot.x1%=rail%+t.width%-ball.dia%
LET spot.y1%=rail%+t.length%-ball.dia%
LET speed%=ball.dia%-1
LET trace.on%=no%
```

Here are the window definition parameters:

```
LET wp.x%=3/4*72
LET wp.x1%=wp.x%+t.width%+2*rail%
LET wp.y%=.3*72
LET wp.y1%=wp.y%+t.length%+2*rail%
LET tw.fourth%=t.width%\4
LET tw.eighth%=t.width%\8
LET tl.fourth%=t.length%\4
LET wd.width%=t.width%+2*rail%
LET wd.length%=3/4*72
LET wd.x%=wp.x%
LET wd.y%=wp.y1%+1/4*72
LET wd.x1%=wd.x%+wd.width%
LET wd.y1%=wd.y%+wd.length%
```

The variables having the prefix wp define the billiard table window. The tw-prefix variables hold locations for the billiard table markings. Wd-prefix variables define the control-panel window. Spot-prefix variables indicate the range of allowable ball locations (anywhere within the black rails). Trace.on% holds the status of the tracer function.

Setting Up the Table and Control Box

Now the program can create the two windows and set up the table and control panel:

```
WINDOW 2,,(wd.x%, wd.y%)-(wd.x1%, wd.y1%), 2
WINDOW 1,,(wp.x%, wp.y%)-(wp.x1%, wp.y1%), 3
GOSUB new.table
GOSUB spot.random
start.dialogue:
WINDOW 2
CLS
PRINT " CHANGE BALL POSITION"; PTAB(204); "TRACER:"
LINE (164,1)-(164,52)
LINE (170,20)-(406,20)
BUTTON 1,1,"RANDOM",(6,20)-(72,45)
BUTTON 2,1,"FIX TABLE",(312,28)-(402,49)
BUTTON 3, 1+ABS(trace.on%), "ON", (270, 2)-(304, 14), 2
BUTTON 4, 1+ABS(NOT trace.on%), "OFF", (314, 3)-(354, 15), 2
BUTTON 5,1,"MOUSE",(90,20)-(156,45)
BUTTON 6,1,"SHOOT",(174,28)-(240,49)
BUTTON 7,1,"QUIT",(258,28)-(294,49)
```

The new.table subroutine draws the billiard table. Spot.random repositions the two balls at randomly chosen locations.

Control Panel Monitor

Here is the block that monitors the buttons of the control panel:

```
sd.loop:
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
IF btn%<5 THEN window.ok
FOR j%=1 TO 7
BUTTON CLOSE j%
NEXT j%
CLS
window.ok:
ON btn% GOSUB spot.random, new.table, set.trace, set.trace,
        spot.mouse, shoot, quit
IF btn%>=5 THEN start.dialogue
WINDOW 2
GOTO sd.loop
```

The subroutine references in the line beginning on btn% GOSUB are self-explanatory and will be covered in more detail later as each subroutine is presented.

Upon return from each subroutine, the program refreshes the control-panel window (if necessary) and re-enters the monitor loop (GOTO sd.loop).

Positioning the Balls at Random

The following lines comprise the random ball position subroutine:

```
spot.random:
WINDOW OUTPUT 1
FOR b%=1 TO 2
```

```
PUT (locx(b%), locy(b%)), ball%(0, b%)   :REM erase previous
get.random:
LET r.x%=INT(RND*(spot.x1%-spot.x%+1))+rail%
LET r.y%=INT(RND*(spot.y1%-spot.y%+1))+rail%
IF b%=1 THEN spot.ok:
IF FNdistance(r.x%, r.y%, locx(1), locy(1))<ball.dia% THEN get.random
spot.ok:
LET locx(b%)=r.x%
LET locy(b%)=r.y%
PUT (locx(b%), locy(b%)), ball%(0, b%)
NEXT b%
RETURN
```

To position each ball, the program first erases the ball from its previous position (PUT...). Then it randomly selects a new location (r.x%,r.y%) on the table. When the object ball (b%=2) is being positioned, the program ensures that the new position isn't already occupied by the cue ball (IF FNdistance...).

After getting a new location for the ball, the program updates the location arrays locx(b%), locy(b%) and draws the ball at its new spot.

Drawing the Table

These lines draw the billiard table's rails:

```
new.table:
WINDOW OUTPUT 1
CLS
CALL PENPAT(VARPTR(black%(0)))
CALL PENSIZE(rail%,rail%)
CALL MOVETO(0,0)
CALL LINE(t.width%+rail%+1,0)
CALL LINE(0,t.length%+rail%+1)
CALL LINE(-(t.width%+rail%+1),0)
CALL LINE(0,-(t.length%+rail%+1))
```

The following block supplies the rest of the table's characteristic markings:

```
CALL PENSIZE(1,1)
CALL PENPAT(VARPTR(gray%(0)))
FOR v.line%=tw.fourth% TO 3*tw.fourth% STEP tw.fourth%
```

```
CALL MOVETO(rail%+v.line%,rail%)
CALL LINE(0,t.length%)
CIRCLE (rail%+v.line%,rail%+t.length%\2),2,1
NEXT v.line%
FOR w.dot%=tw.eighth% TO 7*tw.eighth% STEP tw.eighth%
CIRCLE (rail%+w.dot%,rail%\2),2,0
CIRCLE (rail%+w.dot%,t.length%+rail%+rail%\2),2,0
NEXT w.dot%
FOR l.dot%=tl.fourth% TO 3*tl.fourth% STEP tl.fourth%
CIRCLE (rail%\2,rail%+l.dot%),2,0
NEXT l.dot%
LINE (rail%+t.width%+3*rail%\8, rail%+11*t.length%\24)-
      STEP(rail%\4, t.length%\12),0,b
FOR b%=1 TO 2
PUT (locx(b%),locy(b%)),ball%(0,b%)
NEXT b%
RETURN
```

First the program draws the three dotted lines that divide the table into quadrants (FOR v.line%=... NEXT v.line%). Each line has a circular spot at its midpoint.

Then the program supplies the seven dots along the top and bottom rails (FOR w.dot%=...) and the three dots along the left rail (FOR l.dot%=...). Next the program draws the rectangular chalk-box along the right rail.

Now the table is complete, so the program redraws the balls at their current locations. (PUT...).

Tracer and Quit Options

These lines take over when you change a tracer button or select Quit:

```
set.trace:
LET trace.on%=(btn%=3)
BUTTON 3,1+ABS(trace.on%)
BUTTON 4,1+ABS(NOT trace.on%)
RETURN
quit:
WINDOW CLOSE 2
WINDOW CLOSE 1
END
```

On entry to the set.trace subroutine, btn% is 3 or 4, depending on whether you pressed tracer ON or tracer OFF. The program puts an X in the appropriate box (BUTTON 3,... and BUTTON 4,...) and returns to the control-panel monitor.

Positioning the Balls With the Mouse

When you press the MOUSE button, the following lines take over:

```
spot.mouse:
PRINT "Point to the ball you want to move. Press button"
PRINT "to drag it to the desired spot. Release button to"
PRINT "let it go. Double-click for menu.";
WINDOW 1
CALL SETCURSOR(VARPTR(pool.cursor%(0)))
await.selection:
LET event%=MOUSE(0)
WHILE event%<>-1 AND event%<>2
LET event%=MOUSE(0)
WEND
```

The program activates the cue-stick cursor (CALL SETCURSOR) to indicate that the table window (window 1) is active. Then it waits for you to select a ball or request the menu. (See Figure 7-3).

The following lines respond to your selection:

```
IF event%=2 THEN sm.done
LET mx=MOUSE(1)-4
LET my=MOUSE(2)
LET b%=0
LET ball.select%=no%
WHILE b%<2 AND NOT ball.select%
LET b%=b%+1
LET m.to.ball=FNdistance(mx,my,locx(b%),locy(b%))
LET ball.select%=(m.to.ball<=ball.dia%)
WEND
IF NOT ball.select% THEN await.selection
LET mx=locx(b%)
LET my=locy(b%)
WHILE MOUSE(0)=-1
IF MOUSE(1)=mx+4 AND MOUSE(2)=my THEN skip
PUT (mx,my),ball%(0,b%)
```

```
LET mx=MOUSE(1)-4
LET my=MOUSE(2)
PUT (mx,my),ball%(0,b%)
skip:
WEND
```

Event%=2 indicates that you have double-clicked the mouse. In that case, the program exits from the ball-positioning subroutine (IF event%=2 THEN sm.done).

Otherwise, the program determines whether you are pointing to one of the balls. Mx,my is the current position of the mouse. M.to.ball is the distance from the mouse to the cue ball or object ball. When m.to.ball is less than or equal to the ball diameter, the program recognizes that you want to select that ball.

If the mouse is not pointing to either ball, the program returns to the await.selection routine.

Once you have selected a ball with the cue-stick pointer, the program "attaches" that ball to the cue-stick. Whenever you move the stick, the ball follows. The ball stays attached until you release the mouse button.

The following lines determine whether you have left the ball in an acceptable location:

```
IF mx<spot.x% OR mx>spot.x1% OR my<spot.y% OR my>spot.y1%
       THEN cancel.move
IF FNdistance(locx(3-b%),locy(3-b%),mx,my)<ball.dia% THEN
       cancel.move
LET locx(b%)=mx
LET locy(b%)=my
GOTO await.selection
sm.done:
CALL INITCURSOR
RETURN
cancel.move:
PUT (mx,my),ball%(0,b%)   :REM erase last position
PUT (locx(b%),locy(b%)),ball%(0,b%)   :REM restore initial
        position
GOTO await.selection
```

If the ball is off the table or touching another ball, the program returns it to its original location via the cancel.move routine. If the position is okay, it is stored in the position pointer locx( ) and locy( ).

The program then returns to the await.selection routine.

Shooting the Ball

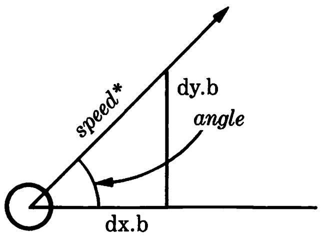
These lines start the shooting procedure:

```
WINDOW OUTPUT 2
CLS
PRINT "Point to destination and click mouse button"
PRINT "to shoot the white cue ball. Double-click for menu."
WINDOW 1
CALL SETCURSOR(VARPTR(pool.cursor%(0)))
s.loop:
LET event%=MOUSE(0)
WHILE event%<>1
LET event%=MOUSE(0)
WEND
```

The lines produce a display similar to that in Figure 7-5. The s.loop routine waits until you press or click the mouse button. After that, the following lines are executed:

```
WINDOW OUTPUT 2
CLS
PRINT "Click once to stop action. Double-click for menu."
WINDOW 1
LET tg.x%=MOUSE(1)-4
LET tg.y%=MOUSE(2)
LET delta.x%=tg.x%-locx(1)
LET delta.y%=tg.y%-locy(1)
GOSUB find.angle
IF NOT a.ok% THEN s.loop
LET dx.b=speed%*COS(angle)
LET dy.b=speed%*SIN(angle)
LET mvb%=1
```

These lines provide a display like that in Figure 7-6 (created with the tracer on). Tg.x%, tg.y% are the coordinates of the destination (the cue-stick location when you pressed the mouse). The program calculates the angle from the cue ball to the destination ball (GOSUB find.angle). It then calculates the corresponding horizontal and vertical increments dx.b and dy.b (see Figure 7-8).



dx.b = *speed* × cos(*angle*)

dy.b = *speed* × sin(*angle*)

**Speed* is the distance the ball travels in direction *angle* in a single cycle of program calculations.

Figure 7-8. Given a ball direction specified as an angle and a speed, the program calculates the corresponding horizontal and vertical increments dx.b and dy.b

Just before executing the next block of lines, the program specifies ball number 1 (cue ball) as the moving object (LET Mvb%=1).

Rolling the Balls

The following lines start the moving-ball procedure:

```
LET event%=MOUSE(0)   :REM begin roll
WHILE event%<>1 AND event%<>2
LET event%=MOUSE(0)
LET new.locx=locx(mvb%)+dx.b   :REM find next location
LET new.locy=locy(mvb%)+dy.b
```

The program will continue to move the balls until you single-click or double-click the mouse button (WHILE event%<>1 AND event%<>2).

First the program looks ahead to the next location—the position the ball will occupy when it has been moved. (LET new.locx=... and LET new.locy=...).

Checking for a Collision

These lines check to see whether the moving ball will collide with the stationary ball:

```
LET hit%=no%   :REM check.for.collision
LET b1%=3-mvb%
LET dist%=FNdistance(new.locx, new.locy, locx(b1%),
      locy(b1%))
IF dist%>ball.dia% THEN cfc.done
LET delta.x%=dx.b
LET delta.y%=dy.b
GOSUB find.angle
IF NOT a.ok% THEN cfc.done
LET b.angle=angle
LET delta.x%=locx(b1%)-new.locx
LET delta.y%=locy(b1%)-new.locy
GOSUB find.angle
IF NOT a.ok% THEN LET angle=b.angle
LET col.angle=angle
LET impact.angle=ABS(b.angle-col.angle)
LET hit%=(impact.angle<qtr.circ) OR
      (impact.angle>threeqtr.circ)
IF NOT hit% THEN cfc.done
LET dx.b=speed%*COS(col.angle)
LET dy.b=speed%*SIN(col.angle)
```

B1% is the number of the stationary ball. If the distance between balls is greater than the ball diameter, no collision can occur, so the collision check is finished (IF dist%>ball.dia% THEN cfc.done).

Otherwise, the program checks to see whether the direction of motion is toward the stationary ball (causing a collision), on a tangent (no collision), or away from the ball (no collision).

B.angle is the angle of cue-ball travel. Col.angle is the angle between the centerpoints of the two balls (see Figure 7-2). Impact.angle is the difference between the two angles. When impact.angle is less than a quarter circle or greater than a three-quarter circle, a collision is imminent.

If no collision is imminent (NOT hit%), the collision check is finished. Otherwise, the horizontal and vertical motion-increments dx.b and dy.b are recalculated from the angle of impact. The new values will be applied to the ball that was stationary at the time of the collision.

The next block of lines checks to see if the moving ball is going to bounce against a rail:

```
cfc.done:
LET rbounce%=(new.locx>=spot.x1%)   :REM check.for.bounce:
IF NOT rbounce% THEN check.lbounce
LET new.locx=spot.x1%+spot.x1%-new.locx
check.lbounce:
LET lbounce%=(new.locx<=spot.x%)
IF NOT lbounce% THEN check.bbounce
LET new.locx=spot.x%+spot.x%-new.locx
check.bbounce:
LET bbounce%=(new.locy>=spot.y1%)
IF NOT bbounce% THEN check.tbounce
LET new.locy=spot.y1%+spot.y1%-new.locy
check.tbounce:
LET tbounce%=(new.locy<=spot.y%)
IF NOT tbounce% THEN done.cfb
LET new.locy=spot.y%+spot.y%-new.locy
done.cfb:
IF rbounce% OR lbounce% THEN LET dx.b=-dx.b
IF bbounce% OR tbounce% THEN LET dy.b=-dy.b
```

A ball bounce is imminent if its next calculated position (new.locx, new.locy) is off the table. The status variables rbounce%, lbounce%, tbounce%, and bbounce% indicate whether the ball will hit the right, left, top, or bottom rail. After a right- or left-side bounce, the horizontal direction dx.b is reversed; after a top- or bottom-side bounce, the vertical direction dy.b is reversed.

After checking for a collision and a bounce, the program moves the ball to its new location:

```
IF trace.on% THEN new.pos   :REM move ball
PUT (locx(mvb%),locy(mvb%)),ball%(0,mvb%)   :REM erase from
      old pos'n
new.pos:
LET locx(mvb%)=new.locx
LET locy(mvb%)=new.locy
PUT (locx(mvb%),locy(mvb%)),ball%(0,mvb%)   :REM show at new
      pos'n
IF hit% THEN SOUND 110,.75
IF rbounce% OR lbounce% OR bbounce% OR tbounce% THEN SOUND
       200,.5
IF hit% THEN mvb%=3-mvb%
```

```
WEND
IF event%=1 THEN shoot
shoot.done:
CALL INITCURSOR
RETURN
```

If the tracer is on, the program skips the line that erases the ball from its old position. Immediately after the ball is shown in its new position, the program makes a sound if a hit or bounce was calculated.

If a hit occurred, the stationary ball becomes the moving ball (IF hit% then mvb%=3−mvb%).

These lines also include the termination of the roll-ball loop. If you have clicked the mouse button during the current roll-ball cycle, the program exits from the WHILE/WEND loop. In the case of a single click (event%=1), the program goes back to the beginning of the shoot procedure. In the case of a double-click, the program goes back to the control-button monitor routine.

Calculating the Angle of Motion

Given a horizontal increment delta.x% and a vertical increment delta.y%, the following subroutine calculates the resulting angle of motion:

```
find.angle:
LET a.ok%=yes%
LET quad.select%=SGN(delta.x%)+1+(SGN(delta.y%)+1)*3
ON quad.select%+1 GOTO
      q2q3,yneg,q1q4,xneg,origin,xpos,q2q3,ypos,q1q4
origin:
LET a.ok%=no%
GOTO fa.done:
xpos:
LET angle=0
GOTO fa.done
q1q4:
LET angle=ATN(delta.y%/delta.x%)
GOTO fa.done
ypos:
LET angle=qtr.circ
GOTO fa.done
q2q3:
```

```
LET angle=pi+ATN(delta.y%/delta.x%)
GOTO fa.done
xneg:
LET angle=pi
GOTO fa.done
yneg:
LET angle=3*qtr.circ
GOTO fa.done
fa.done:
RETURN
```

—Testing and Using the Program

You should be able nearly to duplicate the screens shown in Figures 7-1 and 7-3 through 7-6. Test all of the control-panel buttons. When shooting the cue ball, try aiming for the sides of the object ball. The object ball should bounce away at an angle just as in real billiards.

—Suggested Games

One of the simplest games for one or two players is Call the Shot. Each player starts with the same ball position (use the table spots to help fix a location). Before shooting, the player specifies which rails the ball will bounce off of en route to the object ball. The object of the game is to bounce off the most rails before hitting the ball; but remember, the player must specify the number and sequence of rails that will be hit.

Another game is Circles. The goal is to enclose the object ball in the path of the cue ball without hitting it. Play this game with the tracer on.

Finally, players may take turns at One-upmanship. Each player starts at level 1, meaning the player must hit the object ball after one bounce. Starting a turn with new random ball positions, the player tries to hit the object ball after the number of bounces corresponding to that player's current game level. After successfully hitting the object ball in a specified number of bounces, the player advances to the next level (keeping the latest ball positions). A player continues until he or she misses.

Chapter 8

Concentration

Concentration is usually played with a deck of cards. In this chapter, we present a program that allows you to do the same thing with a computer. Never mind how many decks of cards you can buy for the cost of a single computer—the computer version has unique advantages over its predecessor, such as automatic scoring, shuffling, and card-handling, and a far more interesting deck of cards (see Figure 8-1).

—Rules and Object of the Game

From one to four persons can play Concentration. A deck of playing cards is shuffled and then arranged face-down on a table. For a complete 54-card deck (including two "wild" or blank cards), a 6 × 9 layout is used. The computer handles these details.

The first player "turns over" two of the cards by pointing to each card with the cursor and clicking the mouse button. If the cards match or if either of the cards is wild, the player receives two points and the cards are removed from the table. Otherwise, the cards are turned over again, each in its original position. Play then passes to the next player. The game continues until no more matching pairs remain on the table. The player with the most points wins.

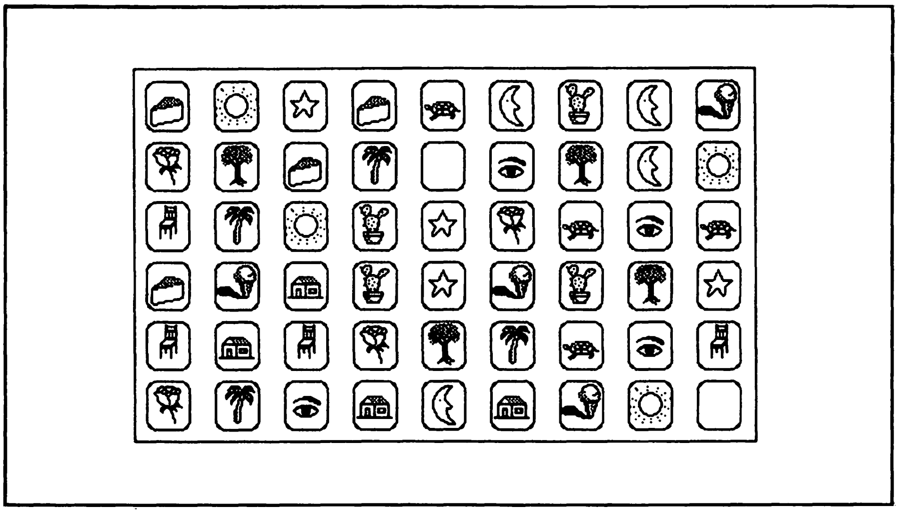


Figure 8-1. The deck of cards used in Concentration, shown face-up. To get these characters, the Cairo type font must be included in the system fonts of your BASIC startup disk

—The Program

The first block sets up the card-deck characters and certain other arrays.

```
DIM cv$(14), sc%(4), d%(54), pair.row%(2), pair.col%(2),
    cn%(2),card%(3), gray%(3)
FOR c%=1 TO 14
READ cv$(c%)
NEXT c%
DATA 1,2,3,5,7,8,@,%,^,*,),!,e," "
FOR j%=0 TO 3
LET gray%(j%)=&H55AA
NEXT j%
LET wc%=14
LET np%=1
LET yes%=(1=1)
RANDOMIZE TIMER
```

The array CV$( ) stores the 14 characters that appear on the faces of the playing cards.

Look at the DATA statement. It includes 14 characters, of which the last is a single blank space inside quotes. If you select the Cairo font and then attempt to print these characters, you'll get the characters shown in Figure 8-1. You may change any of the first 13 items in the DATA statement; however, the last item should always be a blank space, to produce the blank wild card.

Sc%( ) stores the players' scores. D%( ) keeps track of which card occupies each space on the table. Pair.row%( ) and pair.col%( ) record the location of the most recently selected pair of cards. Cn%( ) holds the same data in more compressed form. Card%( ) holds parameters used when drawing the cards, and gray%( ) holds data corresponding to the dotted pattern on the backs of the cards.

Wc% indicates which card character is wild. Np% is the initial setting for the number of players.

The next block of lines defines the card's appearance:

```
LET card.width%=26
LET card.length%=29
LET row.tab%=35
LET col.tab%=40
LET oval.x%=card.width%\2
LET oval.y%=card.length%\2
```

Cards are 26 × 29 dots and are placed at 35-dot increments horizontally and 40-dot increments vertically. The corners of the cards are rounded.

The following lines determine the size and location of the three windows used by the program:

```
LET w3.width%=5*72
LET w3.length%=3*72
LET w1.width%=w3.width%
LET bx.c%=(w1.width%-50)\2
LET w1.length%=(1+1/8)*72
LET w2.width%=(1+1/2)*72
LET w2.length%=w3.length%
LET w3.x%=1/4*72
LET w3.x1%=w3.x%+w3.width%
LET w3.y%=3/8*72
LET w3.y1%=w3.y%+w3.length%
LET w1.x%=w3.x%
LET w1.x1%=w3.x1%
```

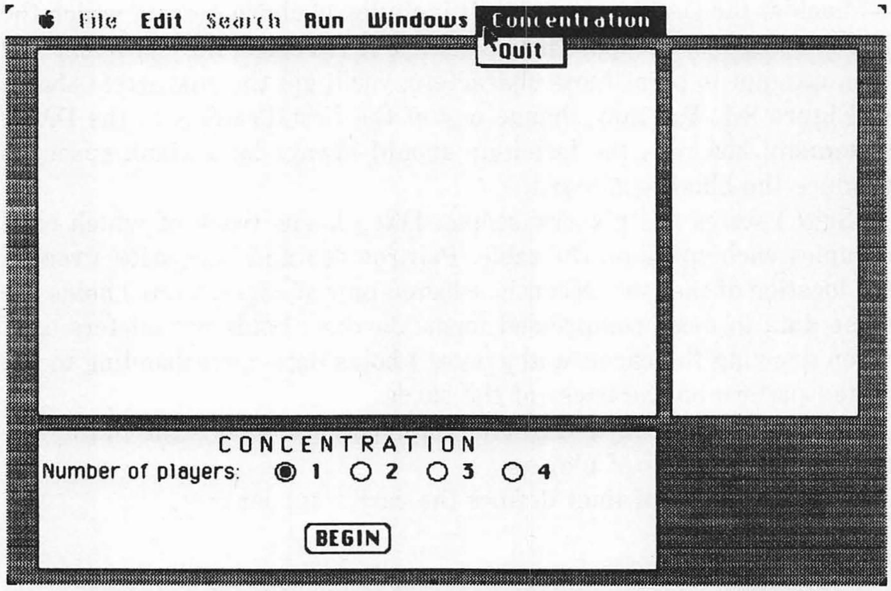

File Edit Search Run Windows Concentration
Quit
CONCENTRATION
Number of players: 1 2 3 4
BEGIN

Figure 8-2. The initial screen when you start Concentration

```
LET w1.y%=w3.y1%+1/8*72
LET w1.y1%=w1.y%+w1.length%
LET w2.x%=w1.x1%+1/8*72
LET w2.x1%=w2.x%+w2.width%
LET w2.y%=w3.y%
LET w2.y1%=w2.y%+w2.length%
```

Refer to Figure 8-2. The w-prefix variables refer to window 3 (the card table), window 2 (the scorebox), and window 1 (the dialog box).

Here are the lines that create the windows:

```
WINDOW 3,,(w3.x%,w3.y%)-(w3.x1%,w3.y1%),3 :REM 8-4
CALL TEXTMODE(1)
CALL TEXTSIZE(18)
CALL TEXTFONT(11)
WINDOW 1,,(w1.x%,w1.y%)-(w1.x1%,w1.y1%),3
CALL TEXTSIZE(12)
CALL TEXTFONT(1)
LET title1$="C O N C E N T R A T I O N"
LET title1.tab%=(w1.width%-WIDTH(title1$))\2
```

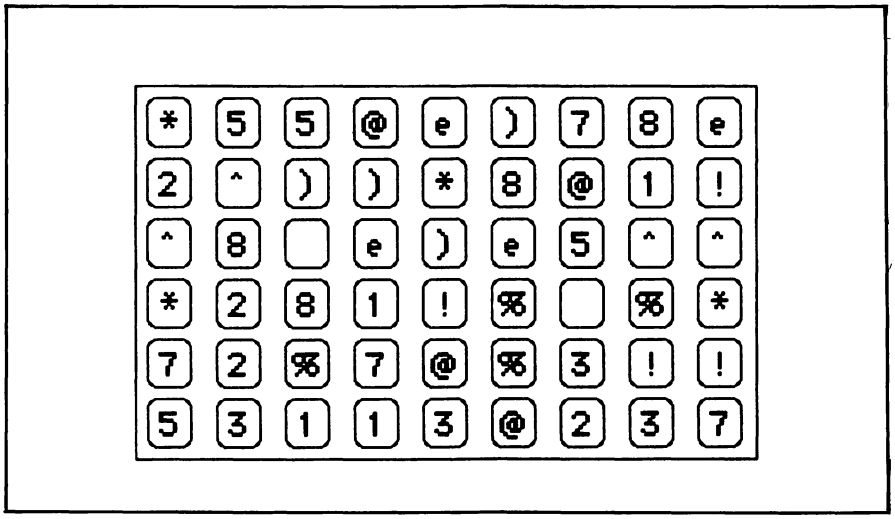


Figure 8-3. If your startup disk doesn't have the Cairo font, your deck of cards will look like this

```
WINDOW 2,,(w2.x%,w2.y%)-(w2.x1%,w2.y1%),3
CALL TEXTSIZE(12)
CALL TEXTFONT(1)
LET title2$="SCOREBOX"
LET title2.tab%=(w2.width%-WIDTH(title2$))\2
```

Each window has its own associated type font and type size. Notice that text font 11 (Cairo) is used in window 3. If your Macintosh BASIC disk does not include that type font, you can use Apple's Font Mover program to add that font to the disk. If you run the program without having the Cairo font available, the characters on the cards will be the same as those shown in Figure 8-3.

The following lines add a menu entry to the top bar:

```
MENU 6,0,1,"Concentration"
MENU 6,1,1,"Quit "
ON MENU GOSUB menu.rq
MENU ON
```

Setting Up the Windows

These lines set up the windows as shown in Figure 8-2:

```
new.game:
WINDOW 3
CLS
WINDOW 2
CLS
WINDOW 1
CLS
PRINT PTAB(title1.tab%);title1$
PRINT "Number of players:"
FOR b%=1 TO 4
bx%=(b%-1)*44+136
BUTTON b%,1-(b%=np%),STR$(b%),(bx%,18)-(bx%+32,30),3

NEXT b%
BUTTON 5,1,"BEGIN",(bx.c%,54)-(bx.c%+50,72)
```

Next comes a loop that waits for you to specify the number of players and press the BEGIN button:

```
hm.loop:
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
IF btn%=5 THEN hm.done
BUTTON np%,1
LET np%=btn%
BUTTON np%,2
GOTO hm.loop
hm.done:
FOR btn%=1 TO 5
BUTTON CLOSE btn%
NEXT btn%
```

Np% is the number of players. Pressing one of the radio-style buttons 1 through 4 changes np% accordingly. Pressing BEGIN causes the program to continue with the next program block.

Shuffling the Cards

The next block of lines shuffles the cards:

```
CLS
PRINT "Shuffling the cards..."
FOR c%=1 TO 54
LET d%(c%)=0
NEXT c%
FOR w%=1 TO 2
pick.wcloc:
LET c%=INT(RND*54)+1
IF d%(c%)>0 THEN pick.wcloc
LET d%(c%)=wc%
NEXT w%
FOR n%=1 TO 52
pick.cloc:
LET c%=INT(RND*54)+1
IF d%(c%)>0 THEN pick.cloc
LET d%(c%)=n% MOD 13 + 1
NEXT n%
CLS
```

First the program sets every array element to 0 (LET d%(c%)=0), which indicates that no card has been assigned to any location. Then the program places the two wild cards in randomly chosen positions in array d%( ) (LET d%(c%)=wc%).

Next the program shuffles the remaining 52 cards in the deck.

The program converts n% (which ranges from 1 to 52) into a value from 1 to 13, corresponding to the 13 card characters, and stores that value in the randomly selected location c% (LET d%(c%)=n% MOD 13 + 1).

Now that the cards are shuffled, the following lines place them face-down on the table:

```
WINDOW 3
FOR row%=1 TO 6
FOR col%=1 TO 9
GOSUB card.down
NEXT col%
NEXT row%
```

The card identities are stored in the single-dimension array d%( ) and are placed on the table in a two-dimensional arrangement. The following formula gives the correspondence between each card on the table and its location in d%( ):

$$\text{index in d\%( )} = (\text{row}-1) \times 6 + \text{col}$$

For instance, the identity of the card at row 5, column 3 is stored in d%( ) at location $(5-1) \times 6 + 3 = 27$.

The card.down subroutine places a face-down card at the table location row%,col%.

The Score Box

The next block of lines sets up the scorebox labels:

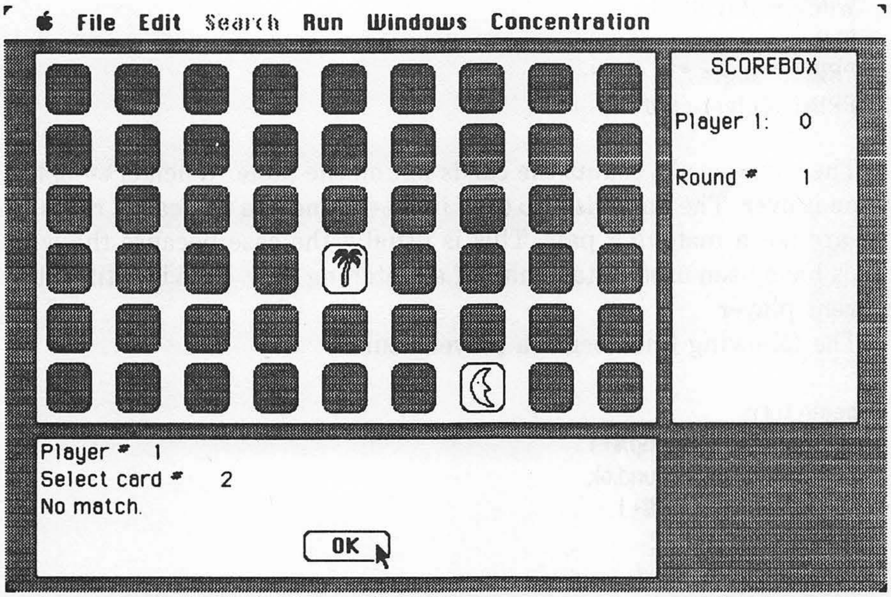
```
score.box:
WINDOW OUTPUT 2
CLS
PRINT PTAB(title2.tab%);title2$
FOR pn%=1 TO np%
LET sc%(pn%)=0
LOCATE 3+2*(pn%-1),1
PRINT USING "Player #:";pn%;
GOSUB update.scores
NEXT pn%
PRINT
PRINT "Round #"
```

The appearance of the scorebox varies with the number of players (compare the boxes in Figures 8-4 and 8-5).

Starting a Game

The next lines are executed at the start of a player's turn:

```
play.game:
LET c1%=54
LET pn%=0
LET round%=0
```



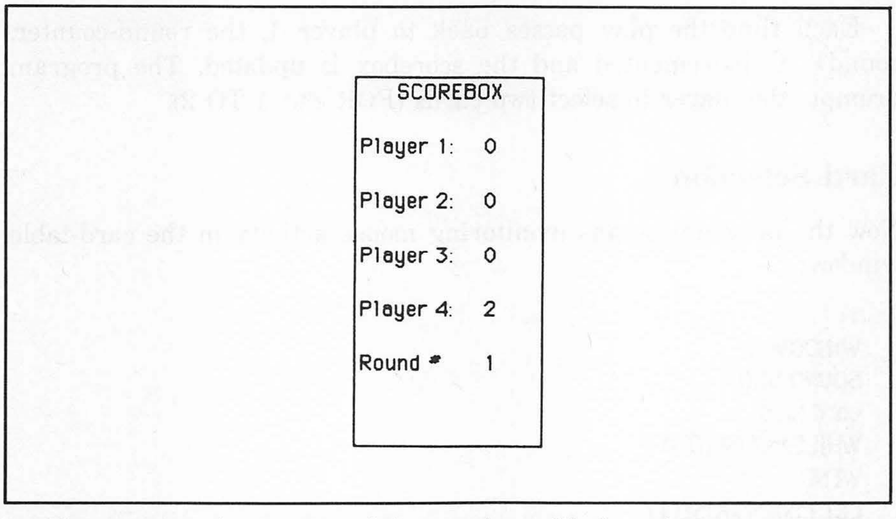
SCOREBOX

Player 1: 0

Round # 1

Player # 1
Select card # 2
No match.

Figure 8-4. Screen appearance after player 1's first turn



SCOREBOX

Player 1: 0

Player 2: 0

Player 3: 0

Player 4: 2

Round # 1

Figure 8-5. Appearance of the scorebox with four players

```
WINDOW OUTPUT 1
CLS
PRINT "Player #"
PRINT "Select card #";
```

The variable cl% counts the cards left on the table. When cl%=0, the game is over. The game is also over if cl%=2 and the two cards remaining are not a matching pair. This is usually the case because the wild cards have been used to take half of a matching pair. Pn% identifies the current player.

The following lines begin a player's turn:

```
begin.turn:
LET pn%=pn% MOD np%+1
IF pn%<>1 THEN round.ok
LET round%=round%+1
WINDOW OUTPUT 2
GOSUB update.round
WINDOW OUTPUT 1
round.ok:
LOCATE 1,9
PRINT USING "#";pn%
FOR c%=1 TO 2
LOCATE 2,13
PRINT USING "##";c%;
```

Each time the play passes back to player 1, the round-counter, round%, is incremented and the scorebox is updated. The program prompts the player to select two cards (FOR c%=1 TO 2).

Card Selection

Now the program begins monitoring mouse activity in the card-table window:

```
WINDOW 3
SOUND 550,1
card.loop:
WHILE MOUSE(0)<>1
WEND
LET mx%=MOUSE(1)
LET my%=MOUSE(2)
LET zone.x%=(mx%-6)\col.tab%+1
LET zone.y%=(my%-6)\row.tab%+1
IF zone.x%<1 OR zone.x%>9 OR zone.y%<1 OR zone.y%>6
     THEN card.loop
IF (mx%-6) MOD col.tab%>card.width% OR (my%-6) MOD
row.tab%>card.length% THEN card.loop
LET row%=zone.y%
LET col%=zone.x%
LET cr%=(row%-1)*9+col%
IF d%(cr%)=0 THEN card.loop
IF c%=2 AND cr%=cn%(1) THEN card.loop
LET cn%(c%)=cr%
LET pair.row%(c%)=row%
LET pair.col%(c%)=col%
GOSUB card.up
WINDOW OUTPUT 1
NEXT c%
```

The WHILE/WEND loop waits for the player to click the mouse. The next 11 lines after WEND check to see whether the player clicked the mouse on a card or not. Three types of invalid selections are possible: invalid row-column location (IF zone.x%<1 OR...); reference to a card already removed from the table (IF d%(cr%)=0...); and a request for the same card for card 1 and card 2 (IF c%=2 AND...). In case of any of these errors, the program returns to card.loop to wait for another click.

If the selection is valid, the program stores the card-pointer in cn%(c%), where c% equals 1 or 2. The program also stores the card's row and column address in pair.r%( ) and pair.c%( ). The selected card is turned face-up (GOSUB card.up).

The card-selection loop is repeated for the second card.

Evaluating a Selection

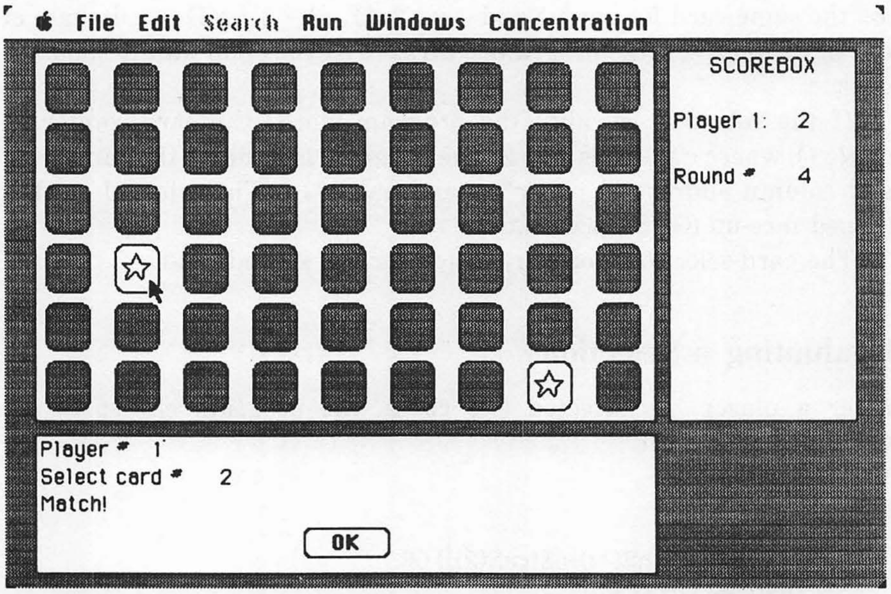
After a player has selected two cards, the program checks to see whether they match:

```
LOCATE 3,1
LET match%=(d%(cn%(1))=d%(cn%(2))) OR
     (d%(cn%(1))=wc%)
LET match%=match% OR (d%(cn%(2))=wc%)
```

```
IF NOT match% THEN no.match
PRINT "Match!   ";
FOR snd%=1 TO 4
SOUND snd%*110,1
NEXT snd%
LET sc%(pn%)=sc%(pn%)+2
WINDOW OUTPUT 2
GOSUB update.scores
WINDOW OUTPUT 1
LET cl%=cl%-2
LET game.over%=(cl%=0)
IF game.over% THEN PRINT "Game.over";: BEEP
GOTO end.turn
```

Figures 8-4 and 8-6 show the program's response to a non-matching pair and a matching pair.

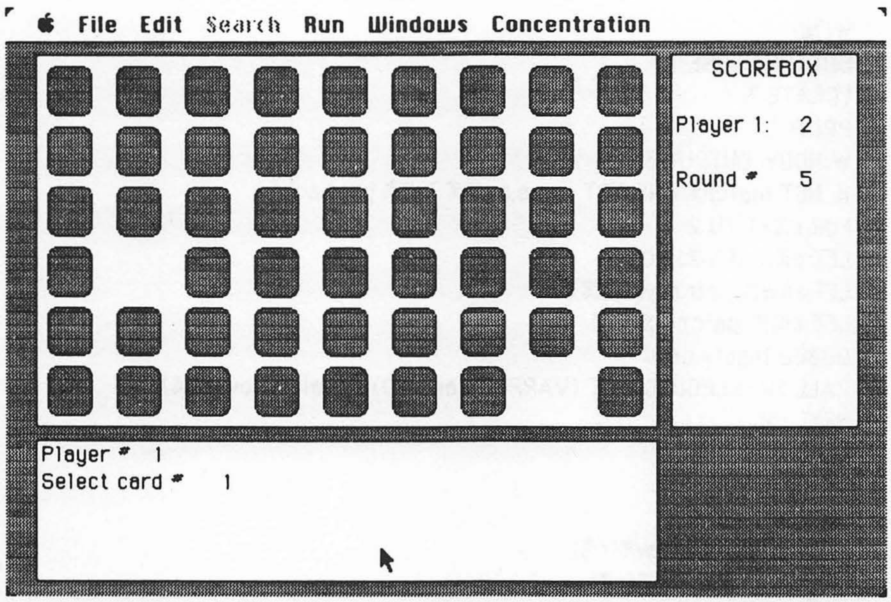
In case of a match, the program sounds an ascending sequence of notes and adds two to the current player's score. Since the matched cards will be removed from the table, the program also deducts two from cl%. If cl%=0, no cards remain and the game is over.



```
SCOREBOX

Player 1:   2

Round #     4
```

```
Player #   1
Select card #   2
Match!
```

Figure 8-6. Screen appearance after a matching pair is found



Player # 1
Select card # 1

Figure 8-7. The matching pair is removed from the table

Figure 8-7 shows the table after a matching pair has been removed. In the case of a non-match, these lines take over:

```
no.match
PRINT "No match.";
SOUND 220,2
SOUND 110,2
LET game.over%=(c1%=2)
IF game.over% THEN PRINT "Game.over";:BEEP
```

The program sounds a descending couplet and checks to see whether the cards selected were the last two on the table. If so, the game is over since no matching pairs are left.

After a player's selections have been evaluated, the following lines wait until the player presses a button to continue:

```
end.turn:
WINDOW 1
BUTTON 1,1,"OK",(bx.c%,54)-(bx.c%+50,72)
WHILE DIALOG(0)<>1
```

```
WEND
BUTTON CLOSE 1
LOCATE 3,1
PRINT "          "
WINDOW OUTPUT 3
IF NOT match% AND NOT game.over% THEN put.back
FOR c%=1 TO 2
LET d%(cn%(c%))=0
LET row%=pair.row%(c%)
LET col%=pair.col%(c%)
GOSUB locate.card
CALL ERASEROUNDRECT (VARPTR(card%(0)), oval.x%, oval.y%)
NEXT c%
GOTO cl.check
put.back:
FOR c%=1 TO 2
LET row%=pair.row%(c%)
LET col%=pair.col%(c%)
GOSUB card.down
NEXT c%
cl.check:
WINDOW 1
IF game.over% THEN new.game ELSE begin.turn
```

The OK button is shown in Figures 8-4 and 8-6. Once the player has pressed the OK button, the program puts the cards face-down again or removes them from the table (if they were a matching pair).

Subroutines

The program uses six subroutines. The first responds to a selection from the Concentration menu:

```
menu.rq:
IF MENU(0)<>6 THEN RETURN
IF MENU(1)<>1 THEN RETURN
WINDOW CLOSE 1
WINDOW CLOSE 2
WINDOW CLOSE 3
END
```

The next two subroutines update the scores and round-counter.

```
update.scores:
LOCATE 3+2*(pn%-1),9
PRINT USING "**";sc%(pn%)
RETURN
update.round:
LOCATE 5+2*(np%-1),9
PRINT USING "**";round%
RETURN
```

On entry to the update.scores subroutine, pn% is the player number. Here's the subroutine that turns a card face-up:

```
card.up:
GOSUB locate.card
CALL ERASEROUNDRECT (VARPTR(card%(0)), oval.x%, oval.y%)
LET c$=cv$(d%((row%-1)*9+col%))
LET c.tab%=(card.width%-WIDTH(c$))\2
CALL MOVETO(cx%+c.tab%,rx%+22)
PRINT USING "!"; c$;
CALL FRAMEROUNDRECT (VARPTR(card%(0)), oval.x%, oval.y%)
RETURN
```

Given card location row%, column%, the locate.card subroutine finds the actual card location in terms of window coordinates. It loads those coordinates into the array card%( ) so that the ERASEROUNDRECT routine can erase the card from the screen and the FRAMEROUNDRECT routine can redraw it with the character c$ showing.

The following lines put a card face-down on the table:

```
card.down:
GOSUB locate.card
CALL FILLROUNDRECT (VARPTR(card%(0)),
     oval.x%,oval.y%, VARPTR(gray%(0)))
CALL FRAMEROUNDRECT (VARPTR(card%(0)),
     oval.x%, oval.y%)
RETURN
```

As with the card.up subroutine, locate.card puts the necessary screen coordinates into the array card%( ). The program then puts a gray rounded rectangle at the appropriate location on the table.

Finally, here's the subroutine that loads the card-location parameters into the card%( ) array.

```
locate.card:
LET rx%=6+(row%-1)*row.tab%
LET cx%=6+(col%-1)*col.tab%
LET card%(0)=rx%
LET card%(1)=cx%
LET card%(2)=rx%+card.length%
LET card%(3)=cx%+card.width%
RETURN
```

Variables rx% and cx% are the actual screen coordinates of the upper left corner of the current card.

—Using the Program

The screens shown in Figures 8-2 through 8-7 are similar to the results you should get when you run Concentration on your computer.

The game must be played without pencil or paper. For fairness, everyone should get a look at the screen after each player completes his turn (and before he presses the OK button).

When playing Concentration solitaire, try to find all the pairs in as few rounds as possible. The next time you play, try to do it in even fewer rounds.

Chapter 9

The Codebreaker

This program lets your Macintosh show its smarts by competing against you in a game called the Codebreaker. With minor variations, the game is also known as "Bulls and Cows" and "Mastermind" (trademarked).

—Rules of the Game

In this two-player game, one player (the codemaker) makes up a secret code and the other player (the codebreaker) tries to guess the code. After each guess, the codemaker gives a score to the codebreaker, who uses this information to make another guess. The object of the game is to guess the code in as few tries as possible.

Codes consist of a sequence of four characters taken from the set A,B,C,D. For example, AAAA, ABCD, DCBA, and BAAB are all valid codes. There are 256 ways of combining the characters into codes.

Each guess receives two scores:

- The number of characters positioned correctly, called "hits."
- The number of characters positioned incorrectly, called "misses."

If a guess includes a character that is not found in the code, the character is not scored at all.

Table 9-1 gives several examples of scoring. Take a minute to study

Table 9-1. Sample Scoring for Secret Code BDBA

Guess	Score Hits	Score Misses	Comments
AAAA	1	0	The A in the rightmost position is a hit; the other A's don't count.
ABBB	1	2	The B second from the right is a hit; the A is a miss; one of the other B's is a miss; the remaining B doesn't count.
BCAB	1	2	The B in the leftmost position is a hit; one of the other B's and the A are misses; the C doesn't count.
DBAB	0	4	All four characters are misses, i.e., all are in the secret code but none is positioned as guessed.
BDBA	4	0	All four characters are hits.

the sample guesses and scores to be sure you understand the scoring system.

The Codebreaker program lets you play the role of codemaker or codebreaker. In the latter case, the program makes up secret codes and scores your guesses. When you take the role of codemaker, the program functions as the codebreaker. You type in your secret code, and the computer scores its own guesses. (Don't worry, the program doesn't cheat; the secret code is kept in a part of the program that the codebreaker never sees.)

You may be surprised to find that the program is an exceptionally good guesser. The process it uses is very systematic—no intuition or artificial intelligence is involved. Of course, you don't have to tell your friends that!

Two people can play this game by taking turns as the codebreaker and letting the computer score each player. The player who guesses the secret code in the fewest tries wins the round.

Figures 9-1 through 9-10 illustrate the operation of the program in its role as codemaker and codebreaker.

—Secrets of Codebreaking

Most players eventually come up with a system for guessing. Here's the Codebreaker's own method:

The program makes its first guess arbitrarily. It then gets the scores (number of hits and misses) and records that information.

For subsequent guesses, the program starts with a potential guess or "hypothesis" chosen from a list of all possible codes. It assumes the hypothesis is correct and scores each of its previous guesses against the hypothesis. If all its scores are consistent with the scores actually received, the program uses the hypothesis as its next guess. If any of the scores are different from the scores you provided, the program discards that hypothesis and gets another.

—The Program

The first block defines the arrays used in the program:

```
LET lg%=10
DIM wnd.w%(2), wnd.l%(2), wnd.x%(2), wnd.y%(2), wnd.x1%(2),
     wnd.y1%(2)
DIM btn.x%(3), btn.y%(3), btn.x1%(3), btn.y1%(3)
DIM fld.x%(2), fld.y%(2), fld.x1%(2), fld.y1%(2)
DIM p$(256), gu$(lg%), s1%(lg%), s2%(lg%), pl$(2)
```

Lg% is the maximum number of guesses you are allowed before the computer reveals the secret code. The arrays prefixed by wnd., btn., and fld. contain parameters for windows, buttons, and edit fields.

P$( ) contains all possible codes. Gu$( ) contains the guesses that the codebreaker (you or the computer) makes. S1%( ) and s2%( ) keep track of the scoring for each guess: s1%( ) stores hits, and s2% stores misses.

For instance, gu$(1) stores the first guess; s1%(1) stores the number of hits assigned to that guess, and s2%(1) stores the number of misses.

Loading the Parameters

The next block of lines loads parameters into the window arrays:

```
FOR n%=1 TO 2
READ inches.wide,inches.long,ulcx,ulcy
LET wnd.w%(n%)=inches.wide*72
LET wnd.l%(n%)=inches.long*72
LET wnd.x%(n%)=ulcx*72
LET wnd.y%(n%)=ulcy*72
LET wnd.x1%(n%)=wnd.x%(n%)+wnd.w%(n%)
LET wnd.y1%(n%)=wnd.y%(n%)+wnd.l%(n%)
NEXT n%
REM  wide  long  left   top
DATA 3.00, 3.625, 0.375, 0.50
DATA 3.00, 3.625, 3.750, 0.50
```

The following lines serve the same function for the button arrays:

```
FOR n%=1 TO 3
READ inches.wide,inches.long,h.zone,v.zone,b.type%(n%)
LET btn.x%(n%)=(wnd.w%(1)-inches.wide*72)*h.zone
LET btn.y%(n%)=(wnd.l%(1)-inches.long*72)*v.zone
LET btn.x1%(n%)=btn.x%(n%)+inches.wide*72
LET btn.y1%(n%)=btn.y%(n%)+inches.long*72
NEXT n%
REM  wide  long  h.zone v.zone type
DATA 1.000, 0.333, 0.500, 0.9375, 1
DATA 0.667, 0.208, 0.333, 0.3750, 3
DATA 0.667, 0.208, 0.667, 0.3750, 3
```

H.zone% and v.zone% indicate the relative horizontal and vertical position of a button within window 1.

And here are the corresponding lines for the edit fields:

```
FOR n%=1 TO 2
READ inches.wide,inches.long,h.zone,v.zone
LET fld.x%(n%)=(wnd.w%(1)-inches.wide*72)*h.zone
LET fld.y%(n%)=(wnd.l%(1)-inches.long*72)*v.zone
LET fld.x1%(n%)=fld.x%(n%)+inches.wide*72
LET fld.y1%(n%)=fld.y%(n%)+inches.long*72
NEXT n%
REM  wide  long  h.zone  v.zone
DATA 1.00, 0.208, 0.500, 0.333
DATA 1.00, 0.208, 0.500, 0.500
```

Program Constants

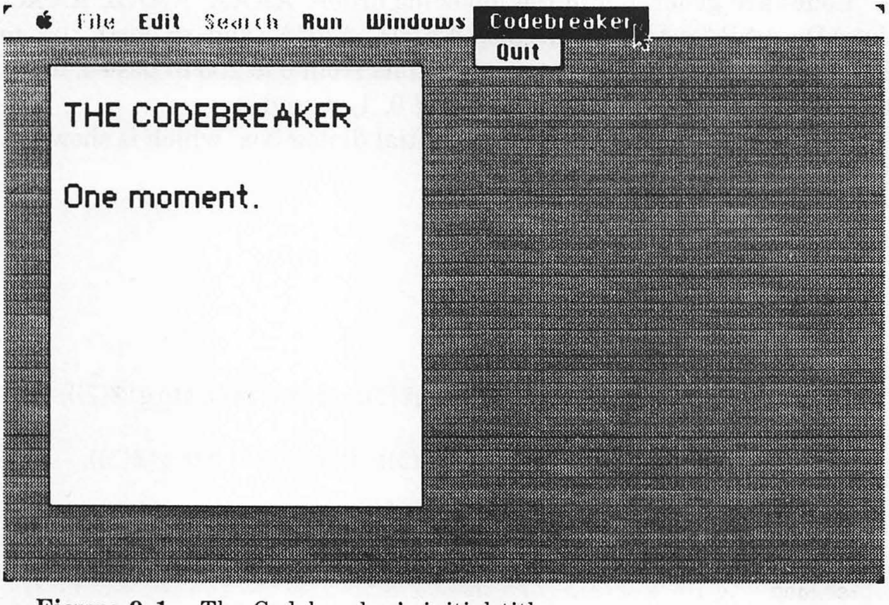
The following lines set up certain values that do not change during program operation:

```
LET pl$(1)="You"
LET pl$(2)="Mac"
LET yes%=(1=1)
LET no%=(1=0)
LET dg$="ABCD"
LET one.space$=" ": REM one space inside quotes
LET one.x$="X": REM one X inside quotes
LET qt$=CHR$(34)  :REM double quote
```

Pl$( ) stores the names of the two players: "Mac" and "You." The next lines activate the Codebreaker menu shown in Figure 9-1:

```
MENU 6,0,1,"Codebreaker"
MENU 6,1,1,"Quit "
ON MENU GOSUB menu.rq
MENU ON
```

The menu offers only one command: Quit.



THE CODEBREAKER

One moment.

Figure 9-1. The Codebreaker's initial title screen

Generating All Possible Codes

The following lines print the title screen shown in Figure 9-1 and also generate all possible codes, storing them in array p$( ).

```
RANDOMIZE TIMER
LET cb%=1
WINDOW 1,,(wnd.x%(1),wnd.y%(1))-(wnd.x1%(1),wnd.y1%(1)),3
CLS
CALL TEXTSIZE(18)
PRINT
PRINT " THE CODEBREAKER"
PRINT
PRINT " One moment."
CALL TEXTSIZE(12)
FOR p1%=1 TO 4
FOR p2%=1 TO 4
FOR p3%=1 TO 4
FOR p4%=1 TO 4
LET ix%=(p1%-1)*64+(p2%-1)*16+(p3%-1)*4+p4%
LET p$(ix%)=MID$(dg$,p1%,1) + MID$(dg$,p2%,1) + MID$(dg$,p3%,1) +
      MID$(dg$,p4%,1)
NEXT p4%,p3%,p2%,p1%
```

Codes are generated in the following order: AAAA, AAAB, AAAC, AAAD, AABA, AABB, AABC, AABD, AACA, and so forth, up to DDDD. In effect, the computer just counts from 0 to 255 in base 4, using the "digits" A, B, C, and D instead of 0, 1, 2, and 3.

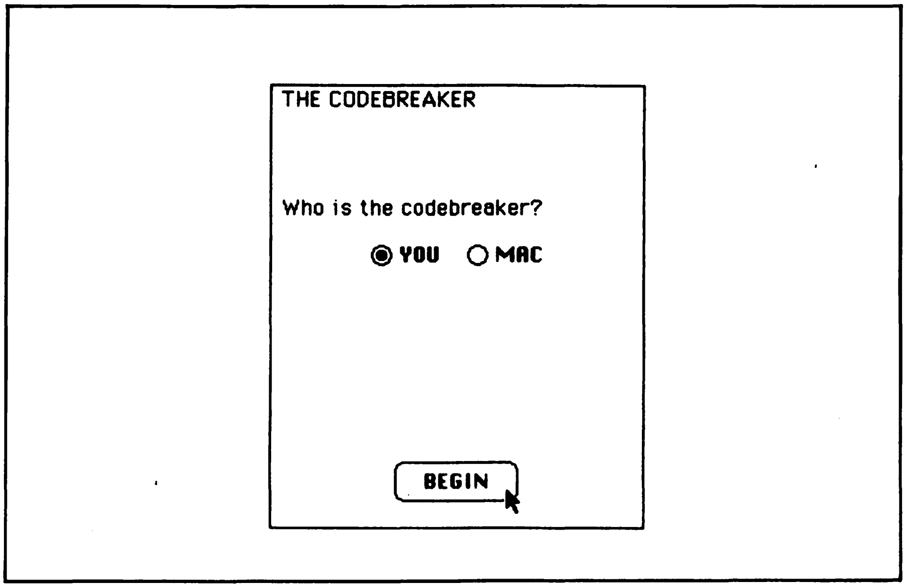
The following lines produce the initial dialog box, which is shown in Figure 9-2:

```
select.codebreaker:
CLS
PRINT " THE CODEBREAKER"
LOCATE 5,1
PRINT " Who is the codebreaker?"
BUTTON 1,3-cb%,"YOU",(btn.x%(2),btn.y%(2))- (btn.x1%(2), btn.y1%(2)),
      b.type%(2)
BUTTON 2,cb%,"MAC",(btn.x%(3),btn.y%(3))- (btn.x1%(3), btn.y1%(3)),
      b.type%(3)
BUTTON 3,1,"BEGIN",(btn.x%(1),btn.y%(1))- (btn.x1%(1),btn.y1%(1)),
      b.type%(1)
sc.loop:
```

```
GOSUB wait.entry
IF event%=6 THEN start
LET btn%=DIALOG(1)
ON btn% GOTO roles,roles,start
roles:
LET cb%=btn%
BUTTON 1,3-cb%
BUTTON 2,cb%
GOTO sc.loop
start:
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON CLOSE 3
ON cb% GOTO you.guess,mac.guess
```

The variable cb%, used in the BUTTON statements, identifies the codebreaker; 1=You, 2=Mac (the Macintosh).

The program gives you a chance to change the codebreaker (initially set to You) before you press BEGIN to start the game.


THE CODEBREAKER

Who is the codebreaker?

Figure 9-2. The Codebreaker selection menu

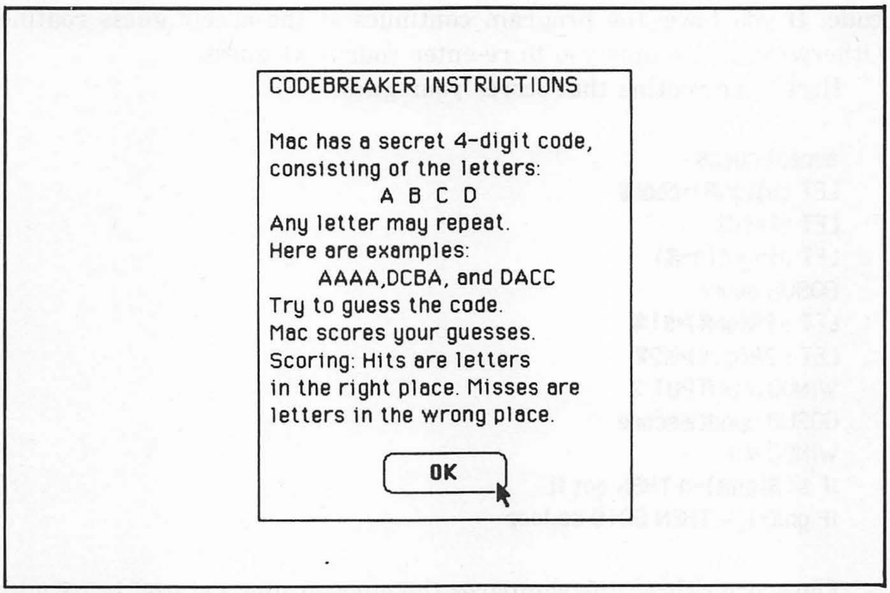
You as Codebreaker

In the next block of lines, the computer randomly selects a secret code, presents instructions for guessing, and prompts you to enter your first guess:

```
you.guess:
LET gn%=0
LET cr%=RND*256+1
LET cd$=p$(cr%)
GOSUB cb.instructions
WINDOW 2,,(wnd.x%(2),wnd.y%(2))-(wnd.x1%(2),wnd.y1%(2)),3
GOSUB label.scorebox
WINDOW 1
cb.loop:
CLS
PRINT " CODEBREAKER:"
LET gn%=gn%+1
LOCATE 5,1
PRINT " Enter guess number";gn%
EDIT FIELD 1,"",(fld.x%(1),fld.y%(1))-(fld.x1%(1),fld.y1%(1))
BUTTON 1,1,"OK", (btn.x%(1), btn.y%(1))- (btn.x1%(1),
      btn.y1%(1)), b.type%(1)
GOSUB wait.entry
LET code$=UCASE$(EDIT$(1))
EDIT FIELD CLOSE 1
BUTTON CLOSE 1
GOSUB check.code
IF code.ok% THEN accept.guess
LET gn%=gn%-1
GOTO cb.loop
```

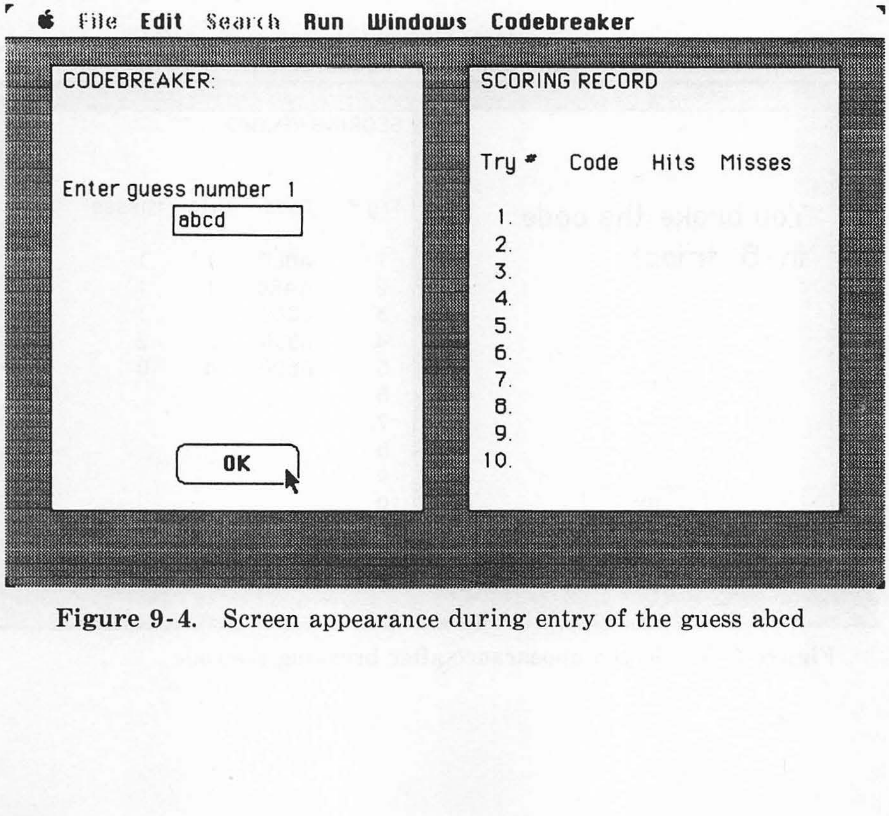
Gn% stores the latest guess number and is set to 0 before you make your first guess. Cr% is a random number from 1 to 256. Cd$=P$(cr%) is the computer's secret code.

The cb.instructions subroutine prints the instruction box shown in Figure 9-3. The label.scorebox subroutine identifies the rows and columns of the scoring table in window 2 (the right-hand window in Figure 9-4). The program prompts you to enter the next guess and then provides an edit field and an OK button. When you press the OK button, the check.code subroutine makes sure you have entered a valid


CODEBREAKER INSTRUCTIONS

Mac has a secret 4-digit code,
consisting of the letters:
A B C D
Any letter may repeat.
Here are examples:
AAAA,DCBA, and DACC
Try to guess the code.
Mac scores your guesses.
Scoring: Hits are letters
in the right place. Misses are
letters in the wrong place.

Figure 9-3. Instructions for playing Codebreaker



CODEBREAKER:

Enter guess number 1
abcd

SCORING RECORD

```
Try #   Code   Hits   Misses
  1.
  2.
  3.
  4.
  5.
  6.
  7.
  8.
  9.
 10.
```

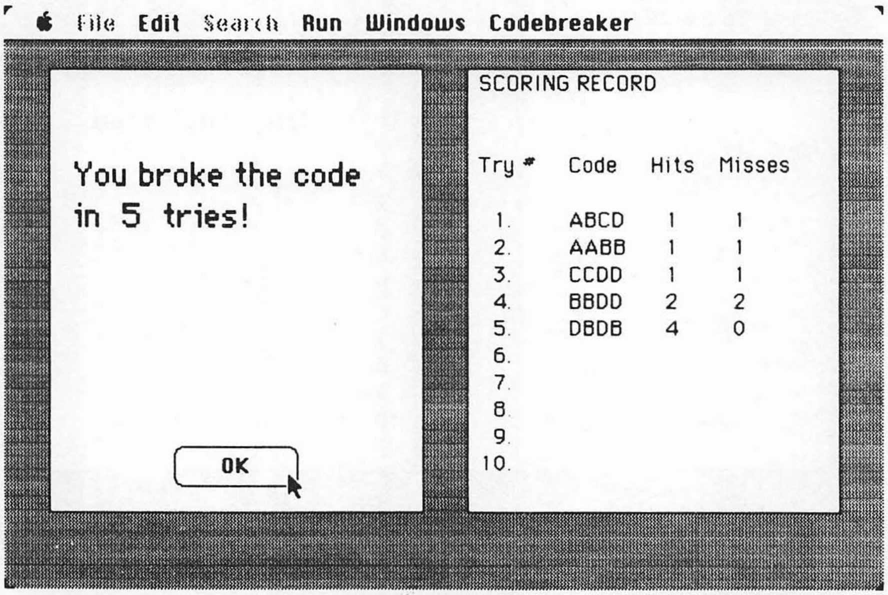
Figure 9-4. Screen appearance during entry of the guess abcd

code. If you have, the program continues at the accept.guess routine. Otherwise, it prompts you to re-enter your next guess.

Here's the routine that scores your guess:

```
accept.guess:
LET gu$(gn%)=code$
LET a$=cd$
LET q$=gu$(gn%)
GOSUB score
LET s1%(gn%)=s1%
LET s2%(gn%)=s2%
WINDOW OUTPUT 2
GOSUB update.score
WINDOW 1
IF s1%(gn%)=4 THEN got.it
IF gn%<lg% THEN GOTO cb.loop
```

The score subroutine compares the current guess stored in q$ with the secret code stored in a$. Upon return from that subroutine, s1% is the number of hits and s2% is the number of misses.



You broke the code in 5 tries!

```
SCORING RECORD

Try #   Code   Hits   Misses

 1.     ABCD    1      1
 2.     AABB    1      1
 3.     CCDD    1      1
 4.     BBDD    2      2
 5.     DBDB    4      0
 6.
 7.
 8.
 9.
10.
```

Figure 9-5. Screen appearance after breaking the code

The program then updates the scorebox. If you have guessed the code (s1%(gn%)=4), the program goes to the got.it subroutine. Otherwise, if you have more tries remaining, the program prompts you to enter another guess.

The following block takes over when you guess correctly or run out of tries:

```
nmt:
CLS
SOUND 440,2
SOUND 110,2
CALL TEXTSIZE(18)
LOCATE 3,1
PRINT " No more tries left."
PRINT " The secret code is"
PRINT "         ";cd$
CALL TEXTSIZE(12)
GOTO end.round
got.it:
CLS
FOR f%=1 TO 4
SOUND f%*110,1
NEXT f%
CALL TEXTSIZE(18)
LOCATE 3,1
PRINT TAB(2);pl$(cb%);" broke the code"
PRINT TAB(2)"in"; gn%; "tries!"
CALL TEXTSIZE(12)
end.round:
GOSUB wait.ok
WINDOW CLOSE 2
GOTO select.codebreaker
```

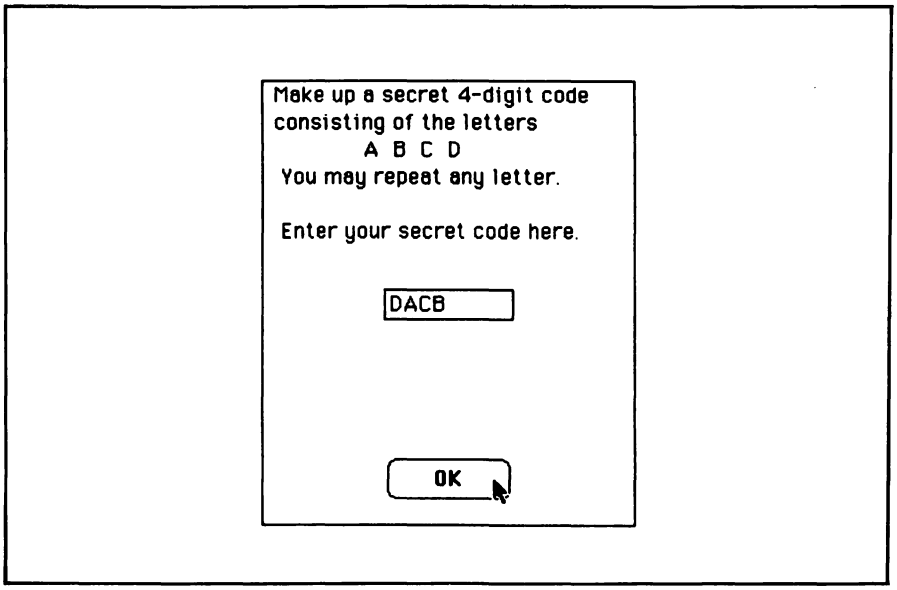
In the case of no more tries left, the program sounds a descending couplet; in the case of a correct answer, the program sounds an ascending sequence of notes.

The end.round routine waits for you to press the OK button, then closes the scorebox window and starts over with the menu shown in Figure 9-2.

Mac as Codebreaker

The next block of lines prints the codemaker instructions that are shown in Figure 9-6:

```
mac.guess:
CLS
PRINT " Make up a secret 4-digit code"
PRINT " consisting of the letters"
PRINT "          A B C D"
PRINT "  You may repeat any letter."
PRINT
PRINT "  Enter your secret code here."
EDIT FIELD 1,"",(fld.x%(2),fld.y%(2))-(fld.x1%(2),fld.y1%(2))
BUTTON 1,1,"OK",(btn.x%(1), btn.y%(1))- (btn.x1%(1),
     btn.y1%(1)), b.type%(1)
GOSUB wait.entry
LET code$=UCASE$(EDIT$(1))
EDIT FIELD CLOSE 1
BUTTON CLOSE 1
GOSUB check.code
IF code.ok% THEN accept.code ELSE mac.guess
```

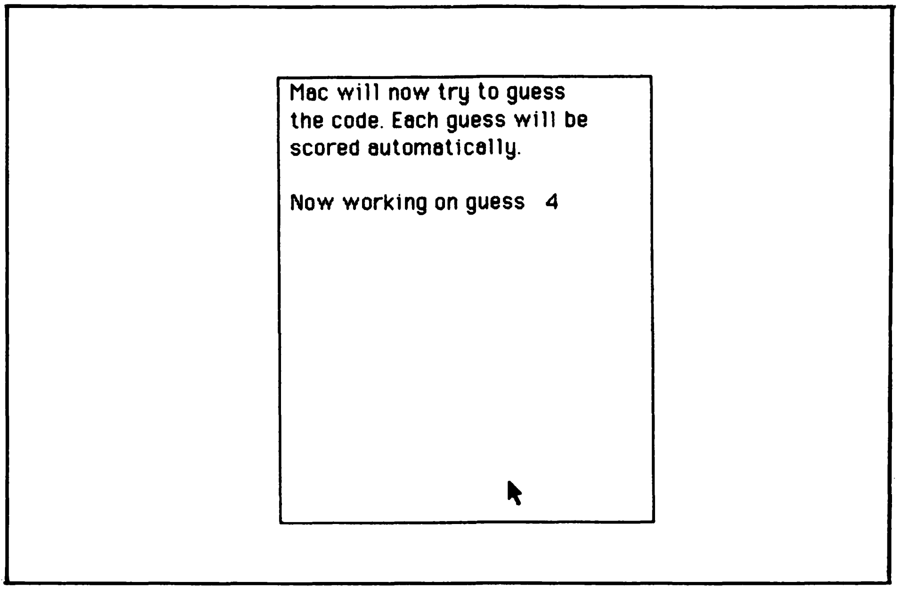


```
Make up a secret 4-digit code
consisting of the letters
          A B C D
 You may repeat any letter.

 Enter your secret code here.
```

Figure 9-6. Instructions for entering a secret code

After you enter a valid secret code, these next lines store it away for use in scoring and then display the status indicator box that is shown in Figure 9-7:

```
accept.code:
LET cd$=code$
WINDOW 2,,(wnd.x%(2),wnd.y%(2))-(wnd.x1%(2),wnd.y1%(2)),3
GOSUB label.scorebox
WINDOW 1
CLS
PRINT " Mac will now try to guess"
PRINT " the code. Each guess will be"
PRINT " scored automatically."
PRINT
PRINT " Now working on guess 1"
LET gn%=1
LET pn%=1
mg.loop:
LET gu$(gn%)=p$(pn%)
LET a$=cd$
LET q$=gu$(gn%)
```



```
Mac will now try to guess
the code. Each guess will be
scored automatically.

Now working on guess   4
```

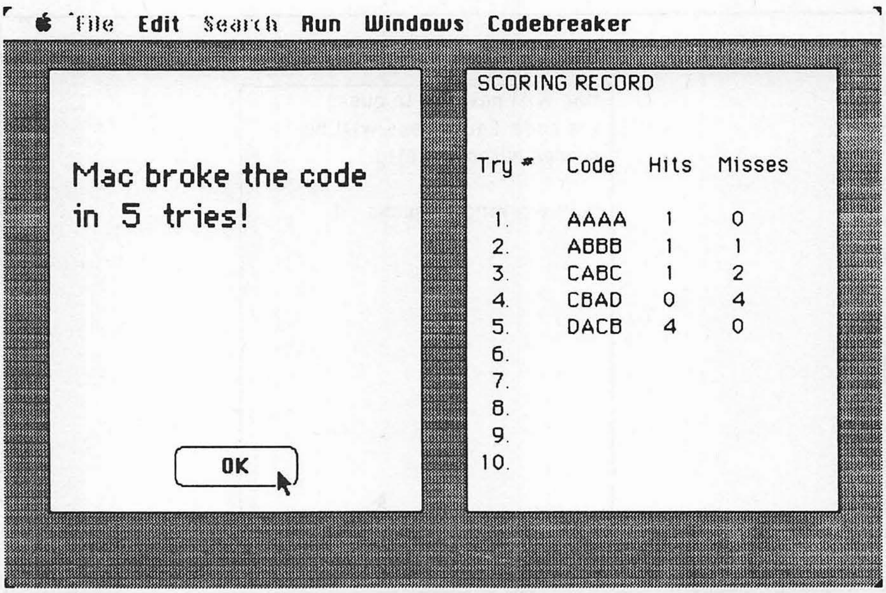
Figure 9-7. Status indicator box

```
GOSUB score
LET s1%(gn%)=s1%
LET s2%(gn%)=s2%
WINDOW OUTPUT 2
LOCATE 5+gn%,8
PRINT gu$(gn%);TAB(14); s1%(gn%); TAB(19); s2%(gn%)
WINDOW 1
IF s1%=4 THEN got.it
IF gn%=lg% THEN nmt
LOCATE 5,19
PRINT USING "##";gn%+1
```

Now the computer is ready to make its first guess. Gn% is the guess number, initially set to 1 for the program's first guess. Pn%, the "pattern number," keeps track of the number of patterns (codes) the program has tried already.

Initially, pn%=1 since the program starts with the first pattern in the array P$( ).

The program scores its own guess just as it scores your guesses. Unless it has guessed correctly or run out of guesses, the program



```
Mac broke the code
in 5 tries!
```

```
SCORING RECORD

Try #    Code    Hits    Misses

1.       AAAA    1       0
2.       ABBB    1       1
3.       CABC    1       2
4.       CBAD    0       4
5.       DACB    4       0
6.
7.
8.
9.
10.
```

Figure 9-8. Screen appearance after Mac breaks the code

makes its next guess according to the following procedure:

```
new.guess:
LET pn%=pn%+1
IF pn%>256 THEN PRINT "Error in scoring routine.": STOP
LET fl%=no%
FOR ih%=1 TO gn%
LET q$=gu$(ih%)
LET a$=p$(pn%)
GOSUB score
IF s1%=s1%(ih%) AND s2%=s2%(ih%) THEN nxt.ih
LET ih%=gn%
LET fl%=yes%
nxt.ih:
NEXT ih%
IF fl% THEN new.guess
LET gn%=gn%+1
GOTO mg.loop
```

First the program increments the pattern number. When pn%>256, all patterns have been tried without success; hence a scoring error has been made. In that case, you have probably entered the scoring subroutine incorrectly.

P$(pn%) becomes the computer's next hypothesis.

The computer tests its hypothesis by reviewing the previous guesses, scoring each guess under the assumption that the hypothesis is correct, and comparing the resultant scores with the scores actually received.

Fl% is a flag indicating whether the hypothesis conflicts with the scoring in previous guesses. Whenever a conflict is found, the program rejects the hypothesis and then moves on to the next one (IF fl% THEN new.guess). If a hypothesis produces no conflicts, it is accepted and used as the next guess.

Scoring Subroutine

Here's the routine that compares a secret code with a guess:

```
score:
LET s1%=0
LET s2%=0
FOR j%=1 TO 4
IF MID$(q$,j%,1)<>MID$(a$,j%,1) THEN s1.next:
```

```
LET s1%=s1%+1
MID$(a$,j%,1)=one.space$
MID$(q$,j%,1)=one.x$
s1.next:
NEXT j%
FOR j%=1 TO 4
LET f%=INSTR(1,a$,MID$(q$,j%,1))
IF f%=0 THEN s2.next
LET s2%=s2%+1
MID$(a$,f%,1)=one.space$
s2.next:
NEXT j%
RETURN
```

On entry to this subroutine, a$ contains the secret code and q$ contains the guess. The program compares each character in a$ with the corresponding character in q$. Whenever a match is found, the program increments the hit counter s1%. In this case, the program must blot out the character that was a hit, so that it won't affect the scoring of misses later on. It replaces the hit character in a$ with a space, and the hit character in q$ with an "X."

The second FOR/NEXT loop examines each character in the guess q$, to see if that character can be found anywhere in the secret code a$. Remember, the hit characters have already been blotted out from both Q$ and A$. Each time a character is found, the program increments the "miss" counter s2% and blots out from A$ the character counted as a miss.

Scorebox Labeling and Instructions

Here's the subroutine to update the scorebox:

```
update.score:
LOCATE 5+gn%,8
PRINT gu$(gn%);TAB(14); s1%(gn%); TAB(19); s2%(gn%)
RETURN
```

Gn% is the current guess number.

The following lines print the instructions shown in Figure 9-3:

```
cb.instructions:
CLS
PRINT " CODEBREAKER INSTRUCTIONS"
```

```
PRINT
PRINT " Mac has a secret 4-digit code,"
PRINT " consisting of the letters:"
PRINT "              A B C D"
PRINT " Any letter may repeat."
PRINT " Here are examples:"
PRINT "        AAAA,DCBA, and DACC"
PRINT " Try to guess the code."
PRINT " Mac scores your guesses."
PRINT " Scoring: Hits are letters"
PRINT " in the right place. Misses are"
PRINT " letters in the wrong place."
GOSUB wait.ok:
RETURN
```

The next lines print the scorebox labels:

```
label.scorebox:
CLS
PRINT " SCORING RECORD"
PRINT
PRINT
PRINT " Try #"; TAB(8); "Code"; TAB(14); "Hits"; TAB(19);"Misses"
PRINT
FOR rec%=1 TO lg%
PRINT USING " ##.";rec%
NEXT rec%
RETURN
```

Check-Code Subroutine

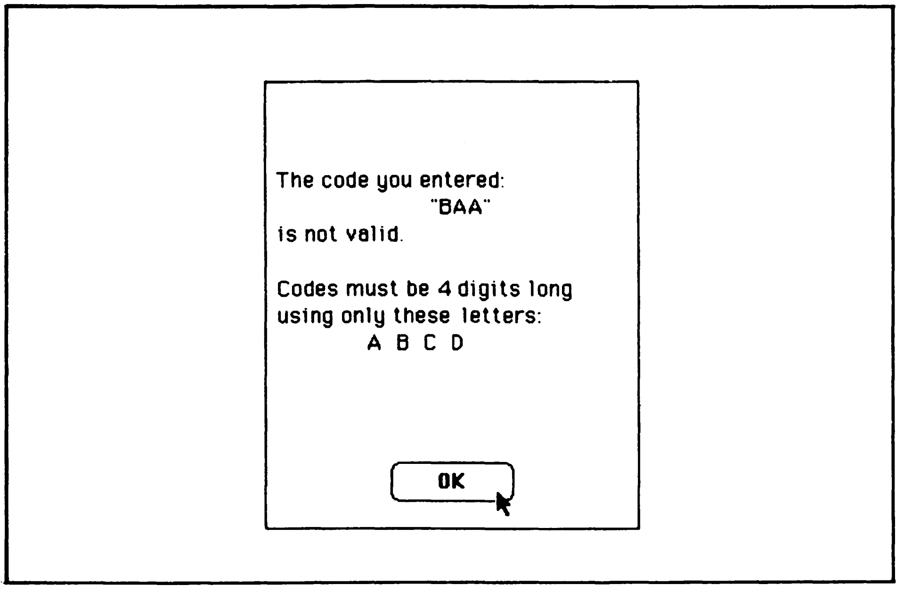
Whenever a code is entered from the keyboard, the following subroutine ensures that it is valid:

```
check.code:
LET code.ok%=(LEN(code$)=4)
IF  NOT code.ok% THEN cc.error
FOR I%=1 TO 4
IF INSTR(1,dg$,MID$(code$,I%,1))=0 THEN  code.ok%=no%
NEXT I%
```

```
IF code.ok% THEN cc.done
cc.error:
BEEP
CLS
LOCATE 4,1
PRINT " The code you entered:"
PRINT PTAB((wnd.w%(1)-WIDTH(code$))/2);qt$;code$;qt$
PRINT " is not valid."
PRINT
PRINT " Codes must be 4 digits long"
PRINT " using only these letters:
PRINT "          A B C D"
GOSUB wait.ok
cc.done:
RETURN
```

The length of the entry must be 4 and the code must consist solely of the letters A, B, C, and D.

In the case of an invalid code entry, the program prints the dialog box shown in Figure 9-9.



```
The code you entered:
              "BAA"
is not valid.

Codes must be 4 digits long
using only these letters:
        A B C D
```

Figure 9-9. Result of entering an invalid code as a guess or as a secret code

Auxiliary Subroutines

The next two subroutines provide pause functions:

```
wait.ok:
BUTTON 1,1,"OK", (btn.x%(1), btn.y%(1))- (btn.x1%(1), btn.y1%(1)),
      b.type%(1)
GOSUB wait.entry
BUTTON CLOSE 1
RETURN
wait.entry:
LET event%=DIALOG(0)
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%<>1 AND event%<>6 THEN wait.entry
RETURN
```

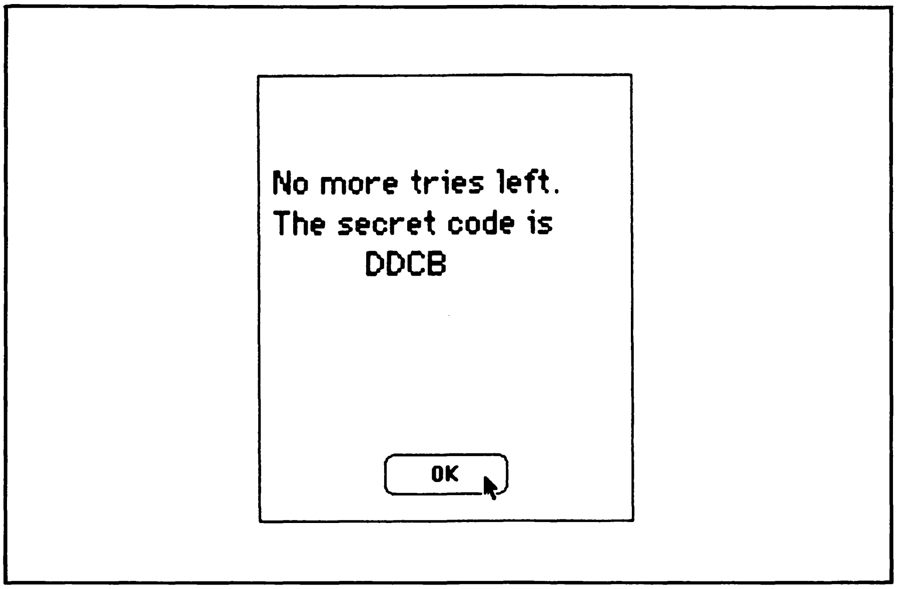

No more tries left.
The secret code is
DDCB

Figure 9-10. The Mac prints this message if you use all 10 tries without guessing the code

The first subroutine, wait.ok, puts an OK button on the screen and waits for you to press it. The second subroutine, wait.event, simply waits for you to press a button or the ENTER key (IF event<>1 AND event%<>6 THEN wait.entry).

Menu Requests

The last block of the program implements the Quit command in the Codebreaker menu shown in Figure 9-1.

```
menu.rq:
IF MENU(0)<>6 AND MENU(1)<>1 THEN RETURN
WINDOW CLOSE 1
WINDOW CLOSE 2
END
```

—Testing and Using the Program

After entering the entire program and eliminating all typographical errors, test the program, as codebreaker and as codemaker.

You should be able to get screens similar to the ones in this chapter.

When the program is running correctly, it will usually guess your secret code within four to six tries. The number of guesses required is determined by where the secret code is in the computer's internal list of codes P$( ). With a little experimentation, you can find out which secret codes always take the longest for the computer to find.

One interesting game to play is first to let the computer make up a code for you to break and see how long it takes you. Then let the computer try to break the same secret code.

This chapter is adapted from "The Code Breaker," by George Stewart, appearing in the December 1982 issue of *Popular Computing* magazine. Copyright 1982 Byte Publications, Inc. Used with the permission of Byte Publications, Inc.

Chapter 10

Tic-Tac-Toe

Although the rules and strategies of tic-tac-toe are simple, getting your computer to play well is no simple task. In this chapter, we teach your Macintosh to play the game to a win or draw almost every time. Compared to a good human player, the tic-tac-toe program's only weakness is that it occasionally settles for a draw when a victory is possible.

In addition to making your computer a passable tic-tac-toe player, the program exemplifies three techniques that are just as applicable to more complex games such as checkers and chess:

- Prepared opening moves.
- "Look-ahead"—checking the consequences of a proposed move by looking ahead to subsequent moves.
- Heuristics—selection of moves based on general principles of good strategy.

—Rules and Strategy

Before explaining the program's operation, we'll review the rules, object, and strategies of the game.

Tic-tac-toe is played on a 3 × 3 grid. Two players take turns marking cells on the grid. The first player marks with an X and the second player marks with an O.

The first player to place three of his marks (X's or O's) in a row, column, or diagonal wins. If all the cells are filled without either player winning, the game is a tie (see Figure 10-1). Before each subsequent game, players reverse their playing order so that the second player becomes the starting player and vice versa.

The lowest level of strategy for the game involves three steps:

1. If you can win on your next turn, do so.
2. If your opponent can win on his next turn, block him.
3. If neither condition is true, take any available cell.

It doesn't take a human player long to come up with some improvements or refinements of step 3. Good strategy is based on the idea of the *trap.* A trap is a mark that gives you two winning opportunities for your next turn (see Figure 10-2). Your opponent will be able to block only one of these on his next turn, so that when your turn comes around again, you'll still have one winning opportunity. Conversely, to avoid defeat at tic-tac-toe, you must prevent your opponent from setting such a trap (see Figure 10-3).

Preventing traps is not always easy. In some cases, you must look two

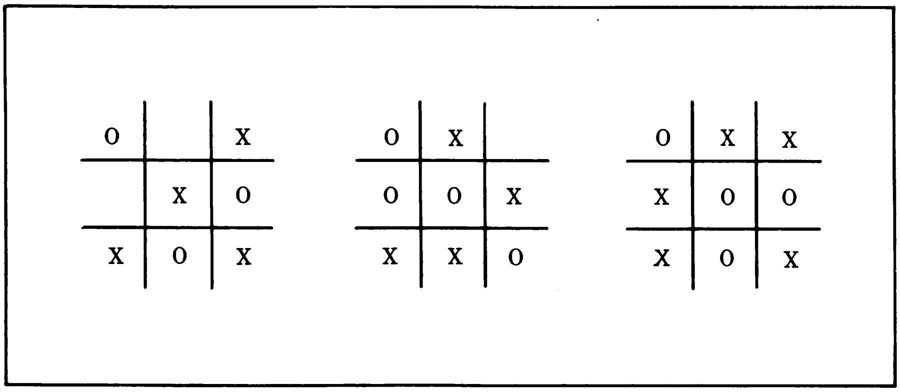


Figure 10-1. A win for player X, a win for player O, and a tie

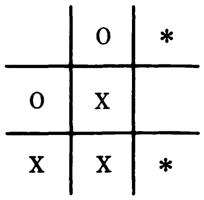


Figure 10-2. Player O is trapped; player X has two winning moves, indicated by asterisks

turns ahead to spot a potential trap. Furthermore, player O's very first mark can set him up for a possible loss. Figure 10-4 shows the seven configurations that player O must avoid on his first turn.

Before reading further, you may find it helpful to confirm for yourself that X can indeed force a win in each of the seven situations depicted in Figure 10-4.

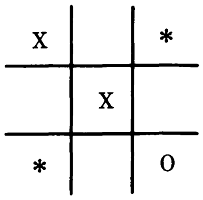


Figure 10-3. Player O can foil a trap by taking either safe cell indicated by an asterisk

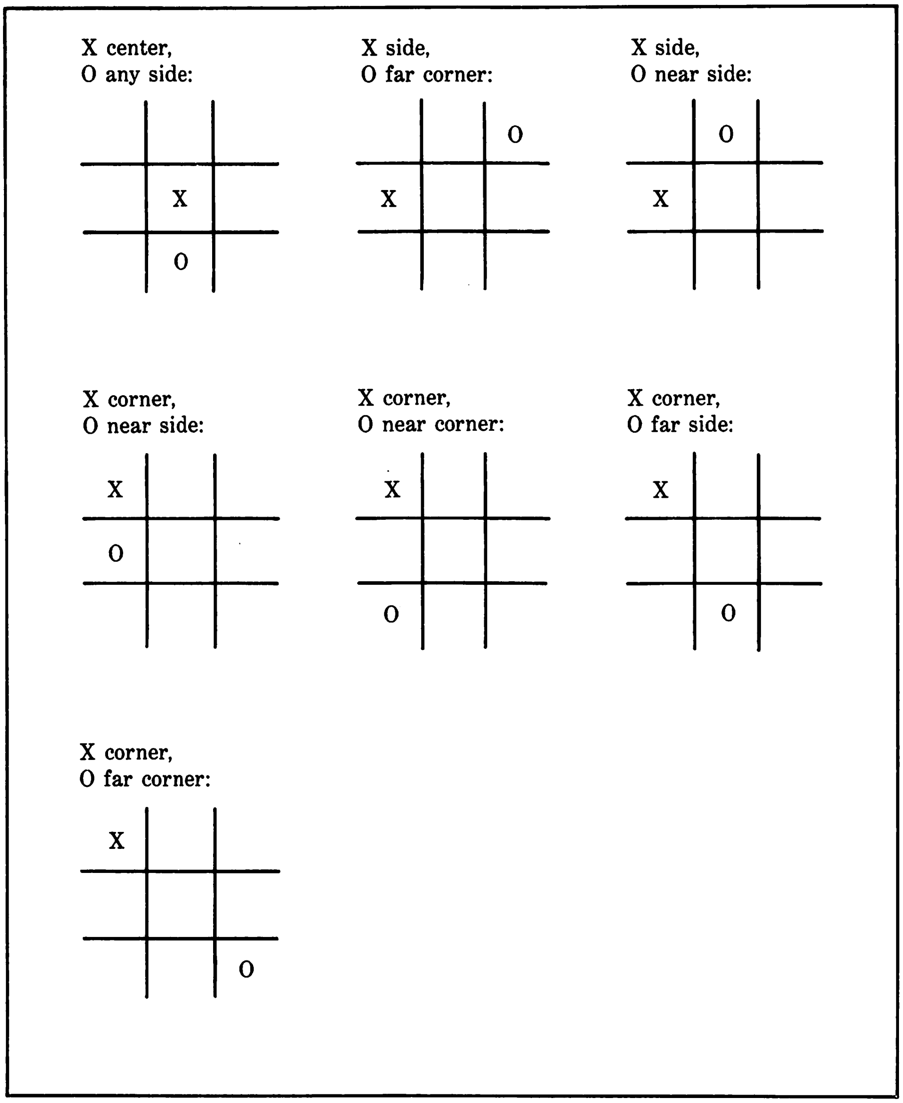


Figure 10-4. The seven losing positions for player O

—How the Program Plays

In the following discussion, we'll explain how the computer handles both roles—player X and player O. Occasionally, it may sound as if the computer is playing against itself, but keep in mind that in an actual game, you play one role and the computer plays the other.

Both players' first marks are treated as special cases. The program plays these turns "by the book" without looking ahead or using heuristic methods.

Before making subsequent marks for either player, the program applies up to five tests. The first two correspond to steps 1 and 2, outlined previously:

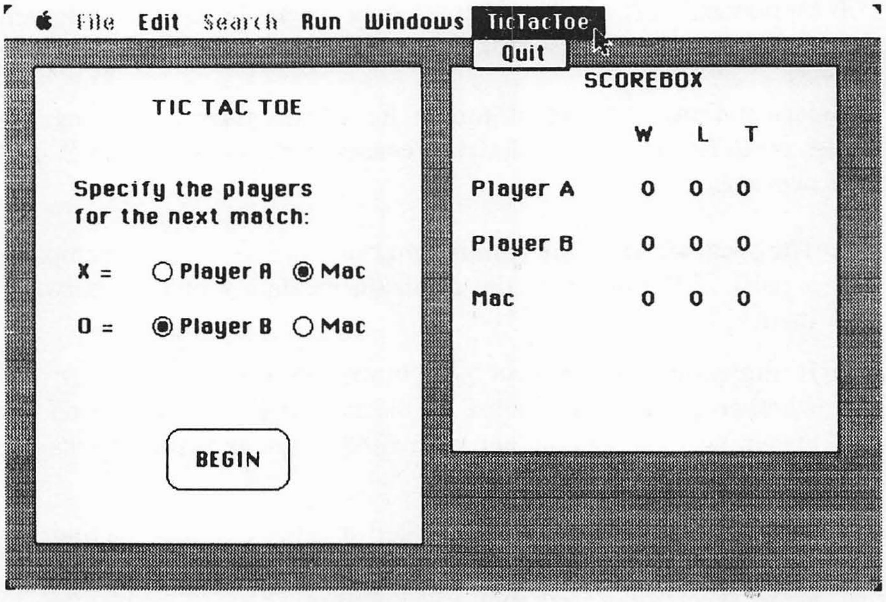
1. The program looks for winning marks—marks that will complete a path. If it finds any, the program randomly chooses between them.
2. If the program can find no winning marks, it checks to see whether the opponent must be blocked to prevent a win on his upcoming turn (looking one turn ahead). The program blocks the first such path it finds.
3. If the program still hasn't marked a cell, it begins looking for cells that will trap the opponent on his upcoming turn. The program chooses the first such cell it finds.
4. If none of these checks has resulted in a cell selection, the program looks for cells that will prevent the opponent from setting a trap on his next turn. This involves looking ahead two turns.
5. The program applies a heuristic method to choose among the cells that pass test 4. It takes that cell which results in the fewest paths that don't include any of its own marks. This makes sense: the fewer paths there are without a player's mark, the fewer chances the opponent has to win the game. However, the principle does not always produce the most aggressive strategy—hence the program's occasional willingness to settle for a draw when a win is possible.

—The Program

The first block adds an item to the menu bar:

```
MENU 6,0,1,"TicTacToe"
MENU 6,1,1," Quit "
ON MENU GOSUB menu.rq
MENU ON
```

The menu is shown in Figure 10-5.


File Edit Search Run Windows TicTacToe
Quit
TIC TAC TOE
Specify the players for the next match:
X = ○ Player A ◉ Mac
O = ◉ Player B ○ Mac
BEGIN
SCOREBOX

	W	L	T
Player A	0	0	0
Player B	0	0	0
Mac	0	0	0

Figure 10-5. The initial screen appearance, showing the scorebox and the player specification box

Array Definitions

The next block creates the arrays described below and reads in the data that is stored in the program:

```
RANDOMIZE TIMER
DIM tc%(3,3), ok%(9,3), t%(3,3), p%(2), dr%(4), dc%(4)
DIM path.r%(8), path.c%(8), path.dir%(8), nw%(2)
DIM p$(3), oc$(2), wins%(3), losses%(3), ties%(3)
FOR r%=1 TO 3
FOR c%=1 TO 3
READ t%(r%,c%)
NEXT c%,r%
DATA 2, 3, 2, 3, 1, 3, 2, 3, 2
FOR dn%=1 TO 4
READ dr%(dn%),dc%(dn%)
NEXT dn%
DATA 0,1,1,1,1,0,1,-1
FOR pa%=1 TO 8
READ path.r%(pa%),path.c%(pa%),path.dir%(pa%)
NEXT pa%
```

```
DATA 1, 1, 1, 1, 1, 2, 1, 1, 3, 1, 2, 3
DATA 1, 3, 4, 1, 3, 3, 2, 1, 1, 3, 1, 1
LET oc$(1)="X"
LET oc$(2)="O"
LET p$(1)="Player A"
LET p$(2)="Player B"
LET p$(3)="Mac"
LET p%(1)=3
LET p%(2)=2
```

Array tc%(,) stores an image of the tic-tac-toe board. For row r%, column c%, tc%(r%,c%)=0 indicates an empty cell; tc%(r%,c%)=1 indicates an X; and tc%(r%,c%)=2 indicates an O. Ok%(,) keeps track of all the prospective cells that prevent the opponent from setting a trap on his next turn. T%(,) stores the type of each grid position—center, corner, or side. This information comes in handy when the computer is analyzing the board position before making its first mark as player O.

P%( ) keeps track of who the players are: p%(pn%)=1 indicates a human, and p%(pn%)=2 indicates the computer. Depending on how p%(1) and p%(2) are set, the computer may play against itself or against a person, or two people can play against each other. Initially, the program sets the first player to "Mac" and the second player to Player B.

Dr%( ) and dc%( ) store the row and column increments producing the four possible path directions. A similar array is used in the Hidden Words and Crossword Puzzle Patterns programs. Path.r%( ), path.c%( ), and path.dir%( ) store information that defines the eight paths on a tic-tac-toe grid (see Figure 10-6).

Nw%( ) identifies paths that contain a specified number of one player's marks. P$( ) stores the name assigned to each player. "Player A" is used for the first human player, "Player B" for the second human player, and "Mac" for the computer. Oc$( ) stores the X and O characters. Wins%( ), losses%( ), and ties%( ) keep track of the performance of each of the three possible players.

Window Definitions

The next block of lines defines and creates the two windows (the game and the scorebox) used by the program:

```
DIM wnd.w%(2), wnd.l%(2), wnd.x%(2), wnd.y%(2), wnd.x1%(2), wnd.y1%(2)
FOR n%=1 TO 2
READ inches.wide,inches.long,ulcx,ulcy
```

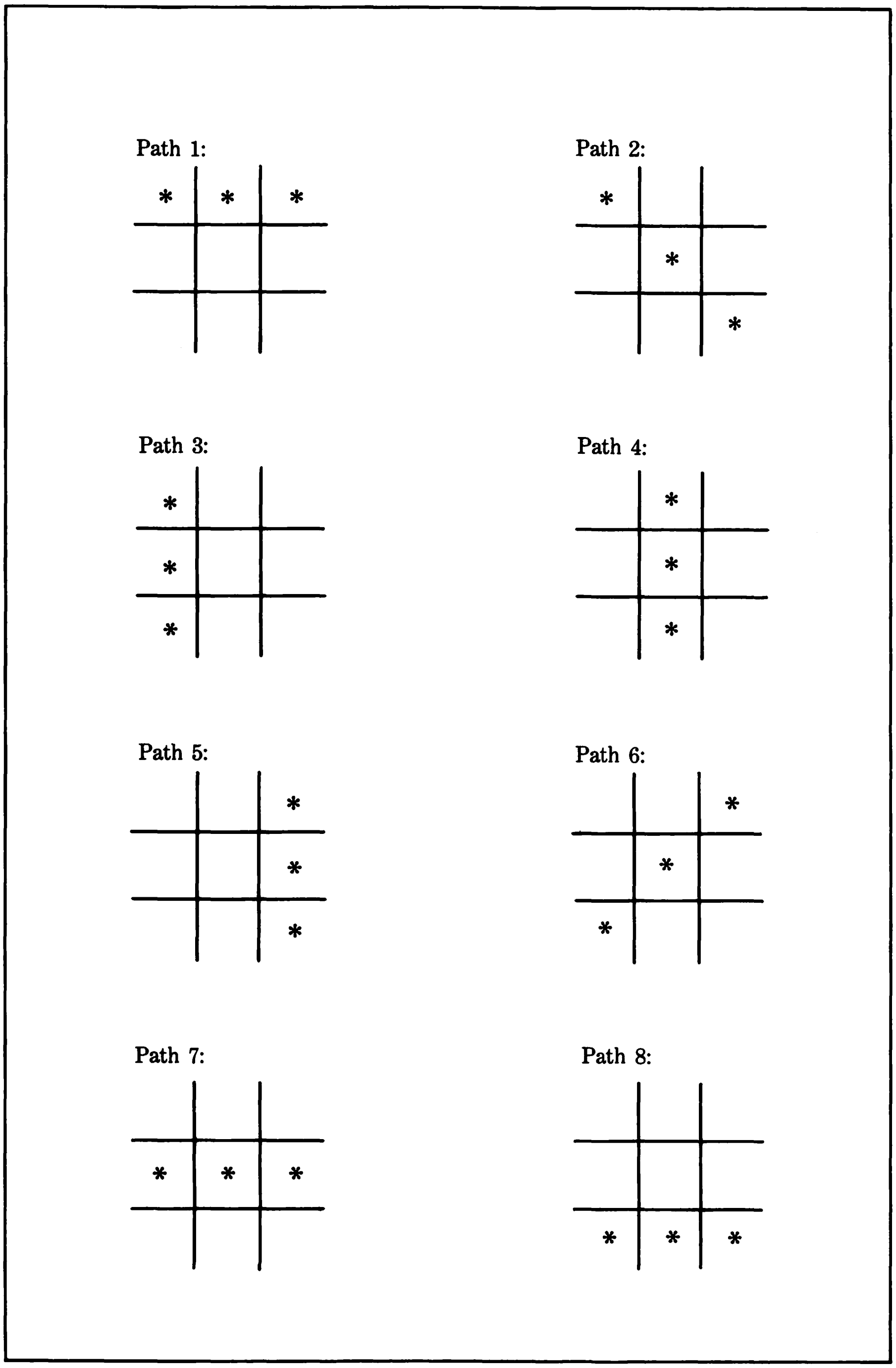
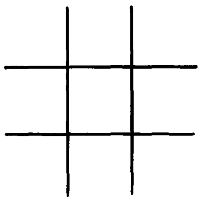


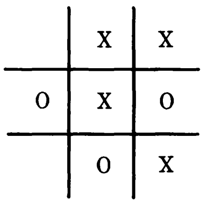
Figure 10-6. Illustration of the eight paths and the use of arrays t%(,) and tc%(,)

The array T(,) stores the type number of each cell: center=1, corner=2, side=3



$$T(\,,\,) = \begin{matrix} 2 & 3 & 2 \\ 3 & 1 & 3 \\ 2 & 3 & 2 \end{matrix}$$

In the following situation, for player X, NW(1)=1 since path 1 contains 2 X's and no O's



$$TC(\,,\,) = \begin{matrix} 0 & 1 & 1 \\ 2 & 1 & 2 \\ 0 & 2 & 1 \end{matrix}$$

Figure 10-6. Illustration of the eight paths and the use of arrays t%(,) and tc%(,) (*continued*)

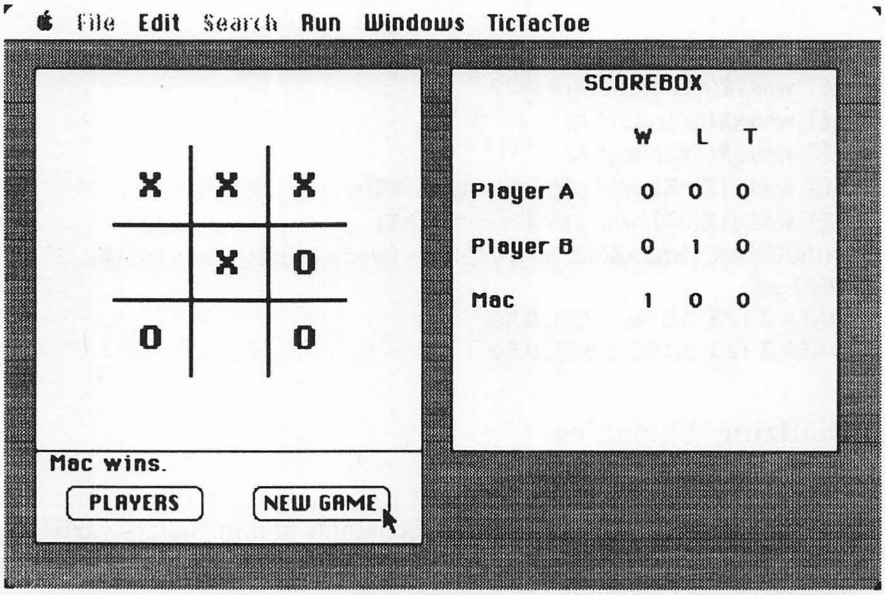
```
LET wnd.w%(n%)=inches.wide*72
LET wnd.l%(n%)=inches.long*72
LET wnd.x%(n%)=ulcx*72
LET wnd.y%(n%)=ulcy*72
LET wnd.x1%(n%)=wnd.x%(n%)+wnd.w%(n%)
LET wnd.y1%(n%)=wnd.y%(n%)+wnd.l%(n%)
WINDOW n%,,(wnd.x%(n%),wnd.y%(n%))- (wnd.x1%(n%), wnd.y1%(n%)), 3
NEXT n%
DATA 3.125, 3.875, 0.250, 0.50
DATA 3.125, 3.125, 3.625, 0.50
```

Initializing Variables And Setting Constants

Now the program establishes certain constants and initializes variables:

```
LET zone%=wnd.w%(1)\5
LET bg.w%=72
LET bg.l%=36
LET bg.x%=(wnd.w%(1)-bg.w%)\2
LET bg.y%=(wnd.l%(1)-bg.l%)*7\8
LET rm.w%=80
LET rm.l%=20
LET rm.x%=(wnd.w%(1)-rm.w%)*1\8
LET rm.y%=(wnd.l%(1)-rm.l%)*19\20
LET pl.w%=80
LET pl.l%=20
LET pl.x%=(wnd.w%(1)-pl.w%)*7/8
LET pl.y%=(wnd.l%(1)-pl.l%)*19\20
FOR j%=1 TO 3
LET wins%(j%)=0
LET ties%(j%)=0
LET losses%(j%)=0
NEXT j%
```

Zone% defines the height and width of a tic-tac-toe grid cell. The bg. variables locate the BEGIN button seen in Figure 10-5. The rm. and pl. variables locate the PLAYERS and NEW GAME buttons seen in Figure 10-7.



Mac wins.

SCOREBOX

	W	L	T
Player A	0	0	0
Player B	0	1	0
Mac	1	0	0

Figure 10-7. Screen appearance after a win by Mac

The Scorebox

The next lines add labels and initial values to the scorebox in window 2:

```
WINDOW 2
CALL TEXTFACE(1)
CLS
PRINT PTAB((wnd.w%(2)- WIDTH("SCOREBOX"))\2); "SCOREBOX"
PRINT
REM             123456789
PRINT TAB(12); " W   L   T"
FOR plr%=1 TO 3
LOCATE plr%*2+3,2
PRINT p$(plr%)
NEXT plr%
GOSUB update.scorebox
```

When entering the lines, use the REM line containing "123456789" to help you space the headings "W L T" correctly.

Setting Up a Match

Next the program lets you determine who the players are (as shown in Figure 10-5):

```
rematch:
WINDOW 1   : REM new match
CALL TEXTSIZE(12)
CALL TEXTFACE(1)
CLS
LOCATE 2,1
PRINT PTAB((wnd.w%(1)- WIDTH("TIC TAC TOE"))\2); "TIC TAC TOE"
LOCATE 5,1
PRINT "   Specify the players"
PRINT "   for the next match:"
LOCATE 8,1
PRINT PTAB(24); "X ="
LOCATE 10,1
PRINT PTAB(24); "O ="
IF p%(1)=2 THEN SWAP p%(1),p%(2)
BUTTON 1,1-(p%(1)=1),p$(1),(66,112)-(138,127),3
BUTTON 2,1-(p%(1)=3),p$(3),(148,112)-(196,127),3
BUTTON 3,1-(p%(2)=2),p$(2),(66,144)-(138,159),3
BUTTON 4,1-(p%(2)=3),p$(3),(148,144)-(196,159),3
BUTTON 5,1,"BEGIN",(bg.x%,bg.y%)- (bg.x%+bg.w%, bg.y%+bg.l%)
```

By selecting the appropriate buttons, you may specify three different matches: Player A against Mac, Mac against Player B, or Mac against Mac.

The next lines wait for you to change the player buttons and press BEGIN:

```
match.loop:
GOSUB wait.entry
IF event%=6 THEN match.set
LET btn%=DIALOG(1)
ON btn% GOTO p1,p1,p2,p2,match.set
p1:
IF btn%=1 THEN LET p%(1)=1 ELSE LET p%(1)=3
BUTTON 1,1-(p%(1)=1)
BUTTON 2,1-(p%(1)=3)
GOTO match.loop
p2:
IF btn%=3 THEN LET p%(2)=2 ELSE LET p%(2)=3
BUTTON 3,1-(p%(2)=2)
BUTTON 4,1-(p%(2)=3)
GOTO match.loop
match.set:
FOR j%=1 TO 5
BUTTON CLOSE j%
NEXT j%
```

Starting a Game

The following lines draw the playing grid and erase the computer's internal game array tc%(,):

```
next.game:
LET mn%=1
LET pn%=1
CLS
CALL TEXTMODE(1)
CALL PENSIZE(2,2)
FOR lin%=2 TO 3
CALL MOVETO(zone%*lin%,zone%)
CALL LINE(0,3*zone%)
CALL MOVETO(zone%,zone%*lin%)
CALL LINE(3*zone%,0)
NEXT lin%
CALL PENSIZE(1,1)
CALL MOVETO(0,zone%*5)
CALL LINE(5*zone%,0)
FOR r%=1 TO 3    : REM erase board
FOR c%=1 TO 3
LET tc%(r%,c%)=0
NEXT c%
NEXT r%
```

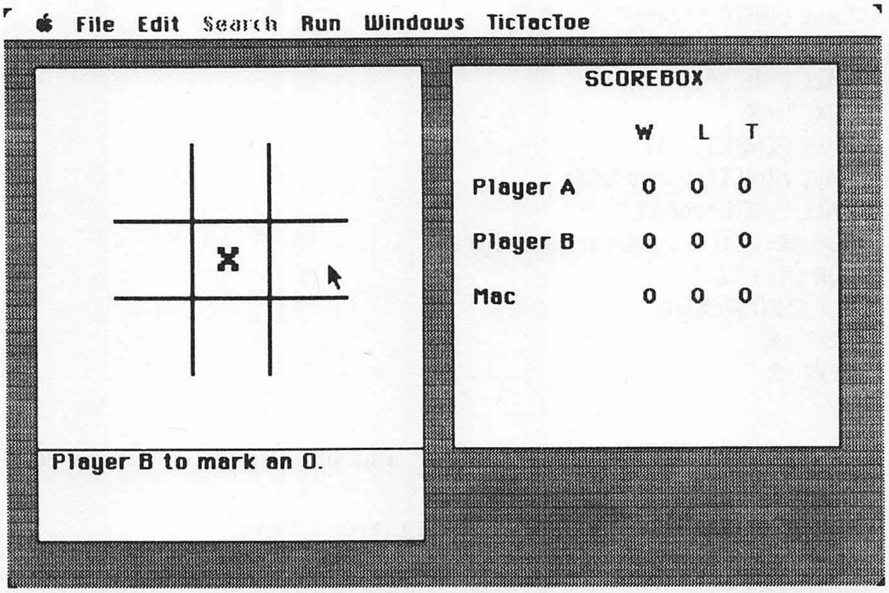
Mn% is the current move number and pn% indicates the current mark (X or O).

Now the program gets the current player's move:

```
next.player:
CALL TEXTSIZE(12)
LINE (0,zone%*5+1)- (5*zone%-1, 6.2*zone%-1) ,0,bf
CALL MOVETO(8,5*zone%+12)
PRINT p$(p%(pn%)); " to mark an ";oc$(pn%);"."
ON p%(pn%) GOSUB players.mark, players.mark, macs.mark
CALL TEXTSIZE(18)    :REM Mark oc$(pn%) in rm%,cm%
LET tx%=cm%*zone%+(zone%-12)\2
LET ty%=rm%*zone%+30
CALL MOVETO(tx%,ty%)
PRINT oc$(pn%);
LET sl%=3
LET st%=pn%
GOSUB analyze.grid
IF n%>0 THEN win.game
IF mn%=9 THEN tie.game
LET mn%=mn%+1
LET pn%=pn% MOD 2 +1
GOTO next.player
```

The program prints a prompt message in the bottom of window 1, as shown in Figure 10-8. Then control is passed to the appropriate subroutine, depending on whether the next player is a person or the Mac (ON p%(pn%) GOSUB players.mark, players.mark, macs.mark).

Upon returning from the players.mark or macs.mark subroutine,



SCOREBOX

	W	L	T
Player A	0	0	0
Player B	0	0	0
Mac	0	0	0

Player B to mark an O.

Figure 10-8. Screen appearance with a game in progress

the program marks the move on the playing grid (CALL MOVETO... and PRINT oc$(pn%)).

Next the program determines whether the latest move has ended the game in a win or a tie. The analyze.grid subroutine searches all eight paths to see if the current player p%(pn%) has won. N%>0 indicates a win.

If n%=0, the computer checks the turn number mn% to see whether the game has ended in a tie. There are only nine cells in the grid; hence if mn%=9 and there is still no winner, the board is full and the computer deduces that the game is a tie.

If the latest move produced neither a win nor a tie, the program gives the next player a turn.

Wins and Ties

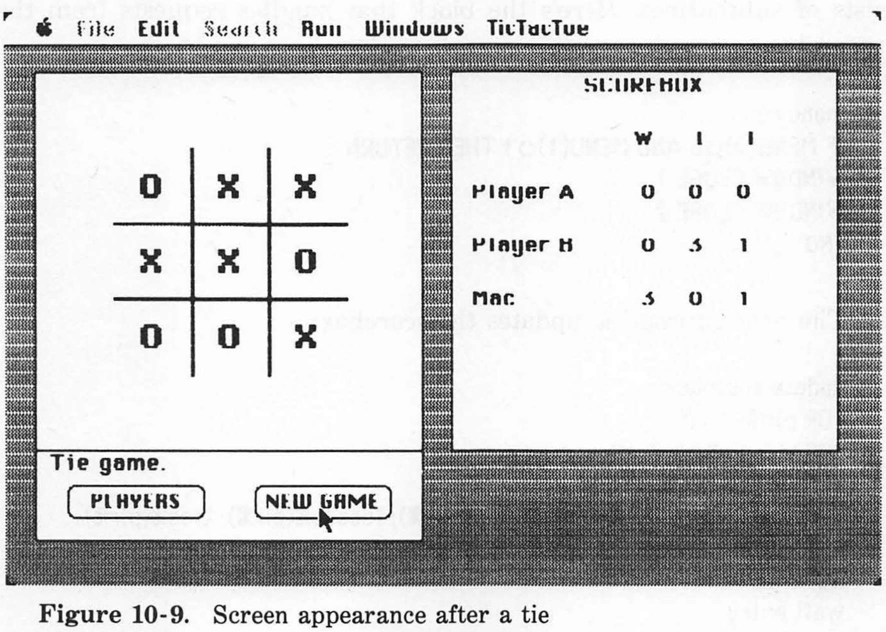
The next block of lines handles wins and ties:

```
win.game:
FOR fq%=1 TO 4
SOUND fq%*110,1
NEXT fq%
```

```
CALL TEXTSIZE(12)
LINE (0,zone%*5+1)- (5*zone%-1, 6.2*zone%-1) ,0,bf
CALL MOVETO(8,5*zone%+12)
PRINT p$(p%(pn%)); " wins."
LET wins%(p%(pn%))=wins%(p%(pn%))+1
LET losses%(p%(3-pn%))=losses%(p%(3-pn%))+1
GOTO prepare.next
tie.game:
SOUND 440,2
SOUND 110,2
CALL TEXTSIZE(12)
LINE (0,zone%*5+1)- (5*zone%-1, 6.2*zone%-1) ,0,bf
CALL MOVETO(8,5*zone%+12)
PRINT "Tie game."
LET ties%(p%(pn%))=ties%(p%(pn%))+1
LET ties%(p%(3-pn%))=ties%(p%(3-pn%))+1
```

In the case of a win, the computer sounds an ascending sequence of notes and announces the winner, as shown in Figure 10-7.

In the case of a tie, the computer sounds a descending couplet and announces the tie, as shown in Figure 10-9.



Tie game.

```
SCOREBOX
              W   L   T
Player A      0   0   0
Player B      0   3   1
Mac           3   0   1
```

Figure 10-9. Screen appearance after a tie

After the win or tie is announced, the following lines take over:

```
prepare.next:
WINDOW 2
GOSUB update.scorebox
WINDOW 1
BUTTON 1,1,"PLAYERS",(rm.x%,rm.y%)-(rm.x%+rm.w%,rm.y%+rm.l%)
BUTTON 2,1,"NEW GAME",(pl.x%,pl.y%)- (pl.x%+pl.w%, pl.y%+pl.l%)
GOSUB wait.entry
IF event%=6 THEN same.match
IF DIALOG(1)=1 THEN rematch
same.match:
SWAP p%(1),p%(2)
GOTO next.game
```

Before starting a new game, the computer updates the scorebox and provides the two continuation buttons shown in Figures 10-7 and 10-9. Pressing the PLAYERS button lets you change the players in the match. Pressing the NEW GAME button starts another game.

Subroutines

That completes the main program. The remainder of the program consists of subroutines. Here's the block that handles requests from the menu bar:

```
menu.rq:
IF MENU(0)<>6 AND MENU(1)<>1 THEN RETURN
WINDOW CLOSE 1
WINDOW CLOSE 2
END
```

The next subroutine updates the scorebox:

```
update.scorebox:
FOR plr%=1 TO 3
LOCATE plr%*2+3,12
REM           123456789
PRINT USING "## ## ##"; wins%(plr%); losses%(plr%); ties%(plr%)
NEXT plr%
RETURN
wait.entry:
```

```
LET event%=DIALOG(0)
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%<>1 AND event%<>6 THEN wait.entry
RETURN
```

When entering these lines, use the REM 123456789 line as a guide in spacing the quoted pound signs on the following line.

Analyzing the Tic-Tac-Toe Grid

The following subroutine analyzes the contents of all eight paths:

```
analyze.grid:
LET n%=0
LET m%=0
FOR p%=1 TO 8
LET ru%=path.r%(p%)
LET cu%=path.c%(p%)
LET dn%=path.dir%(p%)
LET nf%=0
LET mf%=0
FOR ce%=1 TO 3
IF tc%(ru%,cu%)=0 THEN ag.next.cell
IF tc%(ru%,cu%)=st% THEN ag.players
LET mf%=mf%+1
GOTO ag.next.cell
ag.players:
LET nf%=nf%+1
ag.next.cell:
LET ru%=ru%+dr%(dn%)
LET cu%=cu%+dc%(dn%)
NEXT ce%
IF nf%<>sl% OR mf%>0 THEN ag.others
LET n%=n%+1
LET nw%(n%)=p%
ag.others:
IF mf%>0 THEN next.path
LET m%=m%+1
next.path:
NEXT p%
RETURN
```

The variable n% counts the number of unblocked paths containing at least the number of marks indicated by sl% for the player specified by st%. Variable m% counts the number of paths containing none of the other players' marks.

Upon return from this subroutine, the array nw%( ) lists the path numbers of all unblocked paths containing at least the number of marks sl% for player st%.

Getting a Random Mark

The next short block of lines locates a randomly chosen empty cell from the grid:

```
random.mark:
LET rt%=INT(RND*3)+1
LET ct%=INT(RND*3)+1
IF tc%(rt%,ct%)<>0 THEN random.mark
RETURN
```

Rt% and ct% are the cell's row and column numbers. Tc%(rt%,ct%)<>0 indicates the cell is taken; in that case, the program makes another random selection.

Getting the Player's Selection

The following subroutine lets a human player (Player A or Player B) indicate where his next mark should go:

```
players.mark:
WHILE MOUSE(0)<>1
WEND
LET mx%=MOUSE(1)
LET my%=MOUSE(2)
LET cm%=(mx%-zone%)\zone%+1
LET rm%=(my%-zone%)\zone%+1
IF cm%<1 OR cm%>3 OR rm%<1 OR rm%>3 THEN players.mark
IF tc%(rm%,cm%)<>0 THEN players.mark
LET tc%(rm%,cm%)=pn%
RETURN
```

To specify a location, the player points to the desired location on the tic-tac-toe board and clicks the mouse button.

Variables rm% and cm% specify the desired row and column numbers. If they are out of the range 1 to 3 or if tc%(rm%,cm%) is already taken, the program ignores the selection and waits for another one (IF cm%<1 ... and IF tc%(rm%,cm%)< >0...).

Test Point

Before entering the logic that lets your Macintosh play, you can now test the program with two human players, Player A and Player B.

First a printout of the program. Type COMMAND-. and LLIST, and press RETURN.

Carefully check the listing against the listings in this chapter; then close the listing window and run the program (COMMAND-R).

You should see the screen shown in Figure 10-5. Specify a match between player A and player B. (The Mac can't play yet.)

Now play a game. The computer should alternately ask for moves from Player A and Player B until one player wins or there is a tie (nine moves without a win).

After getting the test to work, stop the program (select Quit from the Tic-Tac-Toe menu). Reopen the listing window (COMMAND-L), and position the insertion point at the end of the listing.

The Mac's Move

Now we present the lines that let the computer play tic-tac-toe.

The program uses prepared "book" moves only for the first X and the first O. The first X is a random selection, and the first O is determined by the location of the first X. For subsequent moves, the computer uses its look-ahead logic.

Playing by the Book

The following lines determine the computer's first X or O:

```
macs.mark:
IF mn%>2 THEN look.ahead
IF mn%=2 THEN second.mark
GOSUB random.mark
LET rm%=rt%
LET cm%=ct%
GOTO accept.move
second.mark:
LET t%=t%(rm%,cm%)
sm.loop:
```

```
GOSUB random.mark
ON t% GOTO corner,center,side
corner:
IF t%(rt%,ct%)=3 THEN sm.loop
GOTO sm.done
center:
LET rt%=2
LET ct%=2
GOTO sm.done
side:
ON t%(rt%,ct%) GOTO sm.done,center.1,side.1
center.1:
IF ABS(rt%-rm%)=2 OR ABS(ct%-cm%)=2 THEN sm.loop
GOTO sm.done
side.1:
IF ABS(rt%-rm%)=1 OR ABS(ct%-cm%)=1 THEN sm.loop
sm.done:
LET rm%=rt%
LET cm%=ct%
GOTO accept.move
```

If mn%>2, that is, if the move number is greater than 2, the program jumps to the standard program logic described later. If mn%=1 (first move), the program selects a move at random. If mn%=2 (second move), the program randomly selects a cell and then makes sure that the cell doesn't create one of the losing situations shown in Figure 10-4.

Once the program has located a safe cell, the program stores the row and column address of the cell in rm%,cm% and jumps to the end of the accept.move subroutine.

Looking Ahead

In the case of second and subsequent turns, the program no longer uses prepared moves to play. First it checks to see whether it can win with one mark:

```
look.ahead:
IF mn%<5 THEN check.opponent
LET st%=pn%
LET sl%=2
GOSUB analyze.grid
IF n%=0 THEN check.opponent
LET m%=INT(RND*n%)+1
LET p%=nw%(m%)
GOSUB find.opening
LET rm%=ro%
LET cm%=co%
GOTO accept.move
```

If the computer is making its second mark (mn%<5), there's no point in looking for a winning cell yet (it takes three marks to fill a path). In that case, the program goes immediately to check for a potential loss (check.opponent).

If the move number, mn%, is greater than 4, the program looks for a winning cell. The analyze.grid subroutine counts the number of unblocked paths containing at least two marks of the player specified by pn%. If n%=0, there are no potential wins, so the program skips to the check.opponent routine described later.

If n%>0, the array nw%( ) lists the paths that contain winning cells. The program randomly selects one of these paths, and the find.opening subroutine finds the row and column of the open cell in that path.

Now that the program has located a winning cell, the program stores its row and column address and jumps to the end of the accept.move subroutine.

Preventing Imminent Defeat

If the program cannot find a winning cell, it next checks to see whether it must prevent its opponent from winning on the next turn:

```
check.opponent:
LET st%=3-pn%
LET sl%=2
GOSUB analyze.grid
IF n%=0 THEN set.trap
LET p%=nw%(n%)
GOSUB find.opening
LET rm%=ro%
LET cm%=co%
GOTO accept.move
```

The program first sets st% equal to the number of the opposing player, and then uses analyze.grid to count the number of unblocked paths containing at least two of the opposing player's marks. If n%=0, there are none, so the program skips to the trap-setting routine described in the next section.

If n% does not equal 0, there is at least one way for the opposing player to win on his next move. The find.opening subroutine finds the opponent's winning cell, and the program stores its row and column address so the computer can claim it.

Setting a Trap

If the computer still hasn't made a selection for player number pn%, the computer looks for a move that will trap the opponent and guarantee a win on the computer's next turn.

```
set.trap:
IF mn%<5 THEN prevent.traps
LET st%=pn%
GOSUB find.trap
IF n%<>2 THEN prevent.traps
LET rm%=rv%
LET cm%=cv%
GOTO accept.move
```

If the computer is making its second mark (mn%<5), there is no way it can set a trap yet, so the program skips to the prevent.traps routine. Otherwise, the program looks for a move that will create a trap. The find.trap subroutine tests every empty cell to see which, if any, produces a trap. If n%=2, the program has found such a cell, so the program claims that cell.

Foiling a Trap

If no opportunities to set a trap are found, the program checks every empty cell to see which one will prevent the opponent from setting a trap on his next turn. This is the farthest look ahead the program takes:

```
prevent.traps:
LET f%=0
FOR rm%=1 TO 3
FOR cm%=1 TO 3
IF tc%(rm%,cm%)<>0 THEN pt.next
```

```
LET tc%(rm%,cm%)=pn%
LET st%=pn%
LET sl%=2
GOSUB analyze.grid
IF n%=0 THEN no.traps
IF mn%=3 THEN found.one
LET p%=nw%(1)
GOSUB find.opening
LET st%=3-pn%
LET tc%(ro%,co%)=st%
LET sl%=2
GOSUB analyze.grid
LET tc%(ro%,co%)=0
GOTO record.trap
no.traps:
IF mn%=3 THEN fix.grid
LET st%=3-pn%
GOSUB find.trap
record.trap:
IF n%=2 THEN fix.grid
found.one:
LET f%=f%+1
LET ok%(f%,1)=rm%
LET ok%(f%,2)=cm%
fix.grid:
LET tc%(rm%,cm%)=0
pt.next:
NEXT cm%,rm%
```

The variable f% counts the number of safe cells (those that will prevent the opponent from setting a trap). The computer tries marking each empty cell in the grid one at a time. For each cell marked, the program checks whether its opponent can set a trap.

For each safe cell f that is found, ok(f,1) stores its row and ok(f,2) stores its column location.

Heuristic Method

After the program has located all the safe cells, it applies a heuristic method to choose among them:

```
REM: Heuristic
LET sl%=2
LET st%=3-pn
FOR cn%=1 TO f%
LET tc%(ok%(cn%,1),ok%(cn%,2))=pn%
GOSUB analyze.grid
LET tc%(ok%(cn%,1),ok%(cn%,2))=0
LET ok%(cn%,3)=m%
NEXT cn%
IF f%<>1 THEN pick.best
LET cn%=1
GOTO indicate.move
pick.best:
LET sm%=1
FOR it%=2 TO f%
IF ok%(sm%,3)<ok%(it%,3) THEN no.swap
LET sm%=it%
no.swap:
NEXT it%
pick.cell:
LET cn%=INT(RND*f%)+1
IF ok%(cn%,3)=ok%(sm%,3) THEN indicate.move
GOTO pick.cell
indicate.move:
LET rm%=ok%(cn%,1)
LET cm%=ok%(cn%,2)
accept.move:
LET tc%(rm%,cm%)=pn%
RETURN
```

The program marks each safe cell (LET tc% (ok% (cn%,1), ok% (cn%,2)) = pn%) and then counts the number unblocked paths M that remain. For each safe cell f%, ok%(f%,3) stores the number of unblocked paths that remain when that cell is marked.

Beginning at pick.best, the program compares the results of the trial marks to see which marks result in the lowest number (SM) of unblocked paths. Beginning at pick.cell, the program randomly picks safe cells until it finds one that leaves the number of unblocked paths indicated by SM.

Now that the program has located a suitable cell, the program stores the row-and-column address of that cell so the computer can claim it.

Finally, the computer accepts the chosen board location and ends its turn (accept.move).

Auxiliary Subroutines

The following subroutine locates the first opening in path p%:

```
find.opening:
LET rt%=path.r%(p%)
LET ct%=path.c%(p%)
LET dn%=path.dir%(p%)
FOR ce%=1 TO 3
IF tc%(rt%,ct%)<>0 THEN fo.not.open
LET ro%=rt%
LET co%=ct%
LET ce%=3
fo.not.open:
LET rt%=rt%+dr%(dn%)
LET ct%=ct%+dc%(dn%)
NEXT ce%
RETURN
```

Upon return from the subroutine, ro% and co% specify the row and column of the open cell.

Finally, this subroutine looks for an opportunity to set a trap (mark a cell that creates two winning threats for a player's next turn):

```
find.trap:
FOR rb%=1 TO 3
FOR cb%=1 TO 3
IF tc%(rb%,cb%)<>0 THEN ft.not.open
LET sl%=2
LET tc%(rb%,cb%)=st%
GOSUB analyze.grid
LET tc%(rb%,cb%)=0
IF n%<2 THEN ft.not.open
LET rv%=rb%
LET cv%=cb%
LET cb%=3
LET rb%=3
ft.not.open:
NEXT cb%,rb%
RETURN
```

On entry to the subroutine, st% is the number of the player looking to set the trap. On return from the subroutine, n%=2 indicates that a trap was found, and rv%, cv% identify the row and column of the cell that sets the trap.

—Using the Program

Reprint the second half of the program (COMMAND-. and LLIST macs.mark-) and check it carefully against the listings in this chapter. Close the listing window and run the program (COMMAND-R). The results should be similar to those shown in Figures 10-5 and 10-7 through 10-9.

Chapter 11

Quizmaster

This chapter presents Quizmaster, a program that will help you learn information on any subject you choose. The program lets you enter information into a database (a collection of items that have something in common); it then uses that database to test your knowledge of the facts. You may choose between two types of tests: multiple choice or fill in the blank.

The program lets you store the database in a disk file, so you can have several different databases available to help you study a variety of subjects.

Preparing a Quizmaster Database

A Quizmaster database starts out with pairs of related words, phrases, names, or other items. For instance, a states-and-capitals database would start out this way:

Item A	Item B
Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix

To make a working Quizmaster database, you also need a pair of questions that relate each item in column A with its match in column B, and vice versa. The answers to question A come from column A, and the answers to question B come from column B.

For the states-and-capitals database, you could use these questions: Question A: In what state is the city of...? Question B: What is the capital city of...?

Finally, a title for the database is needed; in the present example, States and Capitals is a good title.

The sample screens given in this chapter (Figures 11-1 through 11-11) show the Quizmaster being used to create a French and English vocabulary database.

—The Program

If you scan through the figures in this chapter, you'll notice two windows and numerous buttons and edit fields. Each button or edit field requires at least four descriptive numbers that determine its shape, labeling, and screen location.

To keep all these numbers organized, the program uses three groups of DATA statements.

Window, Button, and Field Descriptors

First, here are the window descriptors:

```
REM Window descriptors
DATA 2
REM  wide  long   left  top
DATA 6.000, 3.000, 0.500, 0.375
DATA 5.000, 1.125, 1.000, 3.500
```

As indicated in the REM statement, each DATA line consists of specifications for window width, length, left side, and top side (all measured in inches).

Now type in the button descriptors:

```
REM Button descriptors
DATA 16
REM  label              wide  long  hzone vzone  type
```

```
DATA KEY IN NEW,       1.500, 0.333, 0.313, 0.313, 1
DATA EDIT/REVIEW,      1.500, 0.333, 0.750, 0.313, 1
DATA LOAD FILE,        1.500, 0.333, 0.313, 0.625, 1
DATA SAVE FILE,        1.500, 0.333, 0.750, 0.625, 1
DATA BACK,             0.750, 0.333, 0.250, 0.958, 1
DATA FORWARD,          1.000, 0.333, 0.500, 0.958, 1
DATA OK,               0.500, 0.333, 0.750, 0.958, 1
DATA Fill in the blank,1.750, 0.208, 0.000, 0.313, 3
DATA Multiple choice,  1.750, 0.208, 0.000, 0.438, 3
DATA Question A,       5.000, 0.208, 0.000, 0.625, 3
DATA Question B,       5.000, 0.208, 0.000, 0.750, 3
DATA BEGIN,            1.000, 0.333, 0.500, 0.958, 1
DATA first choice,     2.500, 0.208, 0.120, 0.438, 3
DATA second choice,    2.500, 0.208, 0.880, 0.438, 3
DATA third choice,     2.500, 0.208, 0.120, 0.563, 3
DATA fourth choice,    2.500, 0.208, 0.880, 0.563, 3
```

The hzone and vzone variables determine the relative horizontal and vertical position of a button. For example, hzone=0.313 is about 3/16 of the way across the window; vzone=0.500 is halfway down the window.

Type refers to the button type: 1 is a standard pushbutton; 3 is a radio-style button.

Last come the field descriptors:

```
REM Field descriptors
DATA 7
REM  wide  long  hzone vzone
DATA 2.500, 0.208, 0.500, 0.167
DATA 5.000, 0.208, 0.500, 0.375
DATA 5.000, 0.208, 0.500, 0.583
DATA 2.500, 0.208, 0.120, 0.792
DATA 2.500, 0.208, 0.880, 0.792
DATA 0.500, 0.208, 0.934, 0.333
DATA 2.500, 0.208, 0.500, 0.500
```

Reading the Data

Three blocks of lines read the window, button, and field descriptors. The following lines read the window data:

```
READ nw%
DIM ww%(nw%), wl%(nw%), wx%(nw%), wy%(nw%), wx1%(nw%), wy1%(nw%)
FOR n%=1 TO nw%
```

```
READ inches.wide,inches.long,ulcx,ulcy
LET ww%(n%)=inches.wide*72
LET wl%(n%)=inches.long*72
LET wx%(n%)=ulcx*72
LET wy%(n%)=ulcy*72
LET wx1%(n%)=wx%(n%)+ww%(n%)
LET wy1%(n%)=wy%(n%)+wl%(n%)
NEXT n%
```

For windows 1 and 2, ww%( ) and wl%( ) store the width and length expressed in screen points or pixels. Wx%( ) and wy%( ) store the coordinates of the upper-left corner; wx1%( ) and wy1%( )store the coordinates of the lower-right corner.

The next lines read the button data:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
READ bl$(n%),inches.wide,inches.long,h.zone,v.zone,bt%(n%)
LET bx%(n%)=(ww%(1)-inches.wide*72)*h.zone
LET by%(n%)=(wl%(1)-inches.long*72)*v.zone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```

Bl$( ) stores the button labels. Bx%( ), by%( ) and bx1( ), by1%( ) define the upper-left and lower-right corners of each button. Bt%( ) stores the button type.

The last block of lines in this section of the program reads the edit field descriptors:

```
READ nf%
DIM fx%(nf%),fy%(nf%),fx1%(nf%),fy1%(nf%)
FOR n%=1 TO nf%
READ inches.wide,inches.long,h.zone,v.zone
LET fx%(n%)=(ww%(1)-inches.wide*72)*h.zone
LET fy%(n%)=(wl%(1)-inches.long*72)*v.zone
LET fx1%(n%)=fx%(n%)+inches.wide*72
LET fy1%(n%)=fy%(n%)+inches.long*72
NEXT n%
```

The four arrays (fx%( ) and so forth) serve to locate the upper-left and the lower-right corners of each edit field.

Constants and Parameters

The next block of lines initializes a group of variables:

```
RANDOMIZE TIMER
LET qt%=2:  REM quiz type=fill-in blank
LET qn%=1:  REM question 1
LET mc%=4:  REM number of choices/question
LET title$="empty"
LET last.pair%=0
LET max.pairs%=100
LET nu$="":  REM no spaces inside quotes
DIM q$(2),pr$(last.pair%,2),s%(last.pair%),mpc%(4)
```

Max.pairs% is an arbitrarily chosen maximum number of data pairs; you may raise or lower this number depending on how much memory your Macintosh has and how many database items you want to enter.

Last.pair% is the number of items currently in the database. For each database, q$( ) stores the two questions. Pr$(,) stores the data pairs. S%( ) determines the order in which data pairs are used in the quiz. Mpc%( ) is used during the formulation of multiple-choice questions.

Database and Quizmaster Menus

The following lines create the database and Quizmaster screens:

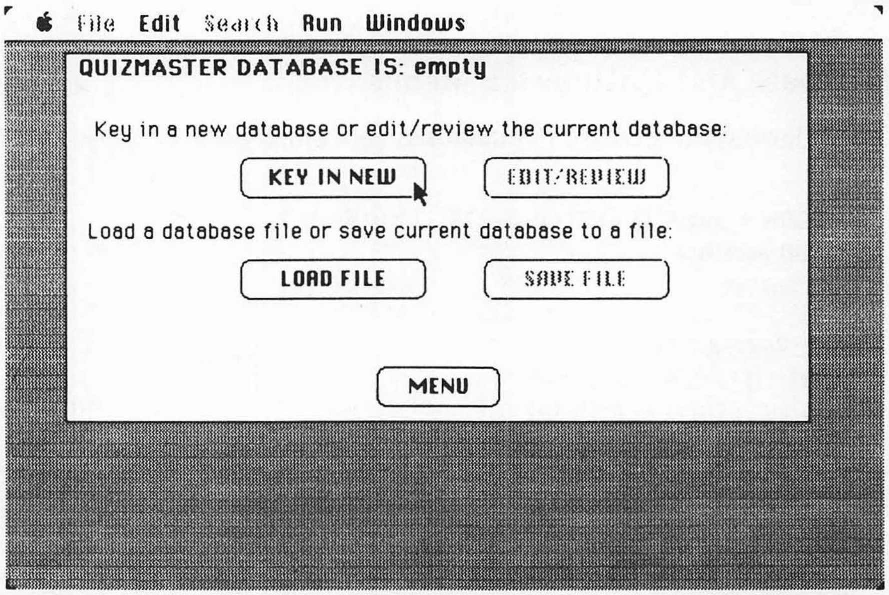
```
WINDOW 1,,(wx%(1),wy%(1))-(wx1%(1),wy1%(1)),3
GOSUB database
quizmaster:
CLS
CALL TEXTFACE(1)
PRINT " QUIZMASTER: ";title$
CALL MOVETO(12,by%(1)-12)
PRINT " Start quiz or work on the database."
BUTTON 1,ABS(last.pair%>0), "QUIZ", (bx%(1), by%(1))- (bx1%(1), by1%(1)),
       bt%(1)
BUTTON 2,1,"DATABASE", (bx%(2), by%(2))- (bx1%(2), by1%(2)), bt%(2)
BUTTON 3,1,"END", (bx%(12), by%(3))- (bx1%(12), by1%(3)), bt%(3)
WHILE DIALOG(0)<>1
WEND
FOR b%=1 TO 3
```

```
BUTTON CLOSE b%
NEXT b%
ON DIALOG(1) GOSUB select.quiz, database, quit.quiz
GOTO quizmaster
quit.quiz:
WINDOW CLOSE 1
WINDOW CLOSE 2
END
```

When you first start the program, its database is empty. The program thus immediately executes the database subroutine, which prompts you to key in a new database or load one from disk, as shown in Figure 11-1.

Upon return from the database subroutine, the program presents the main Quizmaster screen shown in Figure 11-4.

Various combinations of the button descriptors are used to generate the three buttons QUIZ, DATABASE, and END. For instance, the END button uses the horizontal values of button 12 and the vertical values of button 3.



QUIZMASTER DATABASE IS: empty

Key in a new database or edit/review the current database:

Load a database file or save current database to a file:

Figure 11-1. The screen that appears when you start the Quizmaster program

Notice that the QUIZ button is enabled only when the database is not empty (BUTTON 1,(ABS(last.pair%>0)...)).

Upon return from the select.quiz or database subroutines, the program redisplays the Quizmaster screen (GOTO quizmaster).

The Database Subroutine

These lines take over when you press DATABASE from the Quizmaster screen:

```
database:
CLS
CALL TEXTFACE(1)
PRINT " QUIZMASTER DATABASE IS: "; title$
CALL TEXTFACE(0)
CALL MOVETO(12,by%(1)-12)
PRINT " Key in a new database or edit/review the current database:"
CALL MOVETO(12,by%(3)-12)
PRINT "Load a database file or save current database to a file:"
BUTTON 1,1,bl$(1), (bx%(1), by%(1))- (bx1%(1), by1%(1)), bt%(1)
BUTTON 2,-(last.pair%>0), bl$(2), (bx%(2), by%(2))- (bx1%(2), by1%(2)),
      bt%(2)
BUTTON 3,1,bl$(3), (bx%(3), by%(3))- (bx1%(3), by1%(3)), bt%(3)
BUTTON 4,-(last.pair%>0), bl$(4), (bx%(4), by%(4))- (bx1%(4), by1%(4)),
      bt%(4)
BUTTON 5,1,"MENU",(bx%(12), by%(12))- (bx1%(12), by1%(12)), bt%(12)
WHILE DIALOG(0)<>1
WEND
FOR b%=1 TO 5
BUTTON CLOSE b%
NEXT b%
LET btn%=DIALOG(1)
ON btn% GOSUB new.db, edit.db, load.db, save.db
RETURN
```

Refer to Figure 11-1 again. Buttons 1 through 4 correspond to the KEY IN NEW, EDIT/REVIEW, LOAD FILE, and SAVE buttons on the database screen. The program waits for you to press one of the buttons, after which it executes the selected option and returns to the Quizmaster screen in Figure 11-4.

Notice that the EDIT/REVIEW and SAVE FILE buttons are enabled only when the database has at least one data pair (last.pair %>0).

Key In New Option

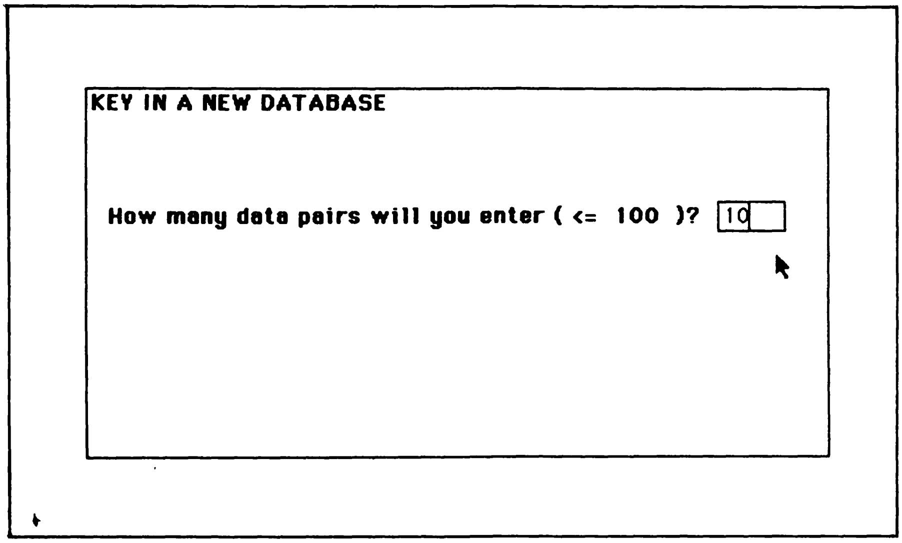
Pressing the KEY IN NEW button activates these lines:

```
new.db:
CLS
CALL TEXTFACE(1)
PRINT "KEY IN A NEW DATABASE"
CALL MOVETO(12,fy%(6)+12)
PRINT "How many data pairs will you enter ( <="; max.pairs%; ")?"
CALL TEXTFACE(0)
EDIT FIELD 1,STR$(last.pair%),(fx%(6),fy%(6))-(fx1%(6),fy1%(6))
siz.loop:
LET event%=0
WHILE event%<>2 AND event%<>6
LET event%=DIALOG(0)
WEND
LET x=INT(VAL(EDIT$(1)))
LET siz.set%=(x>=1) AND (x<=INT(max.pairs%))
IF siz.set% THEN siz.ok
BEEP
GOTO siz.loop
siz.ok:
EDIT FIELD CLOSE 1
LET last.pair%=x
ERASE pr$,s%
DIM pr$(last.pair%,2), s%(last.pair%)
LET title$="empty"
LET q$(1)=nu$
LET q$(2)=nu$
GOTO edit.db
```

Refer to Figure 11-2. The program prompts you to specify the number of data pairs you will enter so it can create the necessary arrays for data storage. The number you enter must be no greater than max.pairs%.

The Edit/Review Subroutine

When you complete the Key In New option or select the Edit/Review option, the following lines take over:



```
KEY IN A NEW DATABASE

  How many data pairs will you enter ( <=  100  )? 10
```

Figure 11-2. After selecting the Key In New option, you specify the number of data pairs in the database

```
edit.db:
CLS
CALL TEXTFACE(1)
PRINT "EDIT/REVIEW DATABASE:"
CALL MOVETO(fx%(1),fy%(1)-4)
PRINT "Title"
CALL MOVETO(fx%(2), fy%(2)-4)
PRINT "Question A"
CALL MOVETO(fx%(3),fy%(3)-4)
PRINT "Question B"
CALL TEXTFACE(0)
LET pno%=1
GOSUB display.pair
EDIT FIELD 3,q$(2),(fx%(3),fy%(3))-(fx1%(3),fy1%(3))
EDIT FIELD 2,q$(1),(fx%(2),fy%(2))-(fx1%(2),fy1%(2))
EDIT FIELD 1,title$,(fx%(1),fy%(1))-(fx1%(1),fy1%(1))
FOR n%= 1 TO 3
m%=n%+4
BUTTON n%, 1, bl$(m%), (bx%(m%)+18,by%(m%))- (bx1%(m%)+18, by1%(m%)),
      bt%(m%)
NEXT n%
```

These lines create the screen shown in Figure 11-3. In the figure, the title and questions have been filled in and the first seven data pairs have been entered. Notice the correspondence between the questions and entries:

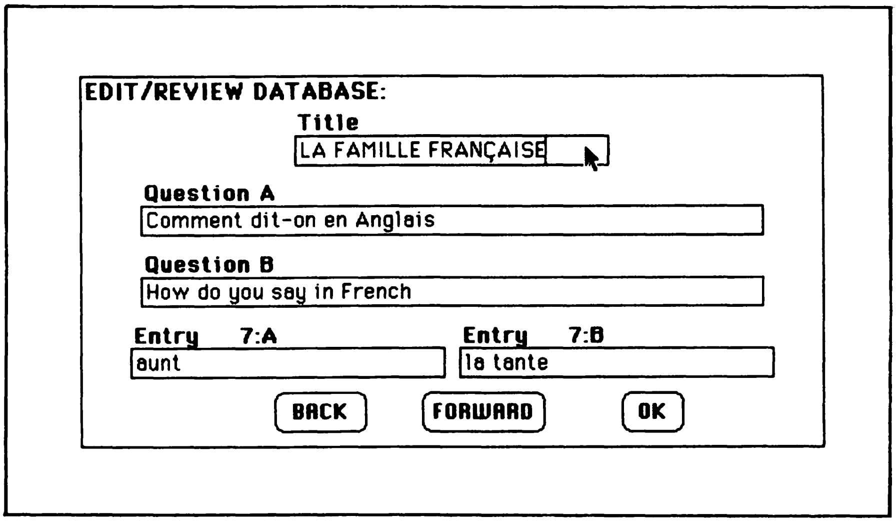
Entries marked "B" can be used to complete Question A, while entries marked "A" constitute answers to the completed questions.

Conversely, entries marked "A" can be used to complete Question B, while entries marked "B" constitute answers to the completed questions.

Each time you fill in one of the edit fields shown in Figure 11-3, you press ENTER, RETURN, or TAB to advance to the next field. Alternatively, you may skip around by pointing to a different field and clicking the mouse button. If you press ENTER, RETURN, or TAB while the insertion point is in Entry B, the program automatically displays the next data pair.

Pressing BACK causes the previous data pair to be displayed in Entries A and B; pressing FORWARD displays the next data pair.

Pressing OK ends the Edit/Review process.


EDIT/REVIEW DATABASE:

Title: LA FAMILLE FRANÇAISE

Question A: Comment dit-on en Anglais

Question B: How do you say in French

Entry 7:A: aunt

Entry 7:B: la tente

Figure 11-3. During entry of a new database or editing of an existing one, this screen shows you the data fields

Edit/Review Event Monitor

The following block monitors your actions within the Edit/Review window:

```
LET fld%=1
ed.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
ON event% GOTO ed.btn,ed.fld,ed.loop,ed.loop,ed.loop,nxt.fld,nxt.fld
ed.btn:
LET btn%=DIALOG(1)
ON btn% GOTO pr.back,pr.fwd,ed.ok
```

A dialog event of 1 corresponds to a button pressed; 2 to a mouse button click; 6 to pressing ENTER or RETURN; and 7 to pressing TAB. 3, 4, and 5 are ignored events. The ON event% statement responds to whichever event occurred.

In case of a button press, the ON btn% statement distinguishes between the BACK, FORWARD, and OK buttons.

The next lines execute all the Edit/Review options except for the last (OK):

```
pr.back:
LET pr$(pno%,1)=EDIT$(4)
LET pr$(pno%,2)=EDIT$(5)
IF pno%=1 THEN pno%=last.pair% ELSE pno%=pno%-1
GOTO select.pair
pr.fwd:
LET pr$(pno%,1)=EDIT$(4)
LET pr$(pno%,2)=EDIT$(5)
IF pno%=last.pair% THEN pno%=1 ELSE pno%=pno%+1
select.pair:
IF fld%=5 THEN LET fld%=4
GOSUB display.pair
```

```
EDIT FIELD fld%
GOTO ed.loop
ed.fld:
LET new.fld%=DIALOG(2)
IF fld%=new.fld% THEN ed.loop
LET fld%=new.fld%
EDIT FIELD fld%
GOTO ed.loop
nxt.fld:
IF event%>5 AND fld%=5 THEN pr.fwd
LET fld%=fld% MOD 5+1
EDIT FIELD fld%
GOTO ed.loop
```

Pr.back and pr.forward handle the Back and Forward options. Ed.fld handles requests to move to another field (mouse clicks in an edit field). Nxt.fld handles ENTER, RETURN, and TAB.

Edit fields 4 and 5 correspond to the fields under Entry A and Entry B. Pno% is the number of the pair currently displayed in these fields. If you press BACK or FORWARD, the program decrements or increments pno% (IF pno%=1 THEN... and IF pno%=last.pair% THEN...).

Ending the Edit/Review Session

The next routine responds to the OK button:

```
ed.ok:
LET title$=EDIT$(1)
LET q$(1)=EDIT$(2)
LET q$(2)=EDIT$(3)
LET pr$(pno%,1)=EDIT$(4)
LET pr$(pno%,2)=EDIT$(5)
FOR n%=1 TO 5
EDIT FIELD CLOSE n%
NEXT n%
FOR n%=1 TO 3
BUTTON CLOSE n%
NEXT n%
RETURN
```

This routine saves the data you've entered for the title, questions A and B, and the currently displayed Entries A and B. Then it closes the edit fields and buttons and ends with a RETURN.

The following subroutine is used at two different points during the Edit/Review process:

```
display.pair:
CALL TEXTFACE(1)
CALL MOVETO(fx%(4),fy%(4)-4)
PRINT  USING "Entry ***:A"; pno%
CALL MOVETO(fx%(5),fy%(5)-4)
PRINT  USING "Entry ***:B"; pno%
CALL TEXTFACE(0)
EDIT FIELD 5,pr$(pno%,2),(fx%(5),fy%(5))-(fx1%(5),fy1%(5))
EDIT FIELD 4,pr$(pno%,1),(fx%(4),fy%(4))-(fx1%(4),fy1%(4))
RETURN
```

These lines copy database entries A and B into edit fields 4 and 5.

Test Point 1

You have now entered enough of the program to test the Quizmaster screen, the database screen, the Key In New option, and the Edit/Review option.

First add these temporary lines in lieu of the remainder of the program:

```
load.db:
RETURN
save.db:
RETURN
```

Close the listing window. Get a printout of your work (use LLIST) and carefully check it against this book. Then run the program (COMMAND-R).

Initially, your screen should resemble Figure 11-1. Notice that the SAVE FILE and EDIT/REVIEW buttons are disabled (ghosted appearance).

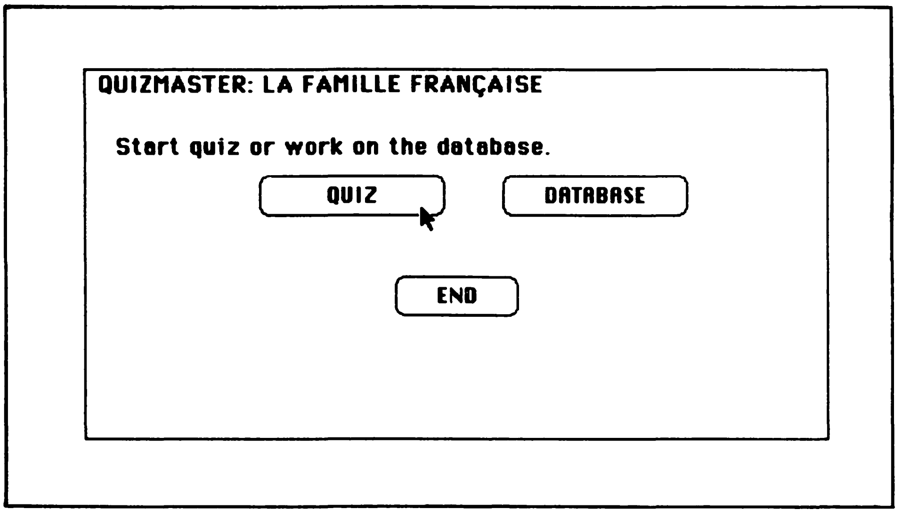
Pressing the LOAD FILE button should cause the program to return to the main menu. Pressing KEY IN NEW should give you the screen shown in Figure 11-2. Type the number 4 and press ENTER. Now the computer should show you a screen resembling the one shown in Figure 11-3. Initially, all the fields will be empty except for the title, which should read "empty."

Try entering data into each field. Try moving from one field to another using the ENTER key, TAB key, and the mouse pointer. Test the BACK and FORWARD buttons. The Entry A and B labels should cycle through the numbers 1 through 4 as you press FORWARD and BACK repeatedly.

Press OK. Your screen should now resemble Figure 11-4. The QUIZ button is enabled, but pressing it has no effect yet. Press DATABASE and you should see Figure 11-1 again, with the SAVE FILE and EDIT/REVIEW buttons enabled.

If something doesn't work properly, check the section of the program logic that controls the function you're trying to use. (Only the Database, End, Edit/Review, Key In New, and Menu options will work now.)

When you're ready to continue, stop the program, reopen the listing window, and delete these lines:


QUIZMASTER: LA FAMILLE FRANÇAISE

Start quiz or work on the database.

Figure 11-4. The Quizmaster main menu after a database has been loaded or typed in

```
load.db:
RETURN
save.db:
RETURN
```

Loading a Database

The following block of lines takes over when you select the Load File option:

```
load.db:
CLS
CALL TEXTFACE(1)
PRINT "LOAD A DATABASE FILE"
CALL TEXTFACE(0)
LET db.file$=FILES$(1,"TEXT")
IF db.file$=nu$ THEN RETURN
ON ERROR GOTO loadio.err
OPEN db.file$ FOR INPUT AS 1
INPUT#1, x
ON ERROR GOTO 0
LET x=INT(x)
IF x<1 OR x>INT(max.pairs%) THEN loadsiz.err
LET last.pair%=x
ERASE pr$,s%
DIM pr$(last.pair%,2),s%(last.pair%)
ON ERROR GOTO loadio.err
LINE INPUT#1, title$
LINE INPUT#1, q$(1)
LINE INPUT#1, q$(2)
ON ERROR GOTO 0
LET pno%=1
loaddb.loop:
ON ERROR GOTO loadio.err
LINE INPUT#1, pr$(pno%,1)
LINE INPUT#1, pr$(pno%,2)
ON ERROR GOTO 0
IF pno%=last.pair% THEN load.done
LET pno%=pno%+1
GOTO loaddb.loop
load.done:
CLOSE
RETURN
```

These lines present the screen shown in Figure 11-10. The FILES$ statement activates the dialog box shown in the center of the figure and shows all the text files available on the currently selected drive. (Use EJECT or DRIVE to switch disks or drives.) However, you should select only database files created with the program's Save File option (or files that are formatted the same way).

After opening the file, the program inputs a single number x, which indicates the number of pairs in the database file. If the number is within the allowable range, the program creates arrays pr$( ) and s%( ) and loads the title, the two question fragments, and the x data pairs.

In case of an error during the loading process, the following lines take over:

```
loadio.err:
CLOSE
IF ERR>=50 THEN load.disk.related
IF ERR<>6 AND ERR<>13 AND ERR<>23 THEN ON ERROR GOTO 0
load.disk.related:
BEEP
CALL TEXTFACE(1)
CALL MOVETO(12,wl%(1)\2)
PRINT "Unable to load file ";db.file$
CALL TEXTFACE(0)
RESUME ack.load.err:
loadsiz.err:
BEEP
CALL TEXTFACE(1)
CALL MOVETO(12,wl%(1)*3\8)
PRINT db.file$
PRINT PTAB(12); "contains"; x; "data pairs."
PRINT PTAB(12); "Allowable range is 1 -"; max.pairs%;"pairs."
CALL TEXTFACE(0)
ack.load.err:
LET last.pair%=0: ON ERROR GOTO 0
LET title$="empty"
BUTTON 1,1,bl$(7),(bx%(7),by%(7))-(bx1%(7),by1%(7))
WHILE DIALOG(0)<>1
WEND
BUTTON CLOSE 1
GOTO load.db
```

The program can handle disk errors and errors caused by incorrect data in the file (IF ERR>=50... and IF ERR< >6...). Any other type of error causes the program to stop with an error message from Microsoft BASIC.

Figure 11-11 shows the error message presented by the load.disk.related routine after a disk error occurs or when the file contains invalid data (such as a character string instead of a number for x).

The loadsiz.err routine gives a similar message when x is larger than the preset maximum max.pairs%.

After any error, the ack.load.err routine sets the database size indicator to 0 and the title to "empty."

Saving a Database

The next block handles the Save File option:

```
save.db:
CLS: ON ERROR GOTO 0
CALL TEXTFACE(1)
PRINT "SAVE DATABASE IN A FILE"
CALL TEXTFACE(0)
LET db.file$=FILES$(0,"Name the file:")
IF db.file$=nu$ THEN RETURN
open.for.save:
ON ERROR GOTO saveio.err
OPEN db.file$ FOR OUTPUT AS 1
PRINT#1,last.pair%
PRINT#1,title$
PRINT#1, q$(1)
PRINT#1, q$(2)
ON ERROR GOTO 0
LET pno%=1
savedb.loop:
ON ERROR GOTO saveio.err
PRINT#1, pr$(pno%,1)
PRINT#1, pr$(pno%,2)
ON ERROR GOTO 0
IF pno%=last.pair% THEN savedb.done
LET pno%=pno%+1
GOTO savedb.loop
savedb.done:
CLOSE 1
RETURN
```

The FILES$ function creates the dialog box shown in the center of Figure 11-9. After you enter the name of the output file, the program stores the data in the following sequence: number of data pairs, title, Question A, Question B, followed by the data pairs. Every data item is on a separate line in the output file.

If an error occurs while the program is attempting to save the data, the following lines are activated:

```
saveio.err:
CLOSE 1
IF ERR<50 THEN ON ERROR GOTO 0
BEEP
CALL TEXTFACE(1)
CALL MOVETO(12,w1%(1)\2)
PRINT "Unable to save database in file ";db.file$
CALL TEXTFACE(0)
BUTTON 1,1,b1$(7),(bx%(7),by%(7))-(bx1%(7),by1%(7))
WHILE DIALOG(0)<>1
WEND
BUTTON CLOSE 1
RESUME save.db
```

Again, only disk-related errors are handled; others cause the program to stop.

Test Point 2

After you have checked over the Load and Save portions of the program logic, you are almost ready to test them. But first type these lines in lieu of the remainder of the program.

```
select.quiz:
RETURN
```

Now close the listing window and run the program. Enter a short sample database (select Key In New). From the main menu select the Database option. Now try the Save File option. After the save is complete, select the Database option again and try the Load File option. After that, use Edit/Review to see if the data was loaded properly.

When you're ready to continue, stop the program, reopen the listing window, and delete these lines:

```
select.quiz:
RETURN
```

The Quiz Selection Menu

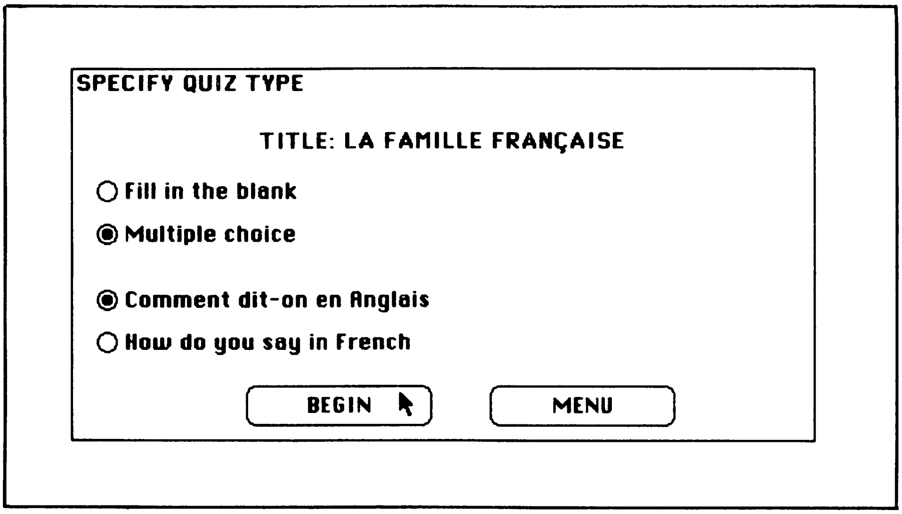
The following lines start the quiz process:

```
select.quiz:
CLS
CALL TEXTFACE(1)
PRINT "SPECIFY QUIZ TYPE"
CALL MOVETO((ww%(1)-WIDTH("TITLE: "+title$))\2,fy%(1)+12)
PRINT "TITLE: "; title$
LET mpc.on%=ABS(last.pair%>=mc%)   :REM 0 or 1
IF last.pair%<mc% THEN LET qt%=1   :REM can't do mpc, so do fib
LET sel%=1+ABS(qt%=2)   :REM 1 or 2
BUTTON 1, 3-sel%, bl$(8), (bx%(8)+12, by%(8))- (bx1%(8)+12, by1%(8)),
      bt%(8)
BUTTON 2,mpc.on%*sel%, bl$(9), (bx%(9)+12,by%(9))- (bx1%(9)+12,
      by1%(9)), bt%(9)
FOR n%=1 TO 2
LET sel%=1+ABS(qn%=n%)   :REM 1 or 2
LET m%=n%+9
LET bl$(m%)=q$(n%)
LET bx%(m%)=12
LET bx1%(m%)=12+18+WIDTH(bl$(m%))
BUTTON 2+n%,sel%,bl$(m%),(bx%(m%),by%(m%))- (bx1%(m%), by1%(m%)),
      bt%(m%)
NEXT n%
BUTTON 5,1,bl$(12),(bx%(3),by%(12))-(bx1%(3),by1%(12)),bt%(3)
BUTTON 6,1,"MENU",(bx%(4),by%(12))-(bx1%(4),by1%(12)),bt%(4)
```

These lines create the quiz selection menu shown in Figure 11-5. The multiple-choice button is enabled only if the database contains at least mc%=4 pairs, since a multiple-choice question has mc% choices (LET mpc.on% = (ABS(last.pair%>=mc%).

Next the program waits for you to select the options (Multiple choice or Fill in the blank, and question A or question B):

```
sq.loop:
WHILE DIALOG(0)<>1
WEND
```



SPECIFY QUIZ TYPE

TITLE: LA FAMILLE FRANÇAISE

○ Fill in the blank

[✓ ◉ Multiple choice]

[✓ ◉ Comment dit-on en Anglais]

○ How do you say in French

Figure 11-5. When you select the Quiz option, this menu prompts you to specify the quiz type

```
LET btn%=DIALOG(1)
ON btn% GOTO sw.qt, sw.qt, sw.qn, sw.qn, start.quiz
FOR b%=1 TO 6
BUTTON CLOSE b%
NEXT b%
WINDOW CLOSE 2
RETURN
```

The following lines handle requests to change the question format or type:

```
sw.qt:
BUTTON 2,(1+ABS(btn%=2))*mpc.on%
BUTTON 1, 1+ABS(btn%=1)
LET qt%=btn%
GOTO sq.loop
sw.qn:
BUTTON 3,1+ABS(btn%=3)
BUTTON 4, 1+ABS(btn%=4)
LET qn%=btn%-2
GOTO sq.loop
```

Starting the Quiz

When you press BEGIN, the following lines start the quiz:

```
start.quiz:
FOR n%=1 TO 6
BUTTON CLOSE n%
NEXT n%
CLS
CALL TEXTFACE(1)
PRINT "Shuffling the questions. One moment."
CALL TEXTFACE(0)
FOR n%=1 TO last.pair%
LET s%(n%)=0
NEXT n%
FOR n%=1 TO last.pair%
shuffle.loop:
LET pno%=INT(RND*last.pair%)+1
IF s%(pno%)<>0 THEN shuffle.loop
LET s%(pno%)=n%
NEXT n%
LET qc%=0
LET succ%=0
LET pct%=0
WINDOW 2,,(wx%(2),wy%(2))-(wx1%(2),wy1%(2)),3
GOSUB label.scorebox
GOSUB update.scorebox
WINDOW 1
```

First the program creates a random question sequence and stores it in s%( ). After shuffling, s%(1) contains the pair number of the first question, s%(2) contains the pair number of the second question, and so forth.

Next, the program initializes counters for questions tried (qc%) and questions answered successfully (succ%) and sets the percentage correct variable pct% to 0.

Then the program creates the scorebox, labels it, and fills in the initial scoring values.

The next lines get a question ready:

```
set.question:
LET qc%=qc%+1
LET question$=pr$(s%(qc%),3-qn%)
LET answer$=pr$(s%(qc%),qn%)
CLS
CALL TEXTFACE(1)
PRINT USING "Question ***:";qc%
CALL MOVETO(0,fy%(2)-40)
CALL LINE(ww%(1),0)
CALL MOVETO(0,fy%(7)+40)
CALL LINE(ww%(1),0)
CALL MOVETO(fx%(2),fy%(2)-16)
PRINT q$(qn%)
CALL MOVETO(fx%(2),fy%(2))
PRINT question$;"?"
CALL TEXTFACE(0)
ON qt% GOTO getanswer.fib,getanswer.mpc
```

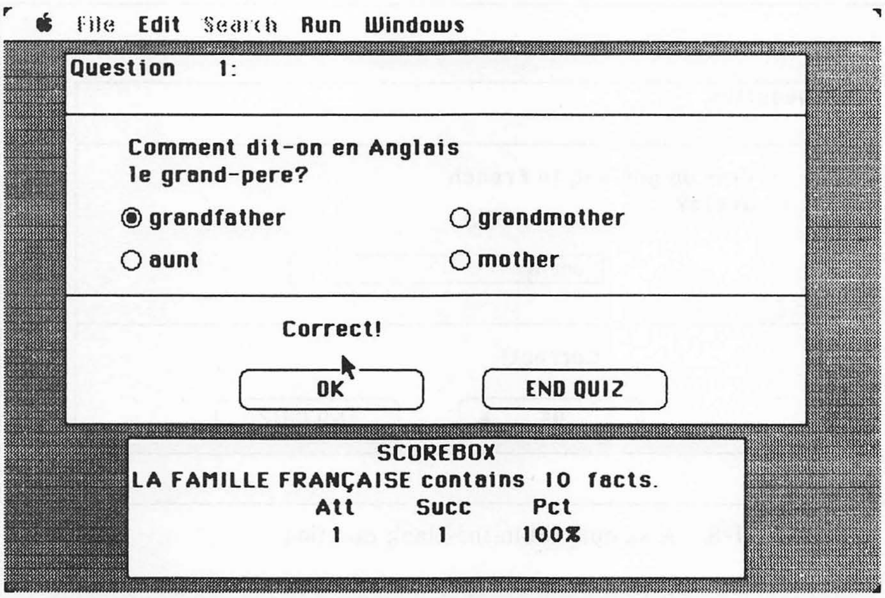
Question% is the data item that correctly completes the question. Answer% is the data item that correctly answers the question. After printing the question (see Figure 11-6 and Figure 11-7), the program branches either to the fill-in-the-blank routine (getanswer.fib) or the multiple-choice routine (getanswer.mpc).

Fill In the Blank

These next lines wait for you to fill in the blank, as shown in Figure 11-8:

```
getanswer.fib:
EDIT FIELD 1,"",(fx%(7),fy%(7))-(fx1%(7),fy1%(7))
LET event%=0
WHILE event%<>6
LET event%=DIALOG(0)
WEND
LET response$=EDIT$(1)
LET right%=(UCASE$(response$)=UCASE$(answer$))
GOTO respond
```

After entering your answer, you press RETURN or ENTER. Before comparing your answer with the correct answer, the program converts both answers to uppercase.



Question 1:

Comment dit-on en Anglais
le grand-pere?

[✓ grandfather] grandmother

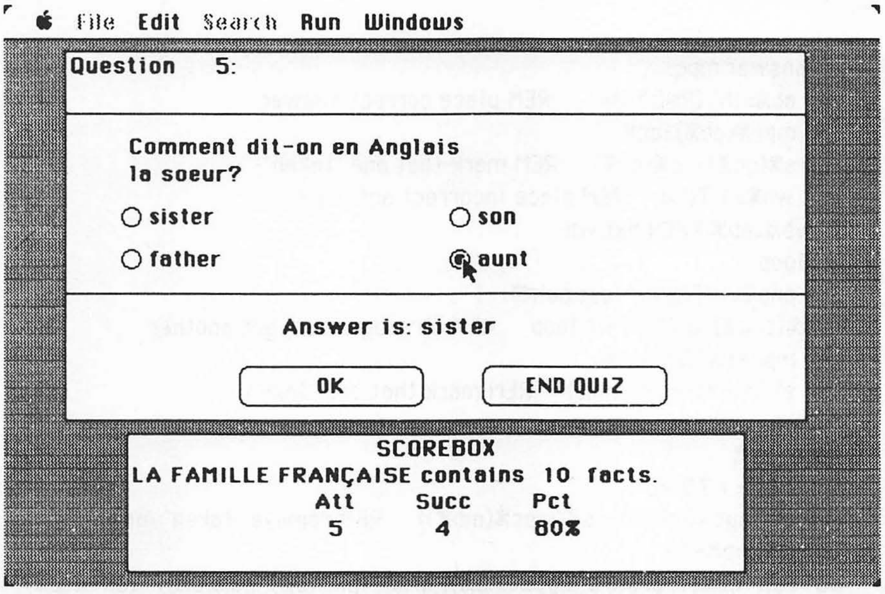
aunt mother

Correct!

SCOREBOX
LA FAMILLE FRANÇAISE contains 10 facts.

```
Att    Succ    Pct
 1      1      100%
```

Figure 11-6. A sample multiple-choice question. Notice that the score box keeps track of your past performance



Question 5:

Comment dit-on en Anglais
la soeur?

sister son

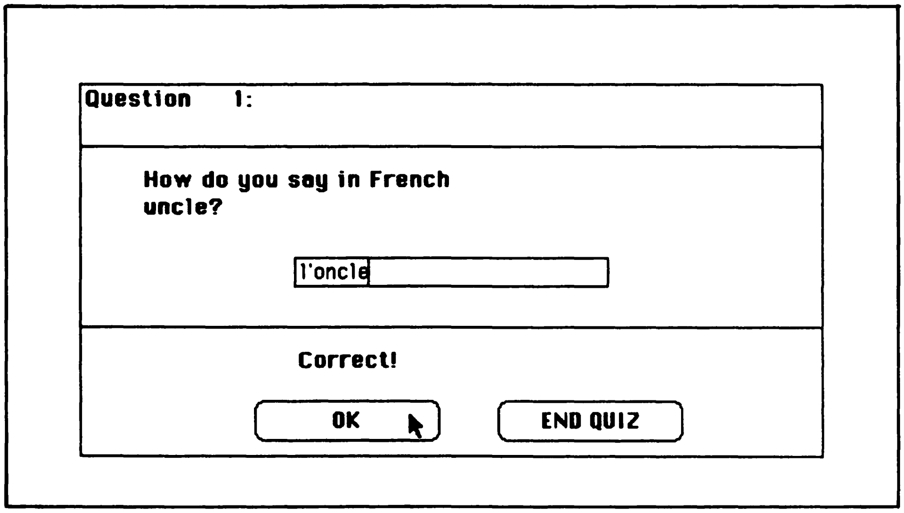
father [✓ aunt]

Answer is: sister

SCOREBOX
LA FAMILLE FRANÇAISE contains 10 facts.

```
Att    Succ    Pct
 5      4      80%
```

Figure 11-7. If you answer incorrectly, the program prints the correct answer


Question 1:

How do you say in French
uncle?

l'oncle

Correct!

Figure 11-8. A sample fill-in-the-blank question

Multiple Choice

The multiple-choice process is more involved than the process for filling in the blanks:

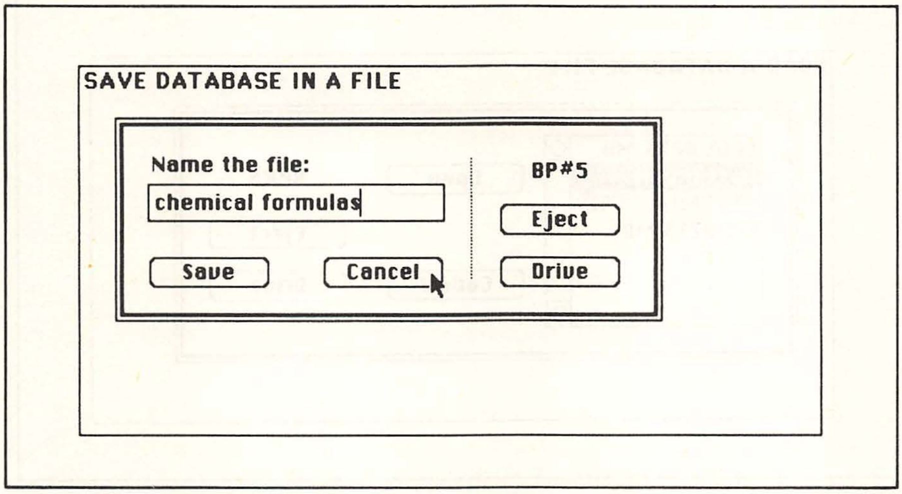
```
getanswer.mpc:
LET ab%=INT(RND*4)+1    :REM place correct answer
LET mpc%(ab%)=qc%
LET s%(qc%)=-s%(qc%)    :REM mark that one "taken"
FOR wb%=1 TO 4   :REM place incorrect answers
IF wb%=ab% THEN nxt.wb
wb.loop:
LET pno%=INT(RND*last.pair%)+1
IF s%(pno%)<0 THEN wb.loop    :REM already taken, get another
LET mpc%(wb%)=pno%
LET s%(pno%)=-s%(pno%)    :REM mark that one "taken"
nxt.wb:
NEXT wb%
FOR mb%=1 TO 4
LET s%(mpc%(mb%))=-s%(mpc%(mb%))    :REM remove "taken" mark
LET b%=mb%+12
BUTTON mb%,1, pr$(s%(mpc%(mb%)),qn%), (bx%(b%), by%(b%))- (bx1%(b%),
       by1%(b%)), bt%(b%)
```

```
NEXT mb%
LET event%=0
WHILE event%<>1
LET event%=DIALOG(0)
WEND
LET b%=DIALOG(1)
BUTTON b%,2
LET right%=(b%=ab%)
```

First the program randomly selects which of the four multiple-choice items will contain the correct answer (LET ab%=RND...). Then it randomly picks out other pairs to use for the other three multiple-choice items.

After receiving your answer to enter type of question, the following lines give you a response:

```
respond:
IF right% THEN LET succ%=succ%+1
LET pct%=INT(succ%/qc%*100+.5)
WINDOW 2
GOSUB update.scorebox
WINDOW 1
IF NOT right% THEN incorrect
```


SAVE DATABASE IN A FILE

Name the file:

chemical formulas

Figure 11-9. Screen appearance when you select the Save File option

```
FOR fq%=1 TO 4
SOUND 110*fq%,1
NEXT fq%
CALL TEXTFACE(1)
CALL MOVETO(fx%(7),fy%(7)+64)
PRINT "Correct!"
CALL TEXTFACE(0)
IF qt%=2 THEN waitack.mpc ELSE waitack.fib
incorrect:
SOUND 440,2
SOUND 110,2
CALL TEXTFACE(1)
CALL MOVETO(fx%(7),fy%(7)+64)
PRINT "Answer is: ";answer$
CALL TEXTFACE(0)
IF qt%=2 THEN waitack.mpc ELSE waitack.fib
```

The succ% and pct% variables are incremented if needed, and the program updates the scorebox accordingly. In case of a correct answer, the program plays an ascending sequence of notes; in the case of an error, the program plays a descending couplet and announces the correct answer.

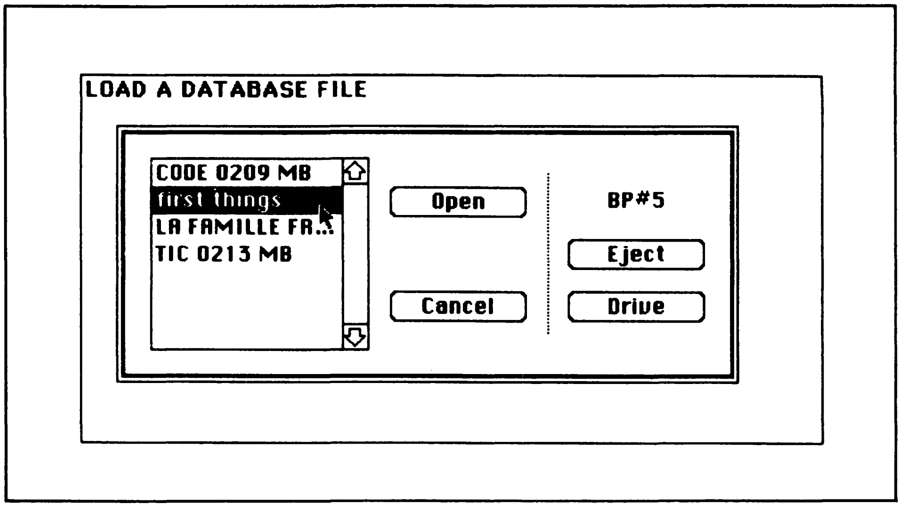
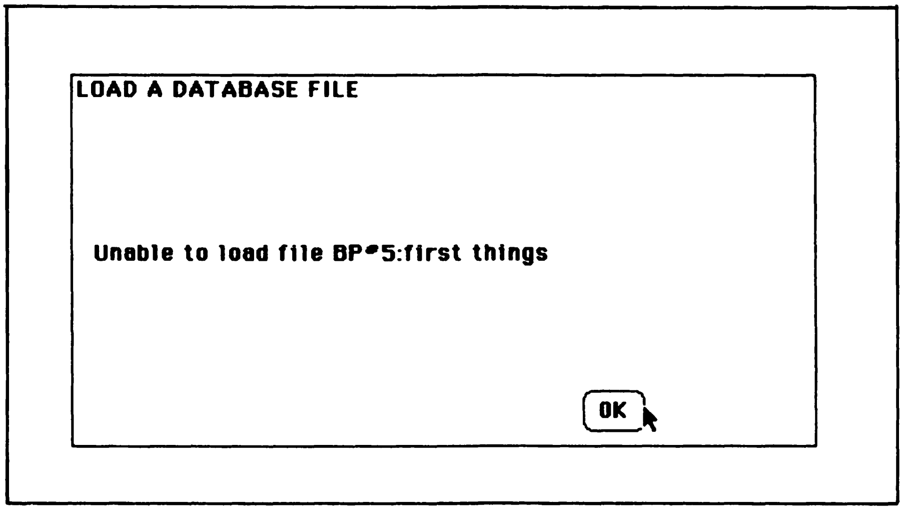


Figure 11-10. Screen appearance when you select the Load File option


LOAD A DATABASE FILE

Unable to load file BP*5:first things

Figure 11-11. An error message like this appears when a disk error occurs during loading

Next the program waits for you to acknowledge the response to your latest guess. The process is different for fill-in-the-blank tests and multiple-choice tests.

The following lines wait after the fill-in-the-blank test:

```
waitack.fib:
BUTTON 1,1,"OK",(bx%(3),by%(12))-(bx1%(3),by1%(12)),bt%(3)
BUTTON 2,1,"END QUIZ",(bx%(4),by%(12))-(bx1%(4),by1%(12)),bt%(4)
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
BUTTON CLOSE 1
BUTTON CLOSE 2
EDIT FIELD CLOSE 1
IF event%=6 OR DIALOG(1)=1 THEN nxt.question
GOTO select.quiz
```

Pressing ENTER, RETURN, or one of the screen buttons ends the loop.

The following lines wait after a multiple-choice test:

```
waitack.mpc:
BUTTON 5,1,"OK",(bx%(3),by%(12))-(bx1%(3),by1%(12)),bt%(3)
BUTTON 6,1,"END QUIZ",(bx%(4),by%(12))-(bx1%(4),by1%(12)),bt%(4)
wm.loop:
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
IF event%=6 THEN continue.mpc
LET b%=DIALOG(1)
IF b%<5 THEN wm.loop
continue.mpc:
FOR btn%=1 TO 6
BUTTON CLOSE btn%
NEXT btn%
IF b%=6 THEN select.quiz
nxt.question:
IF qc%<last.pair% THEN set.question
GOTO select.quiz
```

The nxt.question routine takes over upon completion of every quiz. These lines ensure that every question is asked before the program ends the quiz (IF qc%<last.pair%...).

The Scorebox

The following subroutine takes care of the scorebox:

```
label.scorebox:
CLS
CALL TEXTFACE(1)
PRINT PTAB((ww%(2)-WIDTH("SCOREBOX"))\2); "SCOREBOX"
PRINT title$; " contains"; last.pair%; "facts."
REM      123456789012345678
LET h$="  Att      Succ      Pct"
LET center%=(ww%(2)-WIDTH(h$))\2
PRINT PTAB(center%); h$
CALL TEXTFACE(0)
RETURN
```

When entering these lines, use the line REM 12345... as a guide to help you space out the column headings "Att   Succ   Pct%".

Finally, here are the lines that update the scorebox:

```
update.scorebox:
LOCATE 4,1
CALL TEXTFACE(1)
REM                         12345678901234567890
PRINT PTAB(center%); USING "***    ***    ***%"; qc%;succ%;pct%
CALL TEXTFACE(0)
RETURN
```

Again, use the line REM 12345... as a guide to entering the spacing between numbers correctly.

—Testing and Using the Program

Carefully check the new parts of the program, beginning with the select.quiz routine. If your version doesn't work quite right, try to identify the portion of the program that is supposed to accomplish the desired function.

Enter a complete, usable database such as States and Capitals or English and metric units. Try to use all the options from all the menus.

If you have trouble matching up the question fragments with the data pairs, just remember: incomplete question A is completed by a field from entry B and *answered* by the corresponding field from Entry A; incomplete question B is completed by a field from entry A and answered by the corresponding field from Entry B.

This chapter has been adapted from "The Quiz Master" by George Stewart, appearing in the August 1983 issue of *Popular Computing* magazine. Copyright © 1983 BYTE Publications, Inc. Used with permission of BYTE Publications, Inc.

Chapter 12

Speed Math

Do any of these situations sound familiar?

You're standing at the grocery checkout counter trying to double-check the cashier and the cash register, but you just can't keep up.

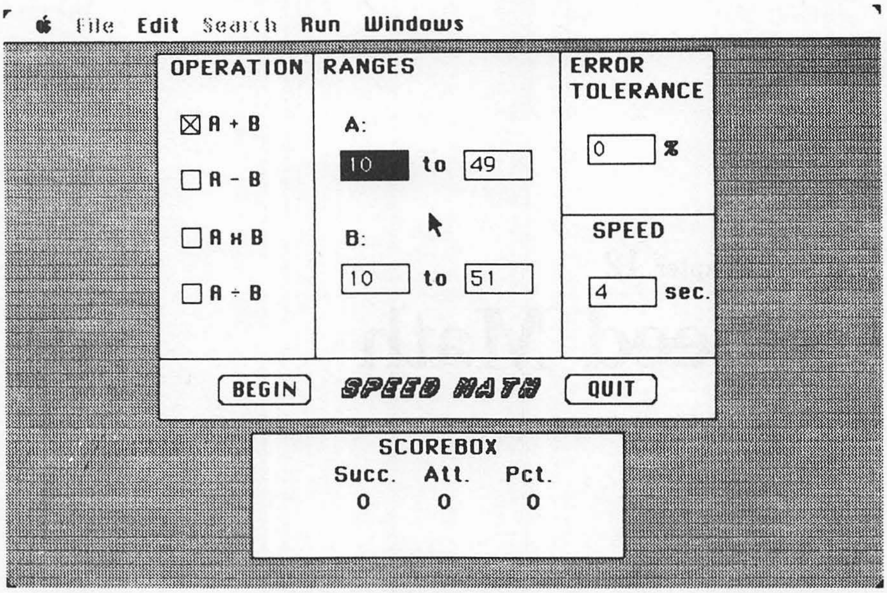
You're speeding down the highway, calculating your gas mileage, but you run out of fuel before you arrive at the answer.

You're at a dinner party and the person next to you starts talking about the national defense budget. You'd like to state the figure on a per capita basis, but the conversation has moved to French wines by the time you have the problem worked out.

This chapter's program can help you master situations that require quick mental calculations. The method used is timed drill and practice. You can practice any of the four basic arithmetic operations over any range of positive whole numbers, adjusting the time limit from one second to two minutes. You can even set an error tolerance of up to 25 percent to help you learn to make quick estimates.

—Speed Math Operation

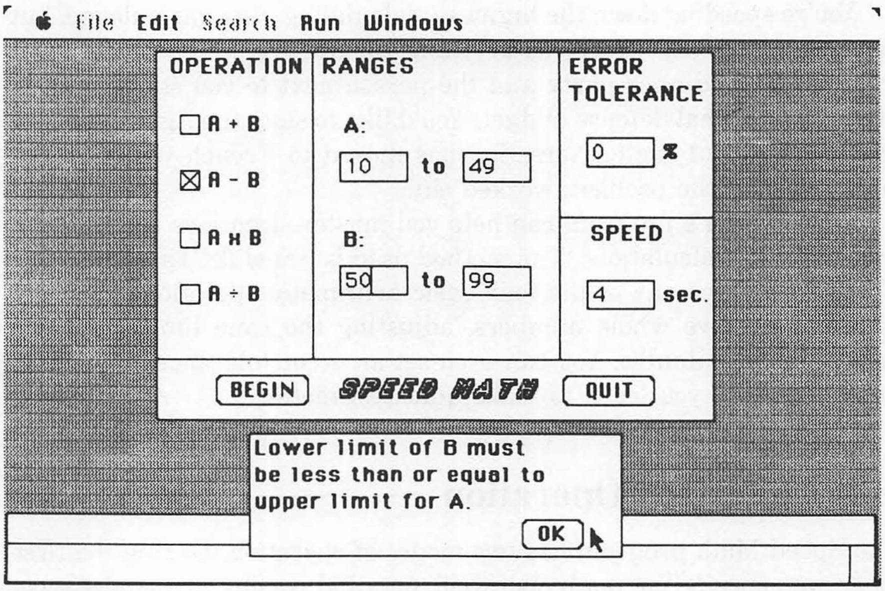
The Speed Math program has two modes of operation. During the first mode, you specify the math operation, operand ranges, error tolerance, and time limit (see Figures 12-1 and 12-2).



SCOREBOX

Succ.	Att.	Pct.
0	0	0

Figure 12-1. The initial Speed Math screen, showing the specification box and scorebox



Lower limit of B must be less than or equal to upper limit for A.

Figure 12-2. Specification box with an error message

During the second mode, the program generates a series of incomplete equations for you to fill in. The program keeps score for you and gives you new equations until you quit or go back to the specification mode (see Figures 12-3 and 12-4).

—The Program

The program starts out with descriptive data for the windows, buttons, and edit fields.

Here are the window descriptors:

```
REM Window descriptors
DATA 2
REM  wide  long   left  top
DATA 4.500, 3.000, 1.250, 0.375
DATA 3.000, 1.000, 2.000, 3.500
```

The second DATA statement describes the larger window (specification box and test box); the third describes the smaller window (scorebox). All measurements are in inches.

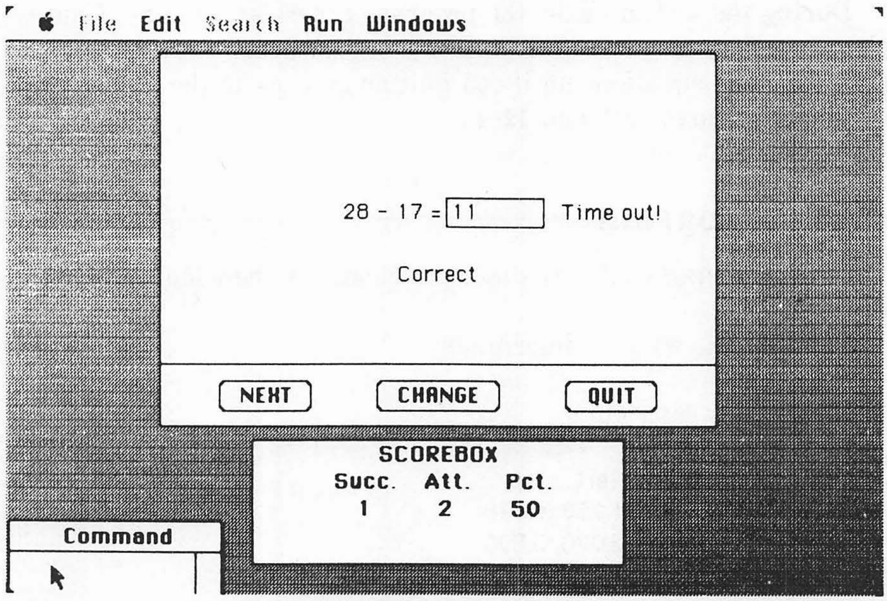
Here are the button descriptors:

```
REM Button descriptors
DATA 9
REM  label  wide  long   hzone vzone type
DATA A + B, 0.750, 0.208, 0.042, 0.167, 2
DATA A - B, 0.750, 0.208, 0.042, 0.333, 2
DATA A x B, 0.750, 0.208, 0.042, 0.500, 2
DATA A ÷ B, 0.750, 0.208, 0.042, 0.667, 2
DATA BEGIN,0.750, 0.250, 0.125, 0.958, 1
DATA QUIT,  0.750, 0.250, 0.875, 0.958, 1
DATA NEXT,  0.750, 0.250, 0.125, 0.958, 1
DATA CHANGE, 1.000, 0.250, 0.500, 0.958, 1
DATA OK, 0.500, 0.250, 0.875, 0.917, 1
```

To key in the division symbol in the fifth DATA statement, hold down the OPTION key and type a slash (/).

Hzone and vzone describe a button's position within a window; for instance, hzone=0.250, vzone 0.500 indicates a button one-fourth of the way across and one-half of the way down the window.

Here are the descriptors for the edit fields:

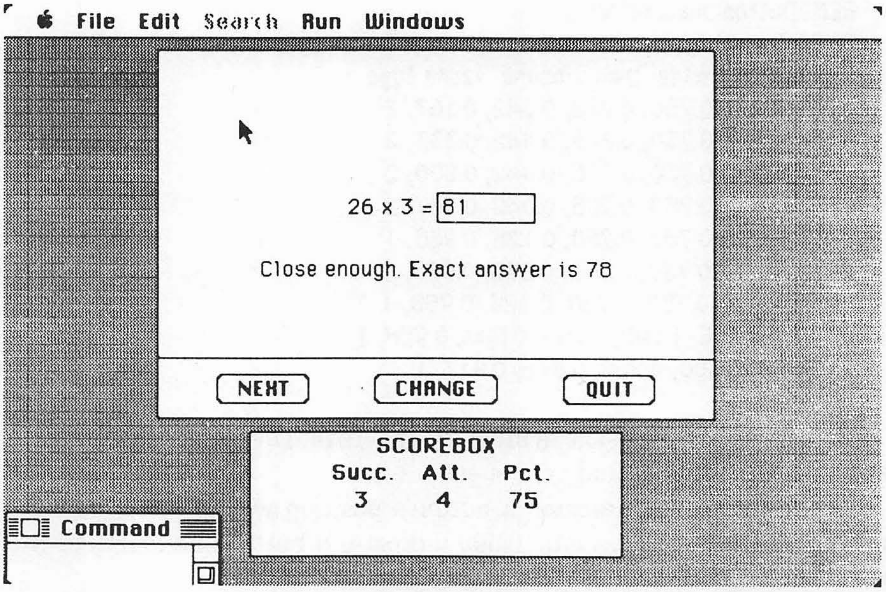


28 - 17 = 11 Time out!

Correct

SCOREBOX
Succ. Att. Pct.
1 2 50

Figure 12-3. The test box with a time out message. The correct answer is acknowledged even though time ran out before the operator pressed ENTER or RETURN



26 x 3 = 81

Close enough. Exact answer is 78

SCOREBOX
Succ. Att. Pct.
3 4 75

Figure 12-4. The test box showing the response to an answer that is within the error tolerance

```
REM Field descriptors
DATA 7
REM  wide  long  hzone vzone
DATA 0.500, 0.208, 0.375, 0.292
DATA 0.500, 0.208, 0.625, 0.292
DATA 0.500, 0.208, 0.375, 0.625
DATA 0.500, 0.208, 0.625, 0.625
DATA 0.500, 0.208, 0.875, 0.250
DATA 0.500, 0.208, 0.875, 0.667
DATA 0.750, 0.208, 0.750, 0.417
```

The first four DATA statements after the REM define edit fields for the lower and upper limits of operands A and B. The next two DATA statements define fields for the error tolerance and speed. The final edit field defines an answer field for the math test.

The following lines read in the window data and convert it from inches to screen points (pixels):

```
READ nw%
DIM ww%(nw%), wl%(nw%), wx%(nw%), wy%(nw%), wx1%(nw%), wy1%(nw%)
FOR n%=1 TO nw%
READ inches.wide,inches.long,ulcx,ulcy
LET ww%(n%)=inches.wide*72
LET wl%(n%)=inches.long*72
LET wx%(n%)=ulcx*72
LET wy%(n%)=ulcy*72
LET wx1%(n%)=wx%(n%)+ww%(n%)
LET wy1%(n%)=wy%(n%)+wl%(n%)
WINDOW n%,,(wx%(n%),wy%(n%))-(wx1%(n%),wy1%(n%)),3
NEXT n%
```

The next lines do the same for the buttons:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
READ bl$(n%),inches.wide,inches.long,h.zone,v.zone,bt%(n%)
IF n%=9 THEN cw%=2 ELSE cw%=1
LET bx%(n%)=(ww%(cw%)-inches.wide*72)*h.zone
LET by%(n%)=(wl%(cw%)-inches.long*72)*v.zone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```

Notice that button 9 is handled a little differently from the others. Since it appears in the smaller window, window 2, its actual position must be calculated using the dimensions of window 2 rather than the large window, window 1 (IF n%=9 THEN cw%=2 ELSE cw%=1).

Finally, the following lines load the data for the edit fields:

```
READ nf%
DIM fx%(nf%),fy%(nf%),fx1%(nf%),fy1%(nf%)
FOR n%=1 TO nf%
READ inches.wide,inches.long,h.zone,v.zone
LET fx%(n%)=(ww%(1)-inches.wide*72)*h.zone
LET fy%(n%)=(wl%(1)-inches.long*72)*v.zone
LET fx1%(n%)=fx%(n%)+inches.wide*72
LET fy1%(n%)=fy%(n%)+inches.long*72
NEXT n%
```

Constants and Parameters

The next block of lines creates the arrays used by the program logic:

```
RANDOMIZE TIMER
DIM l(2),u(2),a(2),op$(4),mc%(5),ml$(5,3)
DEF FNstrip$(x)=RIGHT$(STR$(x),LEN(STR$(x))-1)
READ op%, l(1), u(1), l(2), u(2), er, tl
DATA 1, 10, 49, 10, 51, 0, 4
FOR j%=1 TO 4
READ op$(j%)
NEXT j%
DATA +,-,x,÷
```

Arrays l( ) and u( ) store the lower and upper limits for operands A and B. Op$( ) stores the operator symbols. Mc%( ) and ml$( ) store data used to generate error messages.

The function FNstrip$ removes the leading space from a positive number or the sign from a negative number.

The DATA statement sets initial values for the operand ranges u( ) and l( ), error tolerance er, and time limit tl. After you have gotten the program working properly, you can adjust these starting values to better suit your interests and skills.

Again, to key in the division symbol, type OPTION-/.

The next lines read in the error messages that are sometimes needed during the specification mode:.

```
FOR msg%=1 TO 5
READ mc%(msg%)
FOR l%=1 TO mc%(msg%)
READ ml$(msg%,l%)
NEXT l%,msg%
DATA 3
DATA For operands A and B
DATA 0<=lower limit<=upper limit
DATA upper limit<=9999
DATA 3
DATA Lower limit of B must
DATA be less than or equal to
DATA upper limit for A.
DATA 3
DATA Lower limit of B must
DATA be greater than 0 for
DATA division.
DATA 2
DATA Error tolerance must be
DATA between 0 and 25 percent.
DATA 3
DATA Time limit must be
DATA greater than zero and
DATA less than 180.
```

There are five different error types and a different message for each. Mc%( ) stores the number of lines in each message. Ml$(,) stores the message text.

One of the messages appears in Figure 12-2.

The next block defines a few additional constants and counters:

```
LET yes%=(1=1)
LET no%=(1=0)
LET one.spc$=" "   :REM one space inside quotes
LET att%=0
LET succ%=0
LET pct%=0
```

Att%, succ%, and pct% keep track of the number of attempts, correct answers, and the percentage of accuracy.

Setting Up Windows

The next block sets up the scorebox and specification box:

```
WINDOW 2
CALL TEXTFACE(1)
GOSUB label.scorebox
WINDOW 1
change.parameters:
REM Set up the parameter window
LINE (0,wl%(1)*10/12)-STEP(ww%(1),0),1
LINE (ww%(1)*1.25/4.5,0)-STEP(0,wl%(1)*10/12)
LINE (ww%(1)*3.25/4.5,0)-STEP(0,wl%(1)*10/12)
LINE (ww%(1)*3.25/4.5,fy%(6)-38)-STEP(ww%(1)*1.25/4.5,0)
```

The label.scorebox subroutine is presented later. The LINE commands divide the specification box into five regions.

The following lines print the text labels in the specification box.

```
CALL TEXTMODE(1)
CALL TEXTFACE(2+16+64)   :REM italic shadow extended
LET title.main$="SPEED MATH"
LET t1.tab%=(ww%(1)-WIDTH(title.main$))\2
CALL MOVETO(t1.tab%,by%(5)+12)
PRINT title.main$;
CALL TEXTFACE(1)
LOCATE 1,1
PRINT " OPERATION"; PTAB(fx%(1)-12); "RANGES"; PTAB(fx%(5)-12);
      "ERROR"
PRINT PTAB(fx%(5)-12); "TOLERANCE"
CALL MOVETO(fx%(6),fy%(6)-24)
PRINT "SPEED"
CALL MOVETO(fx1%(5)+6,fy%(5)+12)
PRINT "%"
CALL MOVETO(fx1%(6)+6,fy%(6)+12)
PRINT "sec."
CALL MOVETO(fx%(1),fy%(1)-12)
PRINT "A:"
CALL MOVETO(fx%(2)-WIDTH("to")-12,fy%(2)+12)
```

```
PRINT "to "
CALL MOVETO(fx%(3),fy%(3)-12)
PRINT "B:"
CALL MOVETO(fx%(4)-WIDTH("to")-12,fy%(4)+12)
PRINT "to "
CALL TEXTFACE(0)
```

Monitoring the Specification Box

The next few blocks handle the specification box dialog. The first sets up the six edit fields shown in Figure 12-1:

```
EDIT FIELD 6,FNstrip$(tl),(fx%(6),fy%(6))-(fx1%(6),fy1%(6))
EDIT FIELD 5,FNstrip$(er),(fx%(5),fy%(5))-(fx1%(5),fy1%(5))
EDIT FIELD 4,FNstrip$(u(2)),(fx%(4),fy%(4))-(fx1%(4),fy1%(4))
EDIT FIELD 3,FNstrip$(l(2)),(fx%(3),fy%(3))-(fx1%(3),fy1%(3))
EDIT FIELD 2,FNstrip$(u(1)),(fx%(2),fy%(2))-(fx1%(2),fy1%(2))
EDIT FIELD 1,FNstrip$(l(1)),(fx%(1),fy%(1))-(fx1%(1),fy1%(1))
```

FNstrip$ puts the current value of each parameter (without a leading blank space) into the appropriate field.

The following block creates the six specification box buttons:

```
FOR n%=1 TO 4
BUTTON n%,1-(op%=n%), bl$(n%),(bx%(n%), by%(n%))- (bx1%(n%), by1%(n%)),
      bt%(n%)
NEXT n%
FOR n%=5 TO 6
BUTTON n%,1,bl$(n%), (bx%(n%), by%(n%))- (bx1%(n%), by1%(n%)), bt%(n%)
NEXT n%
```

Buttons 1 through 4 correspond to the operation selectors. The expression 1−(op%=n%) allows only one button to be selected (the one corresponding to the current operation op%).

The next lines wait for a dialog event (mouse click, ENTER, RETURN, TAB, or a button press):

```
par.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
```

```
WEND
ON event% GOTO par.btn, ed.fld, par.loop, par.loop, par.loop, nxt.fld,nxt.fld
```

Dialog events 3, 4, and 5 (change windows, click close box, and refresh output window) are ignored. Other dialog events are handled in the following block:

```
par.btn:
LET btn%=DIALOG(1)
IF btn%=5 THEN begin.quiz
IF btn%=6 THEN end.quiz
BUTTON op%,1
BUTTON btn%,2
LET op%=btn%
GOTO par.loop
ed.fld:
LET new.fld%=DIALOG(2)
IF new.fld%=fld% THEN par.loop
LET fld%=new.fld%
EDIT FIELD fld%
GOTO par.loop
nxt.fld:
LET fld%=fld% MOD 6+1
EDIT FIELD fld%
GOTO par.loop
end.quiz:
WINDOW CLOSE 2
WINDOW CLOSE 1
END
```

The par.btn routine takes over when you press one of the buttons (A+B, A−B, A×B, A÷B, BEGIN, or QUIT). Ed.fld takes over when you click the mouse button inside an edit field; it causes that field to become active. When you press ENTER, RETURN, or TAB, the nxt.fld routine activates the next field on the screen.

Checking Parameters

When you select the BEGIN button, the program checks the contents of all six edit fields for validity. First the program checks the lower bounds for A and B:

```
begin.quiz:
FOR n%=1 TO 3 STEP 2    :REM check lower limits for A and B
GOSUB get.fvalue
IF  ok% THEN lv.ok
LET err.fld%=n%
LET n%=3
GOTO nxt.lv
lv.ok:
LET l((n%+1)\2)=x
nxt.lv:
NEXT n%
IF NOT ok% THEN prm.err
```

Get.fvalue converts the string value from edit field n% into the number x. It also checks to see whether x is a nonnegative whole number less than 10000. If so, ok%=yes%. If not, ok%=no%.

If the value is within range, it is stored in). Otherwise, the parameter error routine prm.err takes over.

Next the program checks the upper bounds for A and B and the error tolerance and time limit:

```
FOR n%=2 TO 4 STEP 2    :REM check upper lim,  A and B
GOSUB get.fvalue
IF ok% AND x>=l((n%+1)\2) THEN u.gt.l
LET ok%=no%
LET err.fld%=n%
LET n%=4
GOTO nxt.uv
u.gt.l:
LET u((n%+1)\2)=x
nxt.uv:
NEXT n%
IF NOT ok% THEN prm.err
IF op%=2 AND u(1)<l(2) THEN sub.err    :REM prevent negative answers
IF op%=4 AND l(2)=0 THEN div.err    :REM prevent division by zero
LET n%=5    :REM check tolerance setting
GOSUB get.fvalue
IF NOT ok% OR x>25 THEN tol.err
LET er=x
LET n%=6    :REM check speed setting
GOSUB get.fvalue
```

```
IF x<1 OR x>=180 THEN time.err
LET tl=x
GOTO select.a.problem
```

The upper bound must be between 0 and 9999 and must also be no smaller than the corresponding lower bound (IF ok% AND x>= 1((n%+1)\2) THEN u.gt.1). If it is not, the parameter error routine takes over.

In the case of subtraction (op%=2) or division (op%=4), the program must make a couple of extra checks to eliminate the possibility of negative remainders or division by zero.

If there are no errors, the program continues with the select.a.problem routine presented later.

The next block contains all the specification box error handlers:

```
prm.err:
LET fld%=err.fld%
LET err.type%=1
GOTO err.msg
sub.err:
LET fld%=3
LET err.type%=2
GOTO err.msg
div.err:
LET fld%=3
LET err.type%=3
GOTO err.msg
tol.err:
LET fld%=5
LET err.type%=4
GOTO err.msg
time.err:
LET fld%=6
LET err.type%=5
GOTO err.msg
```

The five types of errors are parameter, negative remainder (subtraction), division by zero, error tolerance out of range, and time limit out of range. These types of error are handled in a similar manner. The program records the field that triggered the error, sets the error type, and then transfers control to the following err.msg routine:

```
err.msg:
EDIT FIELD fld%
WINDOW 2
CLS
BEEP
FOR 1%=1 TO mc%(err.type%)
PRINT ml$(err.type%,1%)
NEXT 1%
GOSUB wait.ok
GOSUB label.scorebox
WINDOW 1
GOTO par.loop
```

The program selects the field causing the error. Then it prints an appropriate message in the scorebox window and waits for you to press the OK button, ENTER, or RETURN (see Figure 12-2). After that, the program restores the scorebox information and goes back to the specification box monitor loop.

To complete the logic for the specification mode of the program, type in these auxiliary subroutines:

```
get.fvalue:
LET x=VAL(EDIT$(n%))
LET ok%=(x=INT(x)) AND (x>=0) AND (x<=9999)
RETURN
wait.ok:
BUTTON 1,1,b1$(9),(bx%(9),by%(9))-(bx1%(9),by1%(9)),bt%(9)
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
BUTTON CLOSE 1
RETURN
```

Get.fvalue gets a number from an edit field. Wait.ok puts an OK button in the window and waits for you to click on it or to press RETURN or ENTER.

Test Point

To test the specification mode logic, type in these temporary lines in lieu of the test mode logic:

```
select.a.problem:
END
label.scorebox:
CLS
RETURN
```

Now print the program by typing LLIST in the command window. Check the printout carefully against the program lines in this chapter, and correct any errors you find. Close the listing window and run the program.

You should see a screen similar to Figure 12-1, except that the scorebox window will be blank.

Try to select each of the operator buttons. Try various illegal range settings:

- upper limit < lower limit
- any negative value
- any number greater than 9999
- for subtraction, lower limit operand B > upper limit A
- for division, 0 = lower limit operand B.

Press BEGIN to see how the program responds to the illegal range setting. You should see an error message in a dialog box (see Figure 12-2).

Try various illegal settings for the error tolerance and time limit:

- error tolerance < 0 or error tolerance > 25
- time limit < 1 or time limit >= 180.

After you enter an illegal setting, press BEGIN to see how the program responds. You should see an appropriate error message.

Set all the parameters to valid values, and press BEGIN. The program should end without any error messages.

Math Test Mode

Now you can begin typing in the math test logic. Start by opening the listing window and deleting these lines:

```
select.a.problem:
END
label.scorebox:
```

```
CLS
RETURN
```

The following lines take over after the program has verified that all of the parameter settings are valid:

```
select.a.problem:
WINDOW CLOSE 1
LET att%=0  :REM reset scorebox counters
LET succ%=0
LET pct%=0
WINDOW 2
GOSUB label.scorebox
WINDOW 1
next.problem:
FOR j%=1 TO 2
LET ng=u(j%)-l(j%)+1
select.random:
LET nr=INT(RND*ng)+l(j%)
IF j%=2 AND op%=2 AND nr>a(1) THEN LET j%=1: GOTO select.random
IF j%=2 AND op%=4 AND nr=0 THEN select.random    :REM can't ÷ 0
LET a(j%)=nr
NEXT j%
```

The scorebox counters are reset to zero at the start of each new test. Then, for operands A and B, the program randomly selects a number nr that is within the specified range. In the case of subtraction or division, the program makes a further check to prevent negative differences (subtraction) or division by zero (IF j%=2 AND op%=4...).

Given values for A and B and operation op%, the following block computes the result:

```
ON op% GOTO add,subtract,multiply,divide
add:
LET r=a(1)+a(2)
GOTO show.equation
subtract:
LET r=a(1)-a(2)
GOTO show.equation
multiply:
LET r=a(1)*a(2)
GOTO show.equation
```

```
divide:
LET r=a(1)/a(2)
```

The variable r holds the correct result for the indicated operation.

Showing the Equation

The following lines create the test box (as shown in Figures 12-3 and 12-4):

```
show.equation:
LINE (0,wl%(1)*10/12)-STEP(ww%(1),0),1
CALL MOVETO(6,wl%(1)*10/12+24)
PRINT "Type in the answer & press Enter or Return.";
FOR j%=1 TO 3000
NEXT j%
LET tr=ABS(r*er/100)    :REM total allowable error
LET eq$=STR$(a(1))+one.spc$+op$(op%)+STR$(a(2))+one.spc$+"="+one.spc$
LET eq.tab%=(ww%(1)-WIDTH(eq$)-(fx1%(7)-fx%(7)))\2
LET fld.tab%=eq.tab%+WIDTH(eq$)
CALL MOVETO(eq.tab%,fy%(7)+12)
PRINT eq$;
EDIT FIELD 1,"",(fld.tab%,fy%(7))-(fld.tab%+(fx1%(7)-fx%(7)),fy1%(7))
```

The program prompts you to press ENTER when you have typed in the answer. Then the program prints eq$, which contains the left side of the equation. The right side is presented as an empty edit field for your answer.

Timer

The next lines start the timer and wait for you to enter an answer:

```
TIMER OFF
ON TIMER(tl) GOSUB time.out
LET time.left%=yes%
SOUND 880,.875
TIMER ON
WHILE DIALOG(0)<>6 AND time.left%
```

```
WEND
TIMER OFF
```

Tl is the specified time limit in seconds. Time.left% indicates whether the time limit has expired. The timer is turned on (TIMER ON) just before the program enters a dialog monitor loop.

The program exits from the monitor loop when you press ENTER or RETURN or when time.left%=0 (WHILE DIALOG(0)<>6 AND time.left%).

In the following block, the program checks your answer (the contents of the edit field):

```
IF time.left% THEN skip.buzzer
CALL MOVETO(fld.tab%+(fx1%(7)-fx%(7))+12,fy%(7)+12)
PRINT "Time out!"
SOUND 110,9: SOUND 0,18
skip.buzzer:
LET n%=1
GOSUB get.fvalue
LET g=x
LET att%=att%+1
IF ABS(g-r)<=tr THEN correct
SOUND 440,2
SOUND 110,2
LET response$="The answer is"+STR$(r)
GOTO continuation
correct:
LET succ%=succ%+1
FOR fq%=1 TO 4
SOUND fq%*110,1
NEXT fq%
IF r<>g THEN close.enough
LET response$="Correct"
GOTO continuation
close.enough:
LET response$="Close enough. Exact answer is"+STR$(r)
```

If time has run out, a buzzer sounds. However, the program still must check to see whether you had already entered the correct answer when time ran out.

The program checks to see whether your answer is sufficiently close to the correct answer (IF ABS(g−r)<=tr...). When the error tolerance is 0, ABS(g−r) must equal 0.

The program prepares one of three different responses for an answer that is outside the error tolerance, inside the error tolerance, or exactly right. The program updates the scorebox variables att% (attempts), succ% (successes), and pct% (percentage).

Responding to Your Answer

The next lines control the computer's response to your answer:

```
continuation:
LINE (0,wl%(1)*10/12+1)-(ww%(1),wl%(1)),0,bf
LET r.tab%=(ww%(1)-WIDTH(response$))\2
CALL MOVETO(r.tab%,fy%(7)+48)
PRINT response$;
LET pct%=INT(succ%/att%*100+.5)
WINDOW 2
GOSUB update.scorebox
WINDOW 1
FOR b%=6 TO 8
BUTTON b%-5,1,bl$(b%),(bx%(b%),by%(b%))-(bx1%(b%),by1%(b%)),bt%(b%)
NEXT b%
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
FOR btn%=1 TO 3
BUTTON CLOSE btn%
NEXT btn%
EDIT FIELD CLOSE 1
CLS
IF event%=6 THEN next.problem
LET b%=DIALOG(1)
ON b% GOTO end.quiz,next.problem,change.parameters
```

After printing the response and sounding a beep, the program waits for your next action command. Press ENTER, RETURN, or the NEXT button to get another equation. Press CHANGE to go back to the specification box. Press QUIT to end the program.

Here are the last two subroutines of the Speed Math program:

```
time.out:
TIMER OFF
LET time.left%=no%
RETURN
label.scorebox:
CLS
LET title$="SCOREBOX"
LET t.tab%=(ww%(2)-WIDTH(title$))/2
PRINT PTAB(t.tab%); title$
REM    123456789012345
LET h$=" Succ.  Att.   Pct."
LET f$="###   ###    ###"
LET h.tab%=(ww%(2)-WIDTH(h$))/2
PRINT PTAB(h.tab%); h$
update.scorebox:
PRINT PTAB(h.tab%) USING  f$;succ%,att%,pct%;
RETURN
```

The time.out subroutine is executed when time runs out while you are completing an equation. The subroutine signals that fact by setting time.left%=no% and then returns to the main program so the time out will be recognized.

The label.scorebox subroutine prints the row and column headings in the scorebox window. It also includes a subroutine to change only the scorebox values.

When entering the values for h$ and f$, use the REM 12345... statement as a guide for lining up the letters and symbols correctly.

—Testing the Program

Print out the entire program, and check it carefully. Run the program. Try to get results like those shown in Figures 12-3 and 12-4. Type in the correct answer and press ENTER before time runs out. Type in the correct answer, but don't press ENTER. The time out message should appear, but the program should still recognize your correct answer.

Set the error tolerance to 10 percent and type in an answer that isn't exactly right but is within 10 percent. You should get a message similar to that shown in Figure 12-4.

—Using Speed Math

Select an operation and operand ranges that challenge you. Give yourself a generous time limit. When you can answer correctly 90 percent of the time, reduce the time limit. Keep trying at the new time limit until you can reach 90 percent again. Repeat the process for various operations and operand ranges.

Now you're ready for life in the fast lane!

Chapter 13

Text Scanner

Text Scanner counts the number of words in a document (typed in from the keyboard or loaded from disk) and sorts them according to frequency of use. In other words, the program compiles a vocabulary and frequency list of all words used in a document.

Text Scanner can help you spot overused words in your own writing or redo comparative analyses of text from different authors. Using the results from the Text Scanner program, you can calculate the *variety index* of any document. (The variety index is the ratio of vocabulary size to total words used. For instance, if a 1000-word document uses 780 distinct words, it has a variety index of .78. A document in which no word is repeated has a variety index of 1.)

Teachers may find that the program is a convenient way of preparing a vocabulary list for use in conjunction with a reading assignment.

Researchers may find Text Scanner a useful tool for preparing keyword lists. Suppose a 2000-word technical article is to be catalogued according to its contents. Text Scanner will list every word used in the document in order of frequency. With this list, you can easily identify keywords to be used in the catalogue.

If you enjoy the challenge of a good maze, consider the task of designing one. That turns out to be every bit as difficult, and quite a bit more interesting. In this chapter, we'll explore the process of maze construction, and then program your Macintosh to produce an endless supply of mind-boggling mazes of varying complexity.

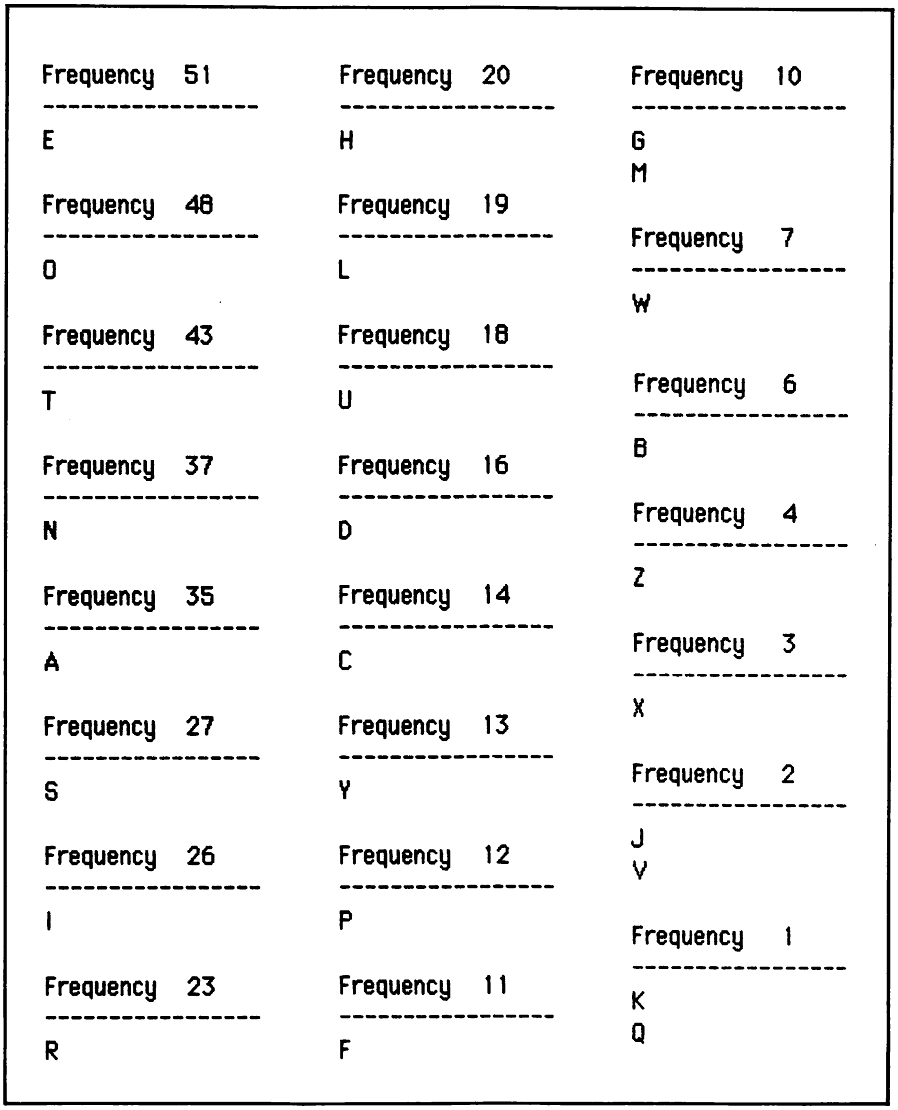
One way to start a maze is to picture the floor plan of a house with the walls in place but no doors. You then add doors until there's just one path between any two rooms in the house. Last of all, you add an entrance and an exit anywhere you choose.

Figure 13-1. The sample document used to test the Text Scanner

Finally, Text Scanner may be used in cryptanalysis (codebreaking). By setting the maximum word length to 1, you can find the frequency distribution of each letter in the alphabet. This is often the first step in breaking a cipher. (Cryptography is discussed further in Chapter 17, Secret Messages.)

```
Frequency   2             CONSTRUCTION        PLACE
----------------          DESIGNING           PROCESS
DOORS                     DIFFICULT           PRODUCE
HOUSE                     ENDLESS             PROGRAM
                          ENJOY               QUITE
Frequency   1             ENTRANCE            ROOMS
----------------          EVERY               START
ANYWHERE                  EXPLORE             SUPPLY
BETWEEN                   FLOORPLAN           THERE'S
CHALLENGE                 INTERESTING         TURNS
CHAPTER                   MACINTOSH           UNTIL
CHOOSE                    MAZES               VARYING
COMPLEXITY                MIND-BOGGLING       WALLS
CONSIDER                  PICTURE             WE'LL
```

Figure 13-2. Frequency analysis of the sample document, showing words with at least five letters



```
Frequency  51
----------------
E

Frequency  48
----------------
O

Frequency  43
----------------
T

Frequency  37
----------------
N

Frequency  35
----------------
A

Frequency  27
----------------
S

Frequency  26
----------------
I

Frequency  23
----------------
R

Frequency  20
----------------
H

Frequency  19
----------------
L

Frequency  18
----------------
U

Frequency  16
----------------
D

Frequency  14
----------------
C

Frequency  13
----------------
Y

Frequency  12
----------------
P

Frequency  11
----------------
F

Frequency  10
----------------
G
M

Frequency  7
----------------
W

Frequency  6
----------------
B

Frequency  4
----------------
Z

Frequency  3
----------------
X

Frequency  2
----------------
J
V

Frequency  1
----------------
K
Q
```

Figure 13-3. Frequency analysis of the sample document, showing the letter frequencies

Figure 13-1 shows a sample document used for testing the program. Figure 13-2 lists the results of a frequency analysis of words at least 5 characters long. Figure 13-3 lists the results of a frequency analysis of each letter in the document.

—How Text Scanner Works

The program has two major functional components: a word finder and a word filer.

The word finder reads text one line at a time from the keyboard or from a disk file, depending on which option is selected. The program examines the line one character at a time. Only a letter or numeral can mark the beginning of a word. If a character is not a letter, numeral, hyphen, or apostrophe, it is taken to be a delimiter. A delimiter marks the end of a word. Hyphens and apostrophes are not delimiters—thus, *didn't* and *red-hot* are treated as single words.

The word filer keeps track of the vocabulary in linked lists. A linked list is one in which each entry contains a pointer to the next entry in the list. Entries can be in any physical sequence at all—the pointers keep them sorted. In an ordinary list, entries must be in the exact physical order that is called for by the sorting scheme.

Figure 13-4 illustrates the linked list system used by Text Scanner. The figure shows the contents of frequency folders after analyzing the sentence, "A small step for man, a giant step for mankind." Looking at

Starting folder=#2

Folder #	**Frequency tag**	**Next-folder link**	**First-word link**
1	1	0	6
2	3	3	1
3	2	1	4

Word #	**Word**	**Next-word link**
1	A	0
2	SMALL	0
3	STEP	0
4	FOR	3
5	MAN	7
6	GIANT	5
7	MANKIND	2

Figure 13-4. The list structure after analysis of the sentence, "A small step for a man, a giant step for mankind."

folder #2, you see that it has a frequency tag of 3, indicating that it contains words that occurred three times in the text.

The first word link for folder #2 is 1. Looking at word number 1, you find the article "A." Thus "A" occurred three times in the text. The next word link for word #1 is 0, indicating there are no more words in that folder.

Going back to frequency folder #2, you see that the next folder link is 3. Folder 3 has a frequency tag of 2 (it contains words that occurred two times). The first word link for folder 3 is 4. Word number 4 is "FOR." Thus "FOR" occurred two times in the text.

The next word link for word #4 is 3. Word #3 is "STEP." Thus "STEP" occurred two times in the text.

You can continue tracing the word links and folder links until you reach the last word in the last folder (word link=0 and folder link=0).

Having a mental picture or model makes it easier to understand operations with linked lists. Figure 13-5 suggests a useful model for the lists used in this program.

Using this model, the process of filing a word can be described quite simply. Each time the program finds a word, it looks to see whether that word is already contained in one of the folders. If it is not, the program adds the word to the folder for words of frequency 1. If the word has already been filed, the program moves the word from its present folder into the folder for words with the next higher frequency.

Figures 13-6 through 13-18 show various phases in the program's operation. Scan through them now before continuing.

—Program Listing

The program starts out with descriptive data for the window, buttons, and edit fields. First, here are the window descriptors:

```
REM Window descriptor
REM  wide  long   left   top
DATA 6.500, 4.150, 0.250, 0.375
```

Here are the button descriptors:

```
REM Button descriptors
DATA 16
REM  label                        wide  long   hzone  vzone type
DATA Count words only,            1.875, 0.208, 0.063, 0.417, 2
```

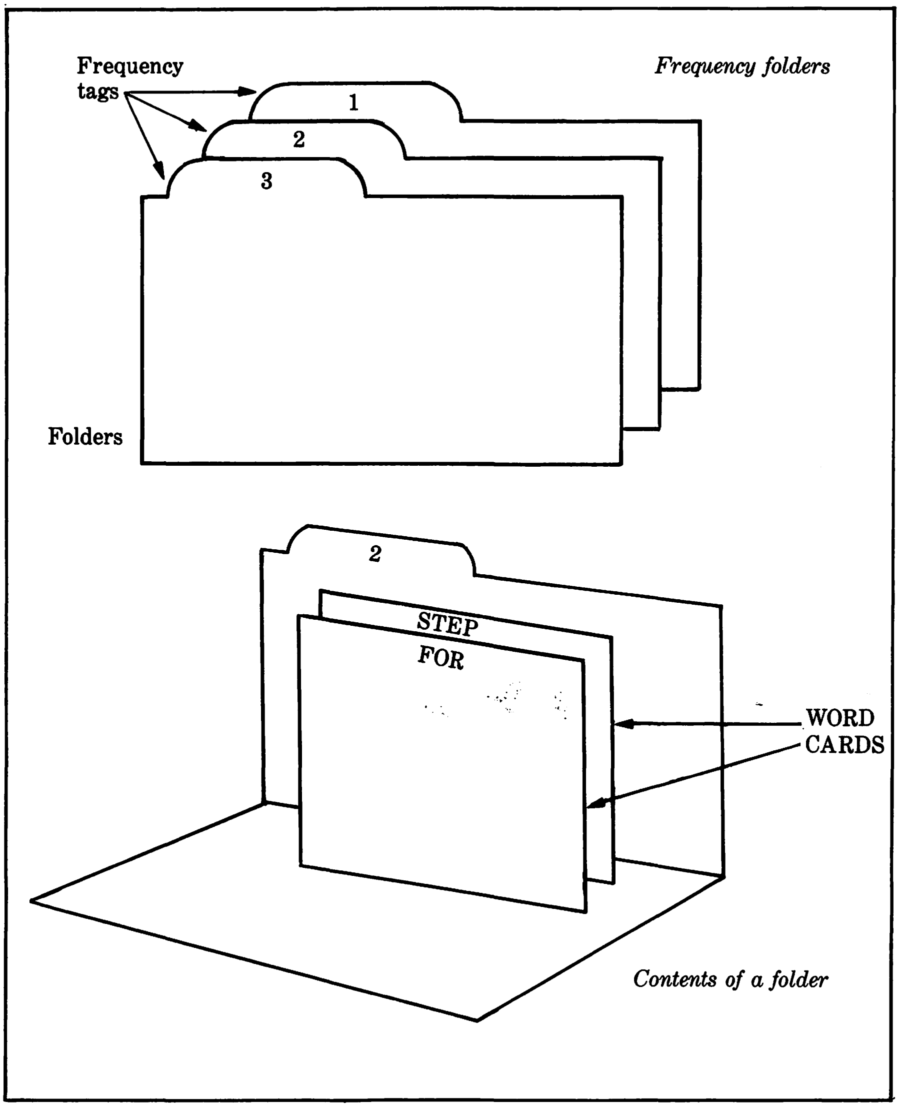
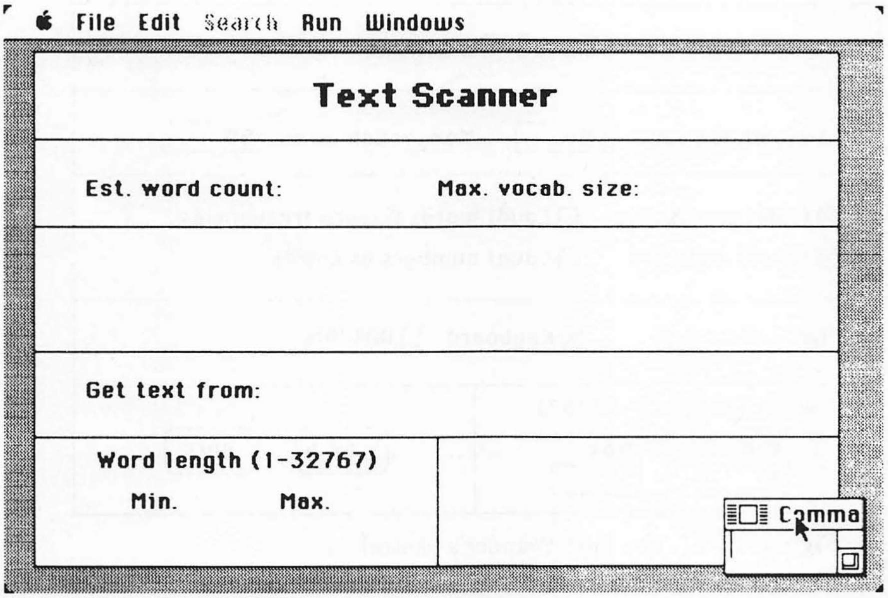


Figure 13-5. A model for the word filing system

```
DATA Count words & word frequencies, 3.313, 0.208, 0.750, 0.417, 2
DATA Ignore numbers,                 1.875, 0.208, 0.063, 0.500, 2
DATA Count numbers as words,         3.313, 0.208, 0.750, 0.500, 2
DATA Keyboard,                       1.125, 0.208, 0.438, 0.667, 2
DATA Disk file,                      1.000, 0.250, 0.667, 0.667, 2
DATA OK,                             0.750, 0.375, 0.688, 0.916, 1
```



Text Scanner

Est. word count: Max. vocab. size:

Get text from:

Word length (1-32767)
Min. Max.

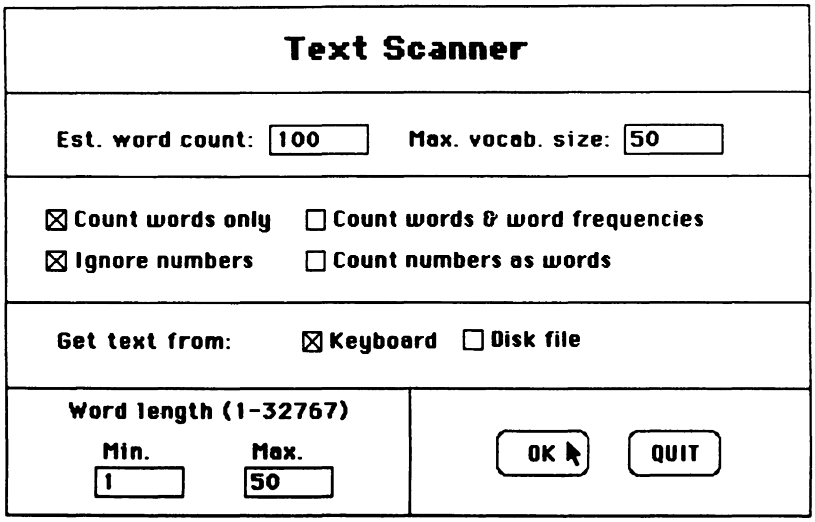
Figure 13-6. The Text Scanner's initial screen at test point 1

```
DATA UPDATE STATS,        1.750, 0.333, 0.333, 0.667, 1
DATA END SCAN,            1.000, 0.333, 0.667, 0.667, 1
DATA CONTINUE,            1.000, 0.333, 0.800, 0.916, 1
DATA SCREEN,              1.000, 0.333, 0.125, 0.666, 1
DATA PRINTER,             1.000, 0.333, 0.500, 0.666, 1
DATA DISK,                1.000, 0.333, 0.875, 0.666, 1
DATA FIND A WORD,         1.500, 0.333, 0.125, 0.916, 1
DATA SCAN ANOTHER,        1.500, 0.333, 0.542, 0.916, 1
DATA QUIT,                0.750, 0.375, 0.875, 0.916, 1
```

H.zone and v.zone indicate the relative horizontal and vertical position within the window. Other values are measured in inches.

Here are the field descriptors:

```
REM Field descriptors
DATA 4
REM  wide  long  hzone vzone
DATA 0.750, 0.208, 0.375, 0.250
DATA 0.750, 0.208, 0.875, 0.250
DATA 0.666, 0.208, 0.125, 0.958
DATA 0.666, 0.208, 0.333, 0.958
```


Text Scanner

Est. word count: 100 Max. vocab. size: 50

☒ Count words only ☐ Count words & word frequencies
☒ Ignore numbers ☐ Count numbers as words

Get text from: ☒ Keyboard ☐ Disk file

Word length (1-32767)
Min. 1 Max. 50

Figure 13-7. The Text Scanner's control box

The four fields being defined here are shown in Figure 13-7; they are estimated word count, maximum vocabulary size, minimum word length, and maximum word length.

The following block reads in the window data and converts the data from inches to screen points (pixels):

```
READ inches.wide,inches.long,ulcx,ulcy
LET ww%=inches.wide*72
LET wl%=inches.long*72
LET wx%=ulcx*72
LET wy%=ulcy*72
LET wx1%=wx%+ww%
LET wy1%=wy%+wl%
WINDOW 1,,(wx%,wy%)-(wx1%,wy1%),3
```

The next lines read in the button data:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
READ bl$(n%),inches.wide,inches.long,h.zone,v.zone,bt%(n%)
```

```
LET bx%(n%)=(ww%-inches.wide*72)*h.zone
LET by%(n%)=(wl%-inches.long*72)*v.zone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```

Finally, here are the lines that read in the edit field data:

```
READ nf%
DIM fx%(nf%),fy%(nf%),fx1%(nf%),fy1%(nf%)
FOR n%=1 TO nf%
READ inches.wide,inches.long,h.zone,v.zone
LET fx%(n%)=(ww%-inches.wide*72)*h.zone
LET fy%(n%)=(wl%-inches.long*72)*v.zone
LET fx1%(n%)=fx%(n%)+inches.wide*72
LET fy1%(n%)=fy%(n%)+inches.long*72
NEXT n%
```

Functions, Constants, and Parameters

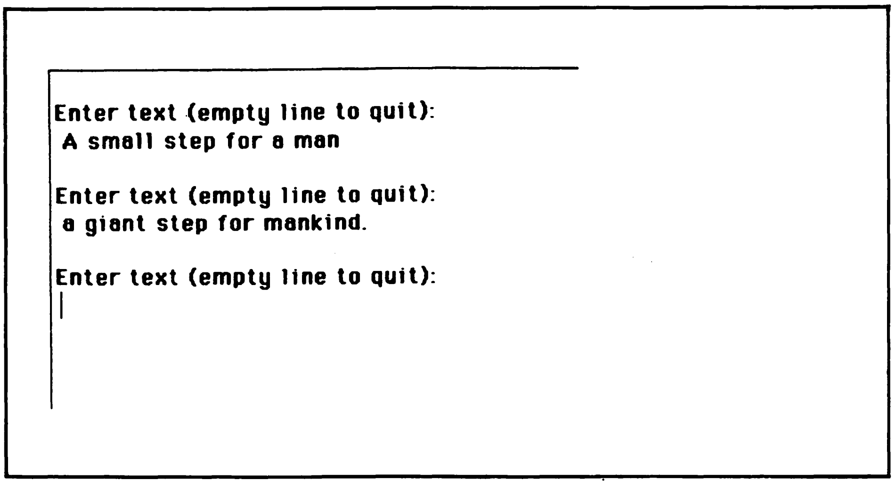
The next block defines four functions to eliminate repetitious code later in the program:

```
DEF FNposint(x)=(x=INT(x)) AND (x>0) AND (x<=32767)
DEF FNstrip$(x)=RIGHT$(STR$(x),LEN(STR$(x))-1)
DEF FNcapletter%(c$)=("A"<=c$) AND (c$<="Z")
DEF FNnumeral%(c$)=("0"<=c$) AND (c$<="9")
```

FNposint%(x) returns −1 if x is an integer between 1 and 32767. FNstrip$(x) returns a string representation of the number x without the usual leading space or minus sign. FNcapletter%(c$) returns −1 if c$ is a capital letter and 0 if it is not. FNnumeral%(c$) returns −1 if c$ is a numeral and 0 if it is not.

The following block sets up two constant arrays:

```
DIM device$(2),m$(4)
LET device$(1)="SCRN:"
LET device$(2)="LPT1:DIRECT"
FOR j%=1 TO 4
READ m$(j%)
NEXT j%
DATA Scanner stopped by user request.
```



```
Enter text (empty line to quit):
 A small step for a man

Enter text (empty line to quit):
 a giant step for mankind.

Enter text (empty line to quit):
```

Figure 13-8. The keyboard entry dialog

```
DATA Analysis completed (reached end of text).
DATA Analysis incomplete. Re-do with a larger est. word count.
DATA Analysis incomplete. Re-do with a larger max. vocab. size.
```

The device names in device$( ) specify which device is used for saving the results of a frequency analysis. For a higher-quality printout, you may set device$(2) equal to "LPT1:PROMPT" or "LPT1:".

M$( ) stores the status messages that are displayed upon completion of a text analysis.

Here are the other constant and parameter definitions:

```
LET max.w%=50   :REM maximum word length
LET min.w%=1   :REM minimum word length
LET max.l%=32767   :REM maximum line length
LET yes%=(1=1)
LET no%=(1=0)
LET cr$=CHR$(13)+CHR$(10)
LET nu$="": REM No spaces in quotes
LET hyphen$="-"   :REM type a single hyphen inside quotes
LET apostrophe$="'"   :REM type a single apostrophe inside quotes
LET cwf%=yes%   :REM count words and frequencies
LET ign%=yes%   :REM ignore numerals
LET src.is.kb%=yes%   :REM input text from keyboard
```

Scanner Control Box

The next section of the program lets you specify the type of operation to be executed.

First the program prints a title and divides the output window into six regions.

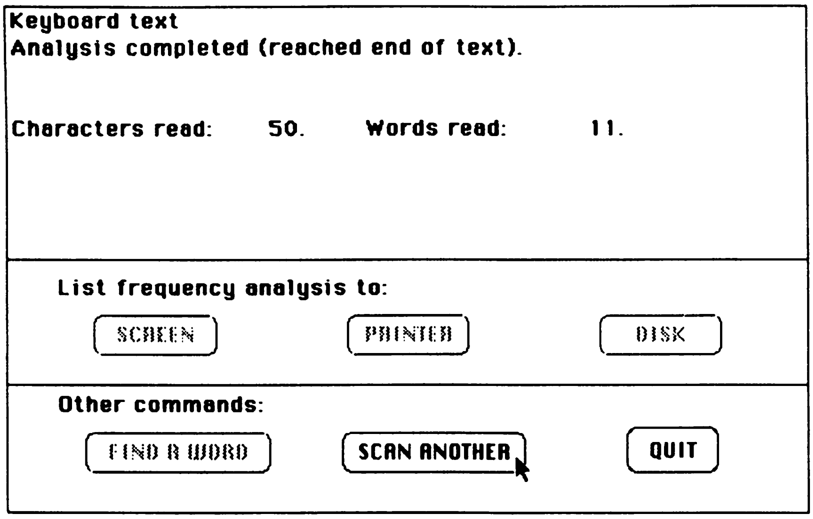
```
spec.dialogue:
CALL TEXTSIZE(18)
CALL TEXTFACE(1)
CLS
LET title$="Text Scanner"
LET title.htab%=(ww%-WIDTH(title$))/2
LET title.vtab%=(wl%-18)/9
CALL MOVETO(title.htab%,title.vtab%)
PRINT "Text Scanner"
CALL TEXTSIZE(12)
LINE (0,wl%/6)-STEP(ww%,0)
LINE (0,wl%/3)-STEP (ww%,0)
LINE (0,wl%*7/12)-STEP (ww%,0)
LINE (0,wl%*9/12)-STEP (ww%,0)
LINE (ww%/2,wl%*9/12)-STEP(0,wl%*3/12)
```

Next the program labels each of the six window regions:

```
CALL MOVETO(ww%/16,fy%(1)+12)
PRINT "Est. word count:"
CALL MOVETO(ww%/2,fy%(2)+12)
PRINT "Max. vocab. size:"
CALL MOVETO(ww%/16,by%(5)+12)
PRINT "Get text from:"
LET wl.title$="Word length (1-32767)"
LET wl.tab%=(ww%/2-WIDTH(wl.title$))/2
CALL MOVETO(wl.tab%,fy%(3)-30)
PRINT wl.title$
CALL MOVETO(fx%(3)+2,fy%(3)-6)
PRINT "Min."
CALL MOVETO(fx%(4)+2,fy%(4)-6)
PRINT "Max."
```

Test Point 1

This is a good time to check what you've typed. Print a listing of your work (in the command window, type **LLIST** and press ENTER). Check



```
Keyboard text
Analysis completed (reached end of text).

Characters read:     50.     Words read:        11.
```

Figure 13-9. Results of the text analysis (Count words only) using the text of Figure 18-8

the listing carefully against the program lines in this chapter. Then close the listing window, reduce the size of the command window, and run the program (COMMAND-R). You should see the screen shown in Figure 13-6.

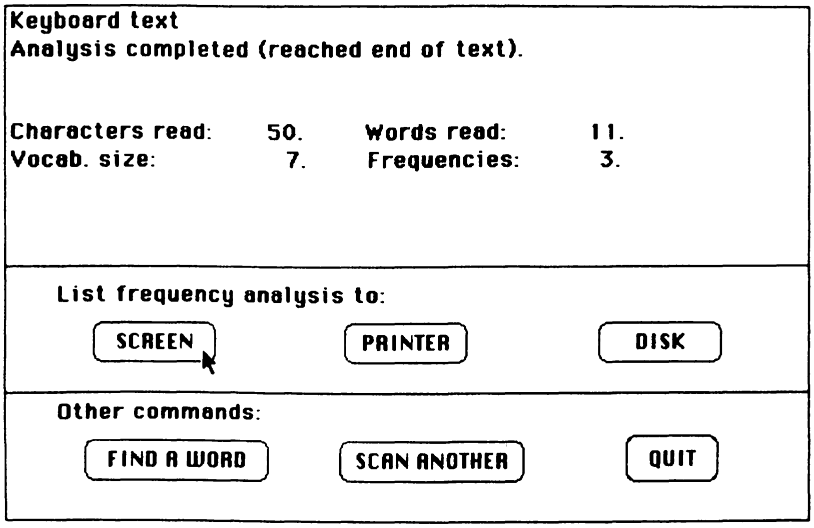
Adding Control Box Buttons And Edit Fields

Type in the following lines to add the edit fields shown in Figure 13-7:

```
EDIT FIELD 4, FNstrip$(max.w%),(fx%(4),fy%(4))-(fx1%(4),fy1%(4))
EDIT FIELD 3, FNstrip$(min.w%),(fx%(3),fy%(3))-(fx1%(3),fy1%(3))
EDIT FIELD 2,FNstrip$(mw%),(fx%(2),fy%(2))-(fx1%(2),fy1%(2))
EDIT FIELD 1,FNstrip$(ewc),(fx%(1),fy%(1))-(fx1%(1),fy1%(1))
```

The following lines create the buttons:

```
BUTTON 1,2+cwf%,bl$(1),(bx%(1),by%(1))-(bx1%(1),by1%(1)),bt%(1)
BUTTON 2,1-cwf%,bl$(2),(bx%(2),by%(2))-(bx1%(2),by1%(2)),bt%(2)
BUTTON 3,1-ign%,bl$(3),(bx%(3),by%(3))-(bx1%(3),by1%(3)),bt%(3)
BUTTON 4,2+ign%,bl$(4),(bx%(4),by%(4))-(bx1%(4),by1%(4)),bt%(4)
```


Keyboard text
Analysis completed (reached end of text).

```
Characters read:    50.    Words read:      11.
Vocab. size:         7.    Frequencies:      3.
```

Figure 13-10. Results of the text analysis (Count words and word frequencies) using the text of Figure 13-8

```
BUTTON 5,1-src.is.kb%,bl$(5),(bx%(5),by%(5))-(bx1%(5),by1%(5)),bt%(5)
BUTTON 6,2+src.is.kb%,bl$(6),(bx%(6),by%(6))-(bx1%(6),by1%(6)),bt%(6)
BUTTON 7,1,bl$(7),(bx%(7),by%(7))-(bx1%(7),by1%(7)),bt%(7)
BUTTON 8,1,bl$(16),(bx%(16),by%(16))-(bx1%(16),by1%(16)),bt%(16)
```

You can run the program again, but pressing the buttons or clicking the mouse will cause an error.

The following lines monitor any actions you make within the control box window:

```
LET fld%=1
spec.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=6 OR event%=7 THEN next.field
IF event%=2 THEN change.field
IF event%<>1 THEN spec.loop
LET btn%=DIALOG(1)
IF btn%=8 THEN quit.scanner
ON (btn%+1)\2 GOTO spec.cwf,spec.ign,spec.src,spec.ok
```

The following block executes your command (keyboard entry or mouse click):

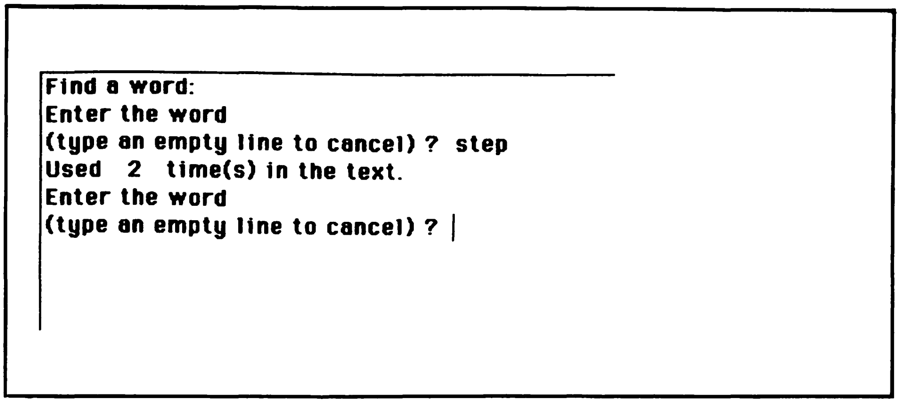
```
spec.cwf:
LET cwf%=(btn%=2)
BUTTON 1,2+cwf%
BUTTON 2,1-cwf%
GOTO spec.loop
spec.ign:
LET ign%=(btn%=3)
BUTTON 3,1-ign%
BUTTON 4,2+ign%
GOTO spec.loop
spec.src:
LET src.is.kb%=(btn%=5)
BUTTON 5, 1-src.is.kb%
BUTTON 6, 2+src.is.kb%
GOTO spec.loop
next.field:
LET fld%=fld% MOD 4 + 1
EDIT FIELD fld%
GOTO spec.loop
change.field:
LET new.fld%=DIALOG(2)
IF new.fld%=fld% THEN spec.loop
LET fld%=new.fld%
EDIT FIELD fld%
GOTO spec.loop
```

Spec.cwf changes the Count words and word frequencies indicator. Spec.ign changes the ignore-numbers indicator. Spec.src changes the source of input (keyboard or disk). Next.field and change.field respond to your commands to select another field.

Checking Specifications

When you press the OK button, the program checks all of the edit field entries:

```
spec.ok:
LET ewc=VAL(EDIT$(1))   :REM estimated word count
IF ewc<>INT(ewc) OR ewc<1 THEN EDIT FIELD 1: BEEP: GOTO spec.loop
LET x=VAL(EDIT$(2))
```



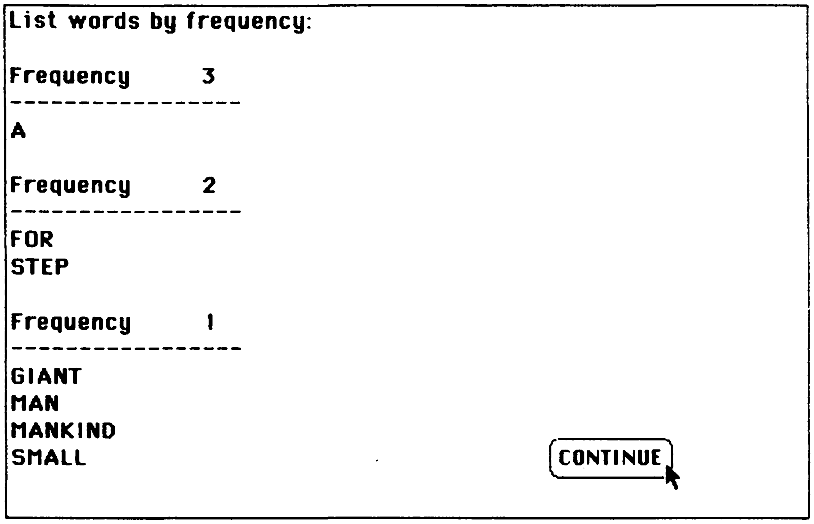
```
Find a word:
Enter the word
(type an empty line to cancel) ?  step
Used  2  time(s) in the text.
Enter the word
(type an empty line to cancel) ?
```

Figure 13-11. Find a word dialog

```
IF NOT FNposint(x) OR x>ewc  THEN EDIT FIELD 2: BEEP: GOTO spec.loop
LET mw%=x        :REM max vocabulary size
LET mf%=(SQR(8*ewc)-1)/2   :REM maximum distinct frequencies
IF mf%>mw% THEN LET mf%=mw%
LET x=VAL(EDIT$(3))
IF  NOT FNposint(x) THEN EDIT FIELD 3: BEEP: GOTO spec.loop
LET min.w%=x   :REM minimum word length
LET x=VAL(EDIT$(4))
IF NOT FNposint(x) OR x>32767 THEN EDIT FIELD 4: BEEP: GOTO spec.loop
LET max.w%=x   :REM maximum word length
IF min.w%>max.w% THEN EDIT FIELD 3: BEEP: GOTO spec.loop
FOR f%=1 TO 4
EDIT FIELD CLOSE f%
NEXT f%
FOR b%=1 TO 8
BUTTON CLOSE b%
NEXT b%
```

The estimated word count (ewc) should be a generous approximation of the total word length of the document to be analyzed. The program makes sure you enter a whole number for ewc; the number can be just about as large as you wish. However, don't make the number excessively large; that will slow down the frequency analysis and may cause the program to run out of memory.

The maximum vocabulary size (mw%) is the largest number of different words the program will be able to handle. In the model shown in



```
List words by frequency:

Frequency      3
----------------
A

Frequency      2
----------------
FOR
STEP

Frequency      1
----------------
GIANT
MAN
MANKIND
SMALL
```

Figure 13-12. Frequency analysis displayed on the screen

Figure 13-5, mw% is the number of word cards that can be stored in folders. Again, the number entered should be a generous approximation. Typically, the maximum vocabulary size should be about one half of the estimated word count.

The maximum number of frequency folders is derived from the estimated word count. If a document has ewc words, it can have at most

$$(-1 + \sqrt{8 \times \text{ewc}})/2$$

distinct word frequencies.

The program ensures that the minimum and maximum word length values min.w% and max.w% are logical and within the range of 1 to 32767. Setting both values equal to 1 causes the program to treat each letter or number as a word.

If all the entries are acceptable, the program closes the buttons and edit fields.

Test Point 2

To test the operation of the scanner control box, type in these temporary lines:

```
quit.scanner:
end
```

Run the program. You should see the control box shown in Figure 13-7. Make sure all the check-boxes work properly. Try switching between

- Counting words only and counting words and word frequencies
- Ignoring numbers and counting numbers as words
- Getting text from the keyboard or from a disk file.

Try various combinations of entries for word count, vocabulary size, and minimum/maximum word length. Press OK each time you want the program to check the settings. Whenever you have an invalid setting (for example, a word count=0), the program should beep and make the invalid error field active so you can fix it.

Before continuing, find and delete these lines:

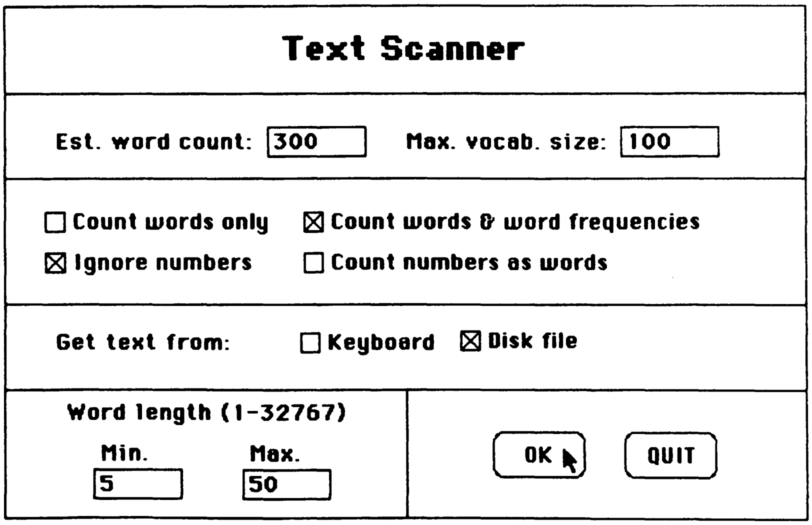
```
quit.scanner:
end
```

Setting Up the Document Scan

The next lines initialize the various counters used during the document analysis:

```
CLS
IF cwf% THEN GOSUB fq.setup
LET kf%=0
LET kw%=0
LET sf%=0
LET nc=0   :REM floating point so it can count more than 32767 characters
LET wc=0   :REM floating point so it can count more than 32767 words
IF NOT src.is.kb% THEN GOSUB disk.source ELSE CLS
LET cp%=0
LET li$=nu$
LET ef%=no%
LET rq.update%=no%
LET rq.end.scan%=no%
```

The variables kf% and kw% keep track of the number of frequency folders and words used. Sf% points to the first folder (Sf%=0 indicates


Text Scanner

Est. word count: 300 Max. vocab. size: 100

☐ Count words only ☒ Count words & word frequencies

☒ Ignore numbers ☐ Count numbers as words

Get text from: ☐ Keyboard ☒ Disk file

Word length (1-32767)

Min. 5 Max. 50

Figure 13-13. Control box set up for reading text from a disk file

no folders are in use). Nc is the total character count and wc is the total word count.

Text Analysis

The following block controls the text analysis:

```
DIALOG ON
LET et%=no%
LET ok%=yes%
at.loop:
IF rq.update% THEN GOSUB update: GOSUB set.interrupt.btns
IF rq.end.scan% THEN LET stop.code%=1: GOTO quit.analysis
GOSUB find.next.word
IF LEN(cw$)<min.w% THEN check.et
IF ew% AND cwf% THEN GOSUB word.filer
IF ok% AND ew% THEN LET wc=wc+1
check.et:
IF NOT et% AND ok% THEN at.loop
IF et% THEN LET stop.code%=2: REM due to end of text
quit.analysis:
```

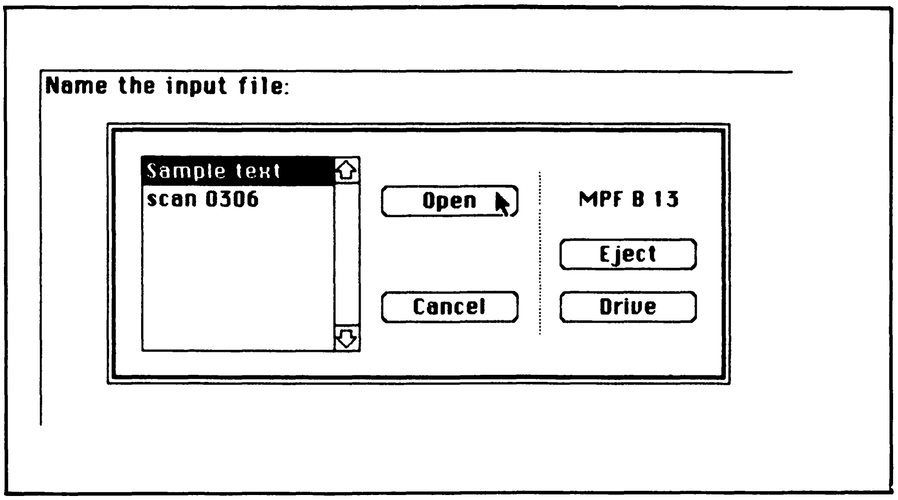


Figure 13-14. Input file specification dialog

```
CLOSE 1
BUTTON CLOSE 1
BUTTON CLOSE 2
DIALOG OFF
```

Et% is an indicator for the end of text. Ok% is a filing status indicator; ok%=no% means a word could not be filed for some reason. The repetitive procedure at.loop finds words and files them until one of several terminating conditions is true: you pressed the END SCAN button shown in Figure 13-15; the program reached the end of text (et%=yes%); or the program ran out of frequency folders or word cards (ok%=no%).

Scan Results Window

The following lines show the results of the analysis, as pictured in Figures 13-9 and 13-10:

```
fa.results:
CLS: ON ERROR GOTO 0
IF src.is.kb% THEN PRINT "Keyboard text" ELSE PRINT fi$
PRINT m$(stop.code%)
LINE (0,wl%*6/12)-STEP (ww%,0)
LINE (0,wl%*9/12)-STEP (ww%,0)
```

```
GOSUB show.stats
CALL MOVETO(ww%/16,by%(11)-12)
PRINT "List frequency analysis to:"
CALL MOVETO(ww%/16,by%(14)-12)
PRINT "Other commands:"
FOR b%=11 TO 14
BUTTON b%-10, -cwf%, bl$(b%), (bx%(b%), by%(b%))- (bx1%(b%), by1%(b%)),
      bt%(b%)
NEXT b%
FOR b%=15 TO 16
BUTTON b%-10,1,bl$(b%),(bx%(b%),by%(b%))-(bx1%(b%),by1%(b%)),bt%(b%)
NEXT b%
```

These lines wait for you to press one of the buttons:

```
fa.loop:
WHILE DIALOG(0)<>1
WEND
LET btn%=DIALOG(1)
FOR b%=11 TO 16
BUTTON CLOSE b%-10
NEXT b%
ON btn% GOTO list.freq, list.freq,list.freq,find.a.word,scan.another,
      quit.scanner
```

The following lines handle the SCAN ANOTHER and QUIT SCANNER buttons:

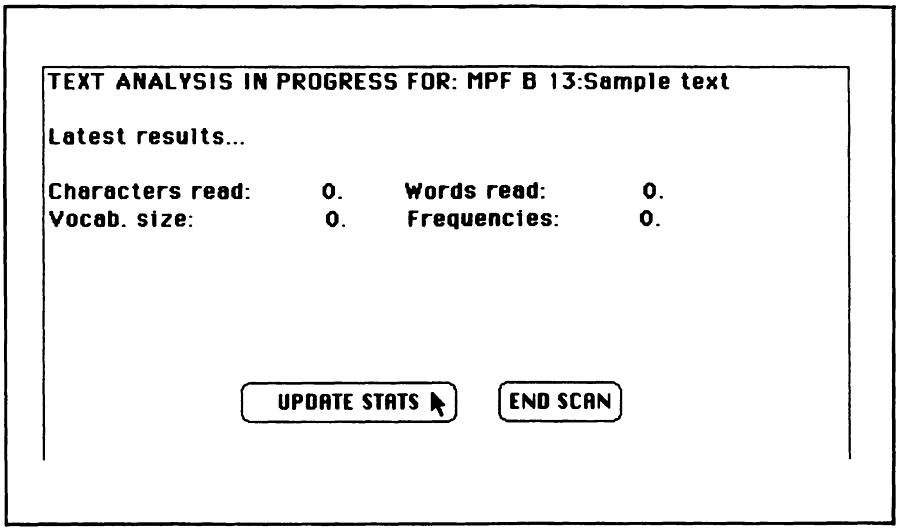
```
scan.another:
IF cwf% THEN ERASE f%, fl%, fw%, wd$, wdl%
GOTO spec.dialogue
quit.scanner:
WINDOW CLOSE 1
END
```

Auxiliary Subroutines

Before presenting the next major portion of the program, here are a number of subroutines that were called earlier.

The following subroutine lets you select a disk file for input:

```
disk.source:
CLS
```



```
TEXT ANALYSIS IN PROGRESS FOR: MPF B 13:Sample text

Latest results...

Characters read:      0.     Words read:        0.
Vocab. size:          0.     Frequencies:       0.
```

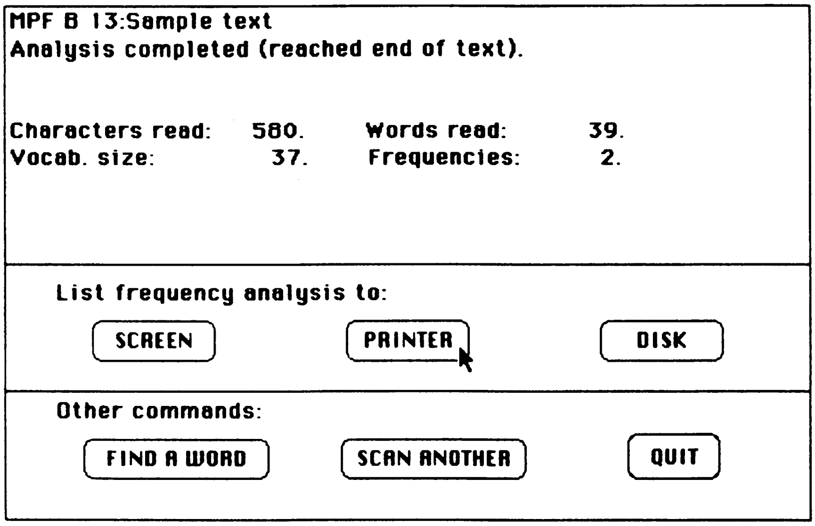
Figure 13-15. Screen appearance during analysis of a disk file document

```
PRINT "Name the input file: "
LET fi$=FILES$(1,"TEXT")
IF fi$=nu$ THEN scan.another
OPEN fi$ FOR INPUT AS 1
GOSUB update
GOSUB set.interrupt.btns
RETURN
```

During disk input, the program lets you request an update of the current document's statistics (as shown in Figure 13-15). The set.interrupt.btns subroutine activates the UPDATE STATS and END SCAN buttons. (During keyboard input, there is no need for periodic updates, so the buttons are not activated.)

Here is the set.interrupt.btns routine:

```
set.interrupt.btns:
FOR b%=8 TO 9
BUTTON b%-7,1,bl$(b%),(bx%(b%),by%(b%))-(bx1%(b%),by1%(b%)),bt%(b%)
NEXT b%
LET rq.update%=no%
LET rq.end.scan%=no%
ON DIALOG GOSUB set.rq.interrupt
RETURN
```



```
MPF B 13:Sample text
Analysis completed (reached end of text).

Characters read:   580.    Words read:      39.
Vocab. size:        37.    Frequencies:      2.
```

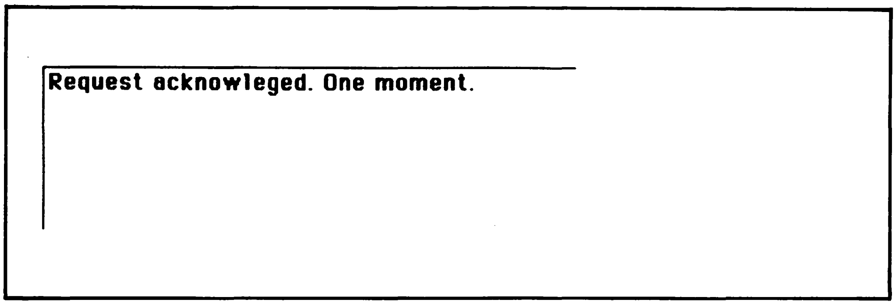
Figure 13-16. Final results of text analysis, using the document shown in Figure 13-1 and counting words with at least five letters

Pressing the UPDATE STATS or END SCAN button during the document analysis causes the program to execute the following subroutine:

```
set.rq.interrupt:
IF DIALOG(0)<>1 THEN RETURN
BUTTON CLOSE 1
BUTTON CLOSE 2
CLS
PRINT "Request acknowleged. One moment."
LET which.int%=DIALOG(1)
LET rq.update%=(which.int%=1)
LET rq.end.scan%=(which.int%=2)
RETURN
```

Sometimes there is a delay of several seconds before the program can update the statistics or end the scan, so the program immediately puts a message ("Request acknowledged...") on the screen, as shown in Figure 13-17.

Here are the lines that update the screen statistics:



```
Request acknowleged. One moment.
```

Figure 13-17. When you press the UPDATE STATS button, this message appears immediately

```
update:
CLS
PRINT "TEXT ANALYSIS IN PROGRESS FOR: "; fi$
PRINT cr$; "Latest results..."
GOSUB show.stats
RETURN
```

The next block puts a CONTINUE button on the screen and waits until you press it (the CONTINUE button is shown in Figure 13-12):

```
wait.ok:
BUTTON 1,1,bl$(10),(bx%(10),by%(10))-(bx1%(10),by1%(10)),bt%(10)
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
BUTTON CLOSE 1
RETURN
```

The following lines display the statistics on the screen:

```
show.stats:
LOCATE 5,1
REM                              1234567890123456789012345678
PRINT USING "Characters read: *****.     Words read:    *****.";nc,wc
IF NOT cwf% THEN RETURN
PRINT USING "Vocab. size:     *****.     Frequencies:  *****.";kw%,kf%
RETURN
```

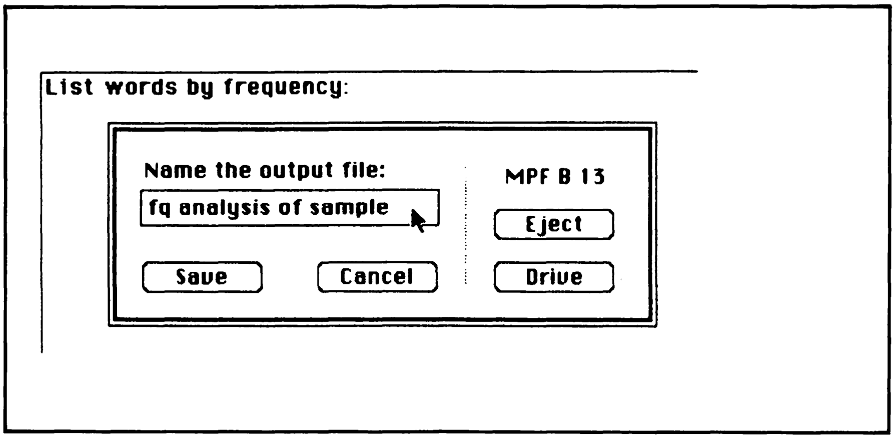


Figure 18-18. Output file specification dialog when listed to a disk file

Use the REM 1234567... statement as an aid in lining up the formats in the two succeeding PRINT USING statements.

Word-Counting Logic

Now we're ready to present the section that reads the text and finds each word. The first block controls the overall process:

```
find.next.word:
LET ew%=no%
LET iw%=no%
LET cw$=nu$
find.nw.loop:
GOSUB get.next.character
IF c$<>nu$ THEN not.null
LET et%=yes%
GOTO word.delimiter
not.null:
LET nc=nc+1
IF FNcapletter%(c$) THEN word.character
IF FNnumeral%(c$) AND NOT ign% THEN word.character
IF (c$=hyphen$ OR c$=apostrophe$) AND iw%=yes% THEN add.to.current.word
```

Ew%=yes% indicates that a word ending has been detected. Iw%= yes% indicates the program has recognized the beginning of a word.

First the program gets a character c$ from the text. If c$ equals nu$ (the null or empty string), the end of text has been reached. If it doesn't, the program determines whether the character is a capital letter or a numeral.

The next lines handle word delimiters:

```
word.delimiter:
IF iw%=yes% THEN LET ew%=yes%
GOTO processed.character
word.character:
LET iw%=yes%
add.to.current.word:
LET cw$=cw$+c$
IF LEN(cw$)=max.w% THEN LET ew%=yes%
processed.character:
IF ew%=no% AND et%=no% THEN find.nw.loop
RETURN
```

The following block controls the character input routine:

```
get.next.character:
IF cp%=LEN(li$) THEN line.processed
LET cp%=cp%+1
LET c$=MID$(li$,cp%,1)
RETURN
line.processed:
IF ef%=no% THEN no.eof
LET c$=nu$
RETURN
no.eof:
IF LEN(li$)=max.l% THEN max.line
LET c$=cr$
GOSUB get.a.line
LET li$=UCASE$(li$)
RETURN
max.line:
GOSUB get.a.line
LET li$=UCASE$(li$)
GOSUB get.next.character
RETURN
```

Characters are taken from the current line (li$) until they have all been read (cp%=LEN(li$)). Then the program must read another line, using the following subroutine:

```
get.a.line:
LET li$=nu$
LET cp%=0
IF src.is.kb% THEN from.kb ELSE from.disk
from.kb:
PRINT cr$; "Enter text (empty line to quit):"
LINE INPUT li$
IF li$=nu$ THEN LET ef%=yes%
RETURN
from.disk:
IF NOT EOF(1) THEN get.disk.line
CLOSE 1
LET ef%=yes%
RETURN
get.disk.line:
LINE INPUT #1, li$
RETURN
```

In the case of keyboard input, the program prints a prompt on the screen, as shown in Figure 13-8. In the case of disk input, the program simply reads another line from the disk file.

Test Point 3

To test your work so far, add these lines at the end of the current program listing:

```
fq.setup:
word.filer:
RETURN
list.freq:
find.a.word:
GOTO fa.results
```

After carefully checking a printout of the listing, run the program. In the scanner control box, select Count words only, because you haven't yet typed in the frequency counting logic.

Try to duplicate the results shown in Figures 13-7 and 13-8. If you have a text format disk file (save a MacWrite document in *text only* format), you can also select the Disk file box. But remember to select the Count words only box as well.

Frequency Analysis Logic

Before continuing, delete these lines from the program:

```
fq.setup:
word.filer:
RETURN
list.freq:
find.a.word:
GOTO fa.results
```

The following block of lines sets up the word filing system:

```
fq.setup:
PRINT "Setting up the filing system..."
DIM f%(mf%), fl%(mf%), fw%(mf%), wd$(mw%), wdl%(mw%)
FOR af%=1 TO mf%-1   :REM Set up available-folder linkage
LET fl%(af%)=af%+1
NEXT af%
LET fl%(mf%)=0
LET af%=1
FOR aw%=1 TO mw%-1   :REM Set up available-word-card linkage
LET wdl%(aw%)=aw%+1
NEXT aw%
LET wdl%(mw%)=0
LET aw%=1
RETURN
```

Now type in the following lines, which control the overall word-filing process.

```
word.filer:
GOSUB search.for.word
IF ok%=no% THEN new.word
GOSUB move.word          :REM from its current frequency folder
GOTO put.in.folder
new.word:
```

```
GOSUB add.a.word
IF ok%=no% THEN no.more.cards
put.in.folder:
GOSUB search.for.folder
IF ok%=yes% THEN folder.exists
GOSUB add.a.folder
IF ok%=no% THEN no.more.folders
folder.exists:
GOSUB insert.a.word
RETURN
no.more.folders:
LET stop.code%=3
RETURN
no.more.cards:
LET stop.code%=4
RETURN
```

On entry to the word filer, CW$ contains the latest word to be recognized by the word counting logic.

The first step in filing a word is to see if it has been previously filed. These lines locate the word:

```
search.for.word:
LET ok%=no%
LET cf%=sf%    :REM Start with first folder
LET pf%=0
look.inside.folder:
IF cf%=0 THEN RETURN
LET cw%=fw%(cf%)    :REM Start with first word in the folder
LET pw%=0
sfw.loop:
IF cw%=0 THEN next.folder   :REM no more cards in this folder
IF wd$(cw%)<>cw$ THEN next.word
LET ok%=yes%    :REM Found the word
RETURN
next.word:
IF wd$(cw%)>cw$ THEN next.folder   :REM not in this folder
LET pw%=cw%
LET cw%=wdl%(cw%)
GOTO sfw.loop
next.folder:
LET pf%=cf%
LET cf%=fl%(cf%)
GOTO look.inside.folder
```

The program looks through each folder until the word is found or the last folder is checked (in which case ok% is set to no%).

If a word has already been filed, it must be moved to the next-higher frequency folder. The following lines remove the word from its present folder:

```
move.word:
LET fq%=f%(cf%)+1
IF pw%<>0 THEN not.the.first.word
LET fw%(cf%)=wdl%(cw%)
GOTO fix.folder
not.the.first.word:
LET wdl%(pw%)=wdl%(cw%)
fix.folder:
IF fw%(cf%)<>0 THEN RETURN    :REM If folder isn't empty, return
IF cf%<>sf% THEN not.first.folder    :REM Is empty folder the 1st folder?
LET sf%=fl%(cf%)    :REM If yes, then reset the first-folder link
GOTO discard.folder
not.first.folder:
LET fl%(pf%)=fl%(cf%)
discard.folder:
LET fl%(cf%)=af%
LET af%=cf%    :REM Discarded folder is now at top of unused folder list
LET kf%=kf%-1
LET f%(cf%)=0
RETURN
```

In the case of a new word, the following lines put the word in the frequency 1 folder:

```
add.a.word:
IF aw%>0 THEN get.a.word.card
LET ok%=no%
RETURN
get.a.word.card:
LET ok%=yes%
LET sw%=aw%
LET aw%=wdl%(aw%)
LET kw%=kw%+1
LET wd$(sw%)=cw$
LET cw%=sw%
LET fq%=1
LET wdl%(cw%)=0
RETURN
```

When a word is being moved up to the next-higher frequency folder, the following lines locate the desired folder:

```
search.for.folder:
LET ok%=no%
LET cf%=sf%   :REM Start at first folder
LET pf%=0
sff.loop:
IF cf%=0 THEN RETURN
IF f%(cf%)<>fq% THEN folder.not.found.yet
LET ok%=yes%   :REM Found it
RETURN
folder.not.found.yet:
IF f%(cf%)<fq% THEN RETURN   :REM Should have found it by now, so quit
LET pf%=cf%
LET cf%=fl%(cf%)   :REM Check next folder
GOTO sff.loop
```

If the folder is not found, the program must create a folder. The next block performs that function:

```
add.a.folder:
IF af%>0 THEN get.an.unused.folder
LET ok%=no%
RETURN
get.an.unused.folder:
LET ok%=yes%
LET sq%=af%
LET af%=fl%(af%)
LET kf%=kf%+1
LET f%(sq%)=fq%
LET fl%(sq%)=cf%
LET fw%(sq%)=0
LET cf%=sq%
IF pf%>0 THEN not.the.first.folder
LET sf%=cf%
RETURN
not.the.first.folder:
LET fl%(pf%)=cf%
RETURN
```

Once a folder has been located for the word, the following lines insert the word among the other words in that folder:

```
insert.a.word:
LET sw%=cw%
LET cw%=fw%(cf%)
LET pw%=0
compare.with.current.word:
IF cw%=0 THEN insert.here
IF wd$(sw%)<wd$(cw%) THEN insert.here
LET pw%=cw%
LET cw%=wdl%(cw%)
GOTO compare.with.current.word
insert.here:
IF pw%>0 THEN fix.pw.pointer
LET fw%(cf%)=sw%
GOTO point.to.word
fix.pw.pointer:
LET wdl%(pw%)=sw%
point.to.word:
LET wdl%(sw%)=cw%
LET cw%=sw%
RETURN
```

That's the end of the filer logic. Now we continue with the logic to display the frequency analysis results.

List by Frequency Command

Pressing SCREEN, PRINTER, or DISK activates these lines:

```
list.freq:
CLS
PRINT "List words by frequency:"
IF btn%<3 THEN LET fo$=device$(btn%) ELSE LET fo$=FILES$(0,"Name the
      output file:")
IF fo$=nu$ THEN fa.results
ON ERROR GOTO file.err
OPEN fo$ FOR OUTPUT AS 1
IF sf%=0 THEN lbf.done
```

On entry to this routine, btn% equals 1, 2, or 3, depending on whether you selected the screen, printer, or disk as an output device for the list. In the case of disk output, the program prompts you to specify the output file name, as shown in Figure 13-18.

After setting up the specified output device, the following lines print the list:

```
LET cf%=sf%
LET line.number%=0
new.frequency:
PRINT#1,
PRINT#1, USING "Frequency #####"; f%(cf%)
PRINT#1,          "----------------"
LET line.number%=line.number%+3
IF btn%=1 AND line.number%>=16 THEN GOSUB wait.ok: LET line.number%=0
LET cw%=fw%(cf%)
print.next.word:
PRINT#1, wd$(cw%)
LET line.number%=line.number%+1
IF btn%=1 AND line.number%>=16 THEN GOSUB wait.ok: LET line.number%=0
LET cw%=wdl%(cw%)
IF cw%>0 THEN print.next.word
LET cf%=fl%(cf%)
IF cf%>0 THEN new.frequency
lbf.done:
PRINT#1,
CLOSE 1
PRINT "End of list."
GOSUB wait.ok
GOTO fa.results
file.err:
IF ERR<50 THEN ON ERROR GOTO 0
BEEP
PRINT"Output file error."
CLOSE 1
GOSUB wait.ok
RESUME fa.results
```

Finally, here's the block that handles the FIND A WORD button:

```
find.a.word:
CLS
PRINT "Find a word:"
fw.loop:
LET cw$=nu$
PRINT "Enter the word"
INPUT "(type an empty line to cancel) ";cw$
```

```
LET cw$=UCASE$(cw$)
IF cw$=nu$ THEN fa.results
GOSUB search.for.word
IF ok%=no% THEN word.not.found
PRINT "Used ";f%(cf%);" time(s) in the text."
GOTO fw.loop
word.not.found:
PRINT "Not used in the text."
GOTO fw.loop
```

This block generates a dialog like that shown in Figure 13-11. The program takes the word you enter, converts it to uppercase, and searches through all the frequency folders for a matching word.

—Using the Program

Now you can test the frequency analysis portion of the program.

In the scanner control box, select Count words and word frequencies and Keyboard. Now try to duplicate the keyboard entries and analysis shown in Figures 13-8, 13-10, 13-11, and 13-12.

Next save the program on disk and use a word processing program like MacWrite to make a sample text file. You may want to enter the exact text shown in Figure 13-1 so you can check the program's operation against what is shown in this chapter. Save the document in text format (in MacWrite, use the Save as... command and specify Text only).

Now run the Text Scanner program and try to duplicate the dialogs and results shown in Figure 13-2 and Figures 13-13 through 13-17.

Next set the minimum and maximum word length in the scanner control box to 1 and read the sample text file again. Your results should match those shown in Figure 13-3 (if you copied the sample text from Figure 13-1).

When running the program, you'll notice it takes quite a while to analyze a document. Most of the delay is in filing the words. As the vocabulary grows, the filing delay increases. If it suits your purposes, you can minimize the delay by setting a minimum word length of 3, 4, 5, or more.

Chapter 14

Roman Numerals

Your Macintosh is a whiz at many forms of arithmetic: binary, decimal, hexadecimal, integer, floating point, and so forth. But try using a simple Roman numeral like MIX where the Macintosh expects a number and the computer will accuse you of making a syntax error.

In this chapter we present a program that fills in the void in the Macintosh's education. The program enables the computer to convert those confusing strings of Roman letters into familiar Arabic numbers, and vice versa.

Apart from the pleasure of seeing your computer grow smarter, you might even find a few practical uses for the program. After all, Roman numerals are everywhere—on public monuments and buildings, in movie credits, copyright notices, book prefaces, and outlines. Furthermore, you can sharpen your understanding of the common Arabic or decimal system by considering the Roman way.

Before presenting the program, we'll review the Roman numeral system.

—Refresher Course in Roman Numerals

The Roman system uses seven letters: M, D, C, L, X, V, and I, representing the quantities 1000, 500, 100, 50, 10, 5, and 1, respectively. Any positive whole number may be represented using these seven letters.

In the old Roman system, numbers are always written left to right in order of magnitude, largest first. The resultant number is the sum of its constituent letters, as in MDDCCI=1000+500+100+100+1=1701.

The modern Roman system (used in this chapter) allows certain two-letter combinations in which the magnitude of the smaller number of the reversed pair is subtracted from the larger number. For example, CIX=100+(10−1)=109.

Compared with the old Roman system, the modern Roman system allows more compact representations of numbers that contain 4s and 9s, as shown in the following examples:

Decimal	*Old Roman*	*Modern Roman*
4	IIII	IV
49	XXXXVIIII	XLIX
1492	MCCCCLXXXXII	MCDXCII
1984	MDCCCCLXXXIIII	MCMLXXXIV

Only six reverse-magnitude pairs are allowed: IV, IX, XL, XC, CD, and CM. In general, the second member of a pair must be only 5 or 10 times greater than the first. Thus, IM is not a valid representation of 999 because M is 1000 times greater than I. CMXCIX is the correct Roman numeral for 999.

In the modern Roman system, a letter may be used at most three times in succession. For instance, CCCC is not a valid representation of 400; CD must be used instead. The only exception is that the letter M may be used in unbroken succession any number of times.

There are also two restrictions on the use of the six reversed pairs:

- Reversed pairs may only be used when the numeral preceding the pair is greater than the second member of the pair. As a result, sequences like DCD and CCD are not allowed.
- A numeral following a reversed pair must be smaller than the first member of the reversed pair. By this rule, CMD is invalid.

—Converting Roman to Arabic

Since the context of a letter is often critical in determining its value, we start with a table showing the value that is given to the second letter in the sequence for all possible two-letter combinations (see Table 14-1).

Table 14-1. Values for Roman Numeral Sequences

	M	D	C	L	X	V	I
M	1000	500	100	50	10	5	1
D	0	0	100	50	10	5	1
C	800	300	100	50	10	5	1
L	0	0	0	0	10	5	1
X	0	0	80	30	10	5	1
V	0	0	0	0	0	0	1
I	0	0	0	0	8	3	1

Note: To find the value of Roman letter b when preceded by letter a, read the number at the intersection of row a and column b.

The table shows the result of going from one symbol to another during the evaluation of a numeral. To find the value of letter b when preceded by letter a, look at the intersection of row a, column b. If the intersection contains a positive number, add that number to the running total. Illegal sequences are indicated by 0's in the table.

Notice that for a V after an I, you add 3, not 5. But previously you saw IV=5−1. The contradiction is only apparent: the net effect of the sequence is the same, because for the letter I, you always add 1 according to the table. In other words, IV=1+3=4.

Let's work through a longer conversion using the numeral MDIV.

When evaluating the first letter, there is no previous letter to consider, so you take the letter's value, in this case 1000, as the initial running total.

Now, to process the second letter in the numeral, D, find the intersection of row M (previous letter) and column D (current letter) in Table 14-1. The result is 500, indicating that you must add 500 to the running total (1000+500=1500).

To process the third letter, I, get the contents of row D, column I, which is 1, and add it to the running total (1500+1=1501). To process the final letter (V), get the contents of row 1, column V, which is 3, and add it to the running total (1501+3=1504, the decimal equivalent of MDIV).

—Converting Arabic to Roman

Converting from Arabic to Roman is simpler. We start with the understanding that any decimal number *d* can be expressed as a sum of Roman numeral elements:

$$d = a*M + b*CM + c*D + d*CD + e*C + f*XC + g*L + h*XL + i*X + j*IX + k*V + l*IV + m*I$$

The letters a through m represent non-negative numbers. Thus, converting from Arabic to Roman is simply a matter of factoring out each of these values (the Arabic equivalents of M, CM, and so forth) from the Arabic number, and replacing each occurrence of a factor with the corresponding Roman symbol.

To factor a particular term, subtract it from the decimal number *d* repeatedly until the result is smaller than the term you are factoring. As an illustration, we'll convert 2411 into a Roman numeral:

Factoring steps	*Cumulative Roman numeral*
2411 − M = 1411	M
1411 − M = 411	MM
411 − CD = 11	MMCD
11 − X = 1	MMCDX
1 − I = 0	MMCDXI

The final value in the right-hand column tells us that 2411 decimal equals MMCDXI Roman.

—The Program

The first block contains descriptors for the window, buttons, and edit fields used in the program.

```
REM Window descriptor
REM  wide,  long,  left,   top
DATA 4.000, 3.000, 1.500, 0.750
REM Button descriptors
DATA 5
REM label,              wide    long    hzone vzone type
```

```
DATA Roman to Arabic, 1.750, 0.208, 0.500, 0.375, 2
DATA Arabic to Roman, 1.750, 0.208, 0.500, 0.500, 2
DATA OK,              0.750,  0.333, 0.500, 0.875, 1
DATA QUIT,            0.750,  0.333, 0.875, 0.875, 1
DATA MENU,            0.750,  0.333, 0.125, 0.875, 1
REM Field descriptor
REM  wide   long   hzone   vzone
DATA 2.000, 0.208, 0.500,  0.375
```

The next block reads in the window data:

```
READ inches.wide, inches.long, ulcx, ulcy
LET ww%=inches.wide*72
LET wl%=inches.long*72
LET wx%=ulcx*72
LET wy%=ulcy*72
LET wx1%=wx%+ww%
LET wy1%=wy%+wl%
```

The program uses only one window.
Here is the corresponding logic for reading the button data:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
READ bl$(n%),inches.wide, inches.long, hzone, vzone,bt%(n%)
LET bx%(n%)=(ww%-inches.wide*72)*hzone
LET by%(n%)=(wl%-inches.long*72)*vzone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```

And here is the routine to read the edit field data:

```
READ inches.wide, inches.long, hzone,vzone
LET fx%=(ww%-inches.wide*72)*hzone
LET fy%=(wl%-inches.long*72)*vzone
LET fx1%=fx%+inches.wide*72
LET fy1%=fy%+inches.long*72
```

Look through the sample screens shown in Figures 14-1 to 14-7 to locate the various buttons and edit fields used in the program.

Program Constants

The next block defines the constants that determine the text's appearance within the output window:

```
WINDOW 1,,(wx%,wy%)-(wx1%,wy1%),3
CALL TEXTFONT(2)  :REM New York
CALL TEXTFACE(1)  :REM bold
CALL TEXTMODE(1)  :REM overprint
LET t.main$="Roman Numerals"
LET t.main.tab%=(ww%-WIDTH(t.main$))/2
LET t.rtoa$="Convert Roman to Arabic"
LET t.rtoa.tab%=(ww%-WIDTH(t.rtoa$))/2
LET t.ator$="Convert Arabic to Roman"
LET t.ator.tab%=(ww%-WIDTH(t.ator$))/2
DEF FNstrip$(n)=RIGHT$(STR$(n),LEN(STR$(n))-1)
LET conv%=1   :REM default selection is Roman TO Arabic
LET yes%=(1=1)
LET no%=(1=0)
```

Loading the Roman Numeral Sequence Table

The following lines load in a copy of the sequence table (Table 14-1).

```
DIM t%(7,7),l%(7),f%(13),fc$(13)
FOR tr%=1 TO 7
FOR tc%=1 TO 7
READ t%(tr%,tc%)
NEXT tc%,tr%
DATA 1000, 500, 100, 50, 10, 5, 1
DATA 0,    0,   100, 50, 10, 5, 1
DATA 800,  300, 100, 50, 10, 5, 1
DATA 0,    0,   0,   0,  10, 5, 1
DATA 0,    0,   80,  30, 10, 5, 1
DATA 0,    0,   0,   0,  0,  0, 1
DATA 0,    0,   0,   0,  8,  3, 1
FOR tc%=1 TO 7
READ l%(tc%)
NEXT tc%
DATA 4,3,3,2,2,1,1
FOR n%=1 TO 13
READ f%(n%)
NEXT n%
```

```
FOR n%=1 TO 13
READ fc$(n%)
NEXT n%
DATA 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1
DATA M,    CM, D,  CD,  C,   XC, L,  XL, X, IX, V, IV, I
LET c$="MDCLXVI"
```

T%(,) is the sequence table. L%( ) stores the order of magnitude of each Roman numeral: 4 indicates thousands (M), 3 indicates hundreds (D and C), 2 indicates tens (L and X), and 1 indicates units (V and I). F%( ) stores the 13 factors used in Roman-to-Arabic conversions, and FC$( ) stores the Roman numerals corresponding to each factor.

The variable c$ stores the seven symbols of the Roman system in descending order of magnitude.

The Menu

The next block presents the menu shown in Figure 14-1:

```
main.menu:
CLS
PRINT
```

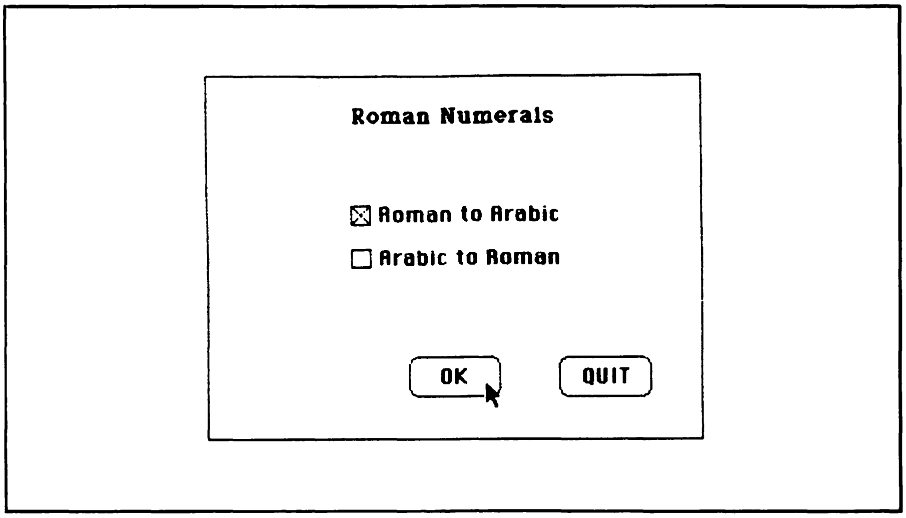


Figure 14-1. Main menu of the Roman-to-Arabic program

```
PRINT PTAB(t.main.tab%); t.main$
FOR n%=1 TO 2
BUTTON n%,1-(conv%=n%), bl$(n%), (bx%(n%), by%(n%))- (bx1%(n%),
      by1%(n%)), bt%(n%)
NEXT n%
FOR n%=3 TO 4
BUTTON n%,1,bl$(n%),(bx%(n%),by%(n%))-(bx1%(n%),by1%(n%)),bt%(n%)
NEXT n%
mm.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=6 THEN option.selected   :REM Pressed Enter or Return
IF event%<>1 THEN mm.loop   :REM Ignore all others except buttons
LET sel%=DIALOG(1)   :REM Which button was pressed?
ON sel% GOTO change.conv,change.conv,option.selected,quit
```

The program repeats the mm.loop procedure until an option is selected. At that point the program jumps to one of the routines contained in the following block:

```
change.conv:
LET conv%=sel%   :REM Note which was selected
BUTTON 1,3-conv%   :REM Update button status
BUTTON 2,conv%
GOTO mm.loop
option.selected:
FOR n%=1 TO 4
BUTTON CLOSE n%
NEXT n%
CLS
ON conv% GOSUB roman.to.arabic, arabic.to.roman
GOTO main.menu
quit:
WINDOW CLOSE 1
END
```

The change.conv routine switches the current conversion type from Roman to Arabic or vice versa. The option.selected routine activates the conversion procedure indicated in the check-boxes. The quit routine ends the program.

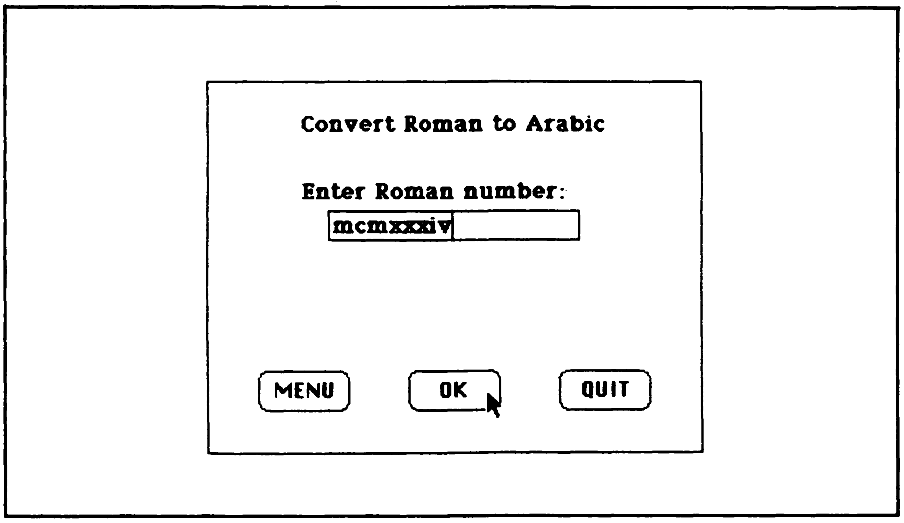


Figure 14-2. Entering a Roman numeral for conversion

Roman-to-Arabic Program Logic

The following lines start the Roman-to-Arabic conversion by presenting the screen shown in Figure 14-2:

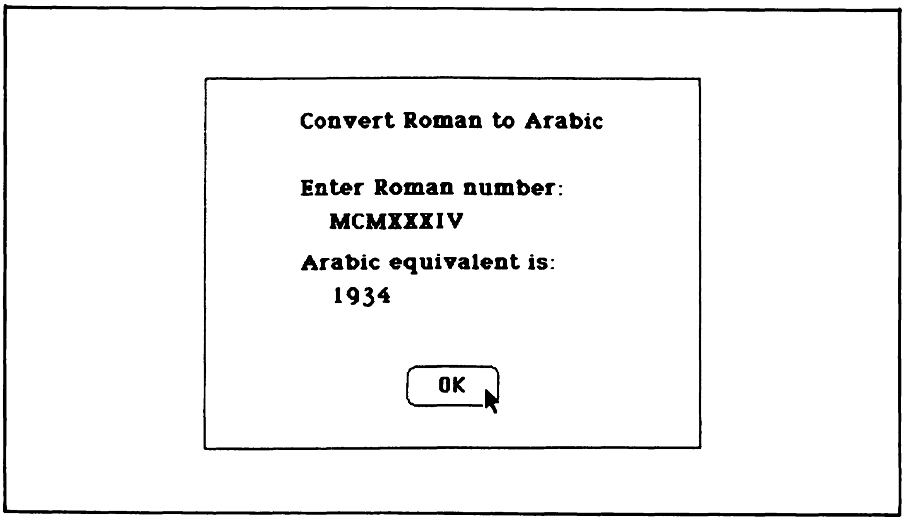
```
roman.to.arabic:
PRINT
PRINT PTAB(t.rtoa.tab%); t.rtoa$
LET n$=nu$
CALL MOVETO(t.rtoa.tab%,fy%-8)
PRINT "Enter Roman number:"
rtoa.loop:
GOSUB field.dialogue
IF btn%=2 THEN quit
IF n$=nu$ OR btn%=3 THEN RETURN
LET n$=UCASE$(n$)
CALL MOVETO(fx%,fy%+12)
PRINT n$
LET tl=0
LET f%=no%
LET pl%=4
LET pc%=1
LET oc%=1
LET d%=1
LET rc%=0
```

The field.dialogue subroutine presents the edit field shown in Figure 14-2 and waits for you to enter a value. The program then re-displays that value, converted to uppercase and stored in n$ as shown in Figure 14-3, and starts the conversion process.

Tl% is the running total of the number being converted. F% is a status variable indicating whether the letter read previously was part of a reverse pair.

Cc% and Cl% (not introduced in the program yet) store the corresponding information about the current character. Pl% and pc% store the order of magnitude and the column number of the previous character (M=column 1, D=column 2, and so forth). Oc% is the column number of the character preceding the previous one. Initially, there are no previous characters corresponding to cc% and oc%, but the structure of the program requires values for them anyway. In effect, the program acts as if the previous two characters are both M.

The variable d% points to the position of the current character


Convert Roman to Arabic

Enter Roman number:
MCMXXXIV

Arabic equivalent is:
1934

Figure 14-3. Result of a Roman-to-Arabic conversion

within n$. Rc% counts the number of times the current character has been used in succession. If rc% exceeds 3, the Roman numeral is invalid, unless the character is M.

Now that the counters are set up, the program extracts the character at position d%:

```
next.character:
LET f$=MID$(n$,d%,1)
LET cc%=INSTR(1,c$,f$)
IF cc%=0 THEN invalid.character
LET cl%=l%(cc%)
IF cc%<>pc% THEN LET rc%=1 ELSE LET rc%=rc%+1
IF rc%>3 AND cc%<>1 THEN too.many
IF f%=yes% AND cl%>=pl% THEN invalid.sequence
LET v%=t%(pc%,cc%)
IF v%=0 THEN invalid.sequence
LET tl=tl+v%
IF cc%>=pc% THEN not.a.reverse.pair
IF l%(oc%)<=pl% THEN invalid.sequence
```

The first three lines ensure that the character is one of the seven Roman letters. If it is, cl% stores its order of magnitude. The program checks whether the character is repeating (IF cc%< >pc%...) and whether it forms a valid sequence with the preceding character (LET v%=... and IF v%=0 THEN invalid.sequence).

If the sequence is valid, the program determines whether it is a reversed pair (IF cc%>=pc% THEN not.a.reverse.pair).

The next block of lines handles valid sequences:

```
LET f%=yes%
LET cl%=l%(pc%)
GOTO character.ok
not.a.reverse.pair:
LET f%=no%
character.ok:
LET pl%=cl%
LET oc%=pc%
LET pc%=cc%
LET d%=d%+1
IF d%<=LEN(n$) THEN next.character
CALL MOVETO(t.rtoa.tab%,fy%+36)
PRINT "Arabic equivalent is:"
```

```
CALL MOVETO(fx%,fy%+56)
PRINT FNstrip$(tl)
GOTO pause.rtoa
```

The first three lines identify reversed pairs. The next two lines identify non-reversed pairs. The character.ok routine advances to the next character in the numeral until none is left, at which time the program prints the accumulated decimal value tl.

Here are the lines that handle invalid sequences and characters:

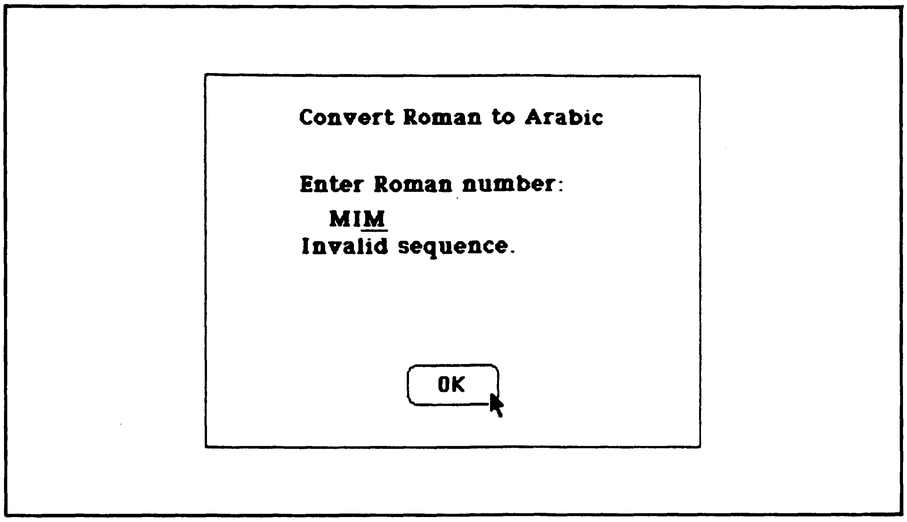
```
invalid.character:
GOSUB show.err
CALL MOVETO(t.rtoa.tab%,fy%+36)
PRINT "Invalid character."
PRINT PTAB(t.rtoa.tab%); "Use only {M,D,C,L,X,V,I}"
GOTO pause.rtoa
too.many:
GOSUB show.err
CALL MOVETO(t.rtoa.tab%,fy%+36)
PRINT "Too many ";f$; "'s in a row."
PRINT PTAB(t.rtoa.tab%); "Limit is 3."
GOTO pause.rtoa
invalid.sequence:
GOSUB show.err
PRINT PTAB(t.rtoa.tab%); "Invalid sequence."
pause.rtoa:
GOSUB pause
LINE (0,fy%)-(ww%,wl%),0,bf
GOTO rtoa.loop
```

Three error routines are included: invalid.character, too.many, and invalid.sequence. All use the show.err subroutine to place an underscore beneath the character that triggered the error condition, as shown in Figures 14-4 and 14-5.

Arabic-to-Roman Program Logic

The following block presents the screen shown in Figure 14-6:

```
arabic.to.roman:
PRINT
PRINT PTAB(t.ator.tab%); t.ator$
```

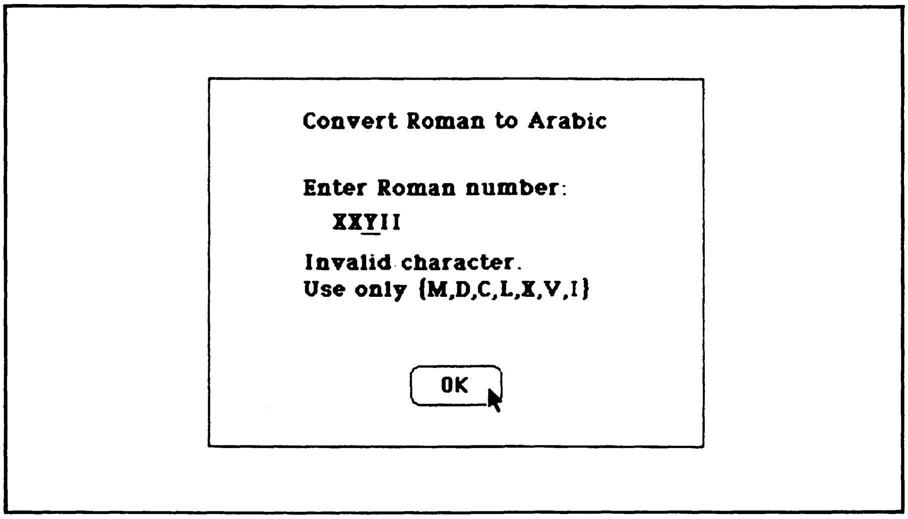

Convert Roman to Arabic

Enter Roman number:

MIM

Invalid sequence.

Figure 14-4. Roman-to-Arabic error notice identifying an out-of-place character

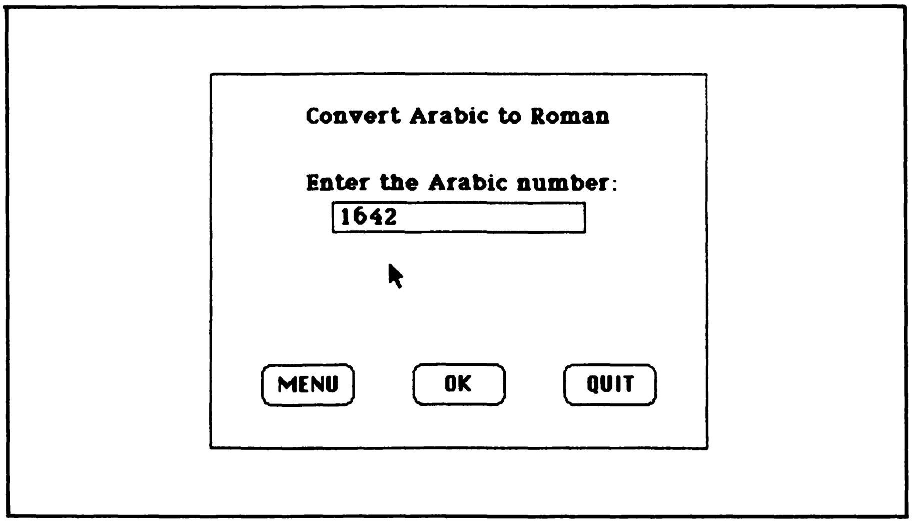

Convert Roman to Arabic

Enter Roman number:

XXYII

Invalid character.
Use only {M,D,C,L,X,V,I}

Figure 14-5. Roman-to-Arabic error notice identifying an invalid character


Convert Arabic to Roman

Enter the Arabic number:

1642

Figure 14-6. Entering an Arabic numeral for conversion

```
LET n$=nu$
CALL MOVETO(t.rtoa.tab%,fy%-8)
PRINT "Enter the Arabic number:"
ator.loop:
GOSUB field.dialogue
IF btn%=2 THEN quit
IF n$=nu$ OR btn%=3 THEN RETURN
CALL MOVETO(fx%,fy%+12)
PRINT n$
LET n=VAL(n$)
IF n<=0 OR n<>INT(n) THEN invalid.arabic
LET r$=nu$
LET fl%=1
```

The field.dialogue subroutine waits for you to enter a value into the edit field. Next the program ensures that you have entered a positive whole number (IF n<=0 or n< >INT(n)...).

R$ builds the string of Roman numerals; each time a factor is subtracted from the number, the corresponding symbol is concatenated to R$. Fl% is an index pointing to the current factor. For instance, when fl%=1, f%(fl%)=1000, and fc$(fl%)="M".

Here's the program logic that factors the number n:

```
subtract.current.factor:
LET nt=n-f%(fl%)
IF nt<0  THEN next.factor
LET r$=r$+fc$(fl%)
LET n=nt
GOTO subtract.current.factor
next.factor:
LET fl%=fl%+1
IF fl%<=13 THEN subtract.current.factor
CALL MOVETO(t.rtoa.tab%,fy%+36)
PRINT "Roman equivalent is:"
CALL MOVETO(fx%,fy%+56)
PRINT r$
GOSUB pause
LINE (0,fy%)-(ww%,wl%),0,bf
GOTO ator.loop
```

This routine subtracts the current factor f%(fl%) from the Roman numeral n. If the result nt is less than 0, the current factor is too large, so the index is advanced to the next factor. Otherwise, the Roman symbol corresponding to the factor f%(fl%) is concatenated to r$, the old value of n is replaced with the value of nt, and the program attempts to subtract the factor again.

This process continues until all 13 factors have been tried. At this point, r$ contains the final value in Roman form. The program displays the result, as shown in Figure 14-7.

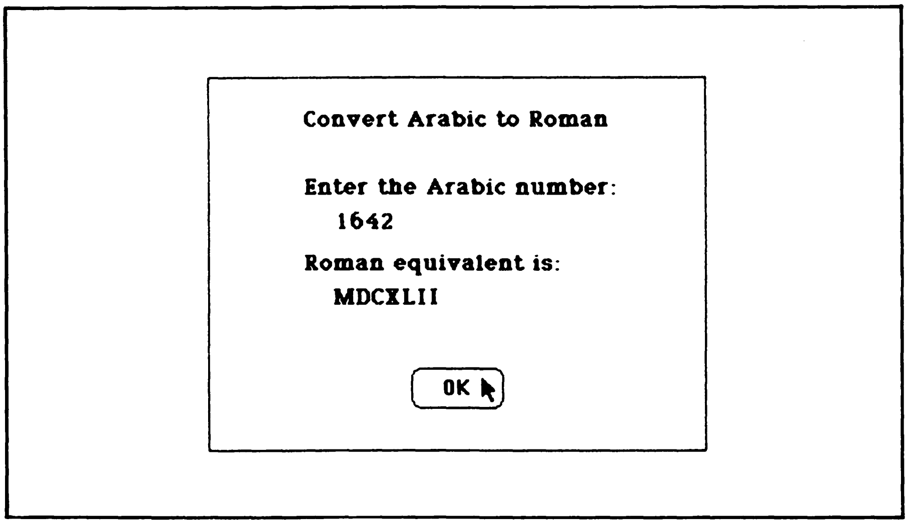
The following routine handles invalid Arabic numbers:

```
invalid.arabic:
CALL MOVETO(t.rtoa.tab%,fy%+36)
PRINT "Not a positive whole number."
GOSUB pause
LINE (0,fy%)-(ww%,wl%),0,bf
GOTO ator.loop
```

Auxiliary Subroutines

The following subroutine creates an edit field and waits for you to enter a value (the edit field is seen in Figures 14-2 and 14-6):

```
field.dialogue:
LET n$=nu$
```


Convert Arabic to Roman

Enter the Arabic number:
1642

Roman equivalent is:
MDCXLII

Figure 14-7. Result of an Arabic-to-Roman conversion

```
LET btn%=0
EDIT FIELD 1,n$,(fx%,fy%)-(fx1%,fy1%)
FOR b%=1 TO 3
LET bn%=b%+2
BUTTON b%,1,bl$(bn%), (bx%(bn%), by%(bn%))- (bx1%(bn%), by1%(bn%)),
     bt%(bn%)
NEXT b%
LET event%=0
WHILE event%<>6 AND event%<>1
LET event%=DIALOG(0)
WEND
IF event%=6 THEN LET btn%=1 ELSE LET btn%=DIALOG(1)
LET n$=EDIT$(1)
EDIT FIELD CLOSE 1
FOR b%=1 TO 3
BUTTON CLOSE b%
NEXT b%
RETURN
```

The following pause subroutine creates a button on the screen and waits for you to press it or type RETURN or ENTER:

```
pause:
BUTTON 1,1,bl$(3),(bx%(3),by%(3))-(bx1%(3),by1%(3)),bt%(3)
```

```
WHILE DIALOG(0)=0
WEND
BUTTON CLOSE 1
RETURN
```

Finally, here is a subroutine that puts an underscore beneath the character at position d% in the string n$. The subroutine is used to highlight invalid entries for Roman numerals.

```
show.err:
LET p1%=fx%+WIDTH(LEFT$(n$,d%-1))-2
LET p2%=WIDTH(MID$(n$,d%,1))+2
LINE (p1%,fy%+14)-STEP(p2%,0)
RETURN
```

—Testing and Using the Program

After comparing a printout of what you've typed with the program listing as it appears in this chapter, run the program. You should be able to duplicate the results shown in Figures 14-1 to 14-7. The program will convert any positive whole number into Roman numeral form, and any Roman numeral into Arabic form. However, large Arabic numbers requiring a long succession of M's cannot fit within the output window. To convert such numbers, enter only the portion of the number smaller than 1000, and then prefix the appropriate number of M's onto the result given by the program. For instance, to convert 32767, enter the value 767 and prefix 32 M's to the Roman equivalent the program gives for 767.

Chapter 15

Poetry Generator

The Macintosh can't really write poetry any more than it can paint a picture or conceive an idea. You can, however, use the Poetry Generator program to combine randomly selected words and fit them into a grammatical skeleton. The result will occasionally pass for a real poem but more often will stand as a silly but entertaining bit of doggerel.

The vocabulary and poem structure that you provide make a big difference in the quality of the final results. While an arbitrary list of words may produce interesting and surprising results, a carefully chosen set of words tends to give the poems more coherence.

Ideally, the vocabulary should be chosen to suit the intended grammatical structures as well—that way you can minimize cases of incorrect subject/verb agreement or incorrect verb forms.

To illustrate the possibilities, the favorite words and verse formats of three poets were entered into the Poetry Generator. The results are shown in Figure 15-1. The poems were edited to eliminate obvious grammatical errors; otherwise they are straight from the Mac.

```
William Shakespeare
Shall I compare thee to a minion's bosom?
Thou are more tyrannous and more twain
Saucy senses do assail the obsequious lips of sense,
and nymph's music hath all too tender a muse.
Shall I compare thee to a tomb's duty?
Thou art more seemly and more marigold.
Sovereign loves do assail the tender minions of love,
And syllable's actor hath all too decrepit a sphere.

Emily Dickinson
The bird covets her own victory;
Then guesses the company;
In her silent truth buzz no more.
The definition presumes her own thing;
Then covets the victory;
Of her condensed journey buzz no more.
The thing presumes her own civility;
Then advocates the nectar;
With her forbidden victory perish no more.

Robert Frost
The guests are arched, yellow, and reluctant,
But I have seeds to wake,
And grounds to find before I dwell,
And orchards to stop before I hear.
The birches are snowy, long, and lone,
But I have stones to wake,
And steeples to see before I look,
And birches to prefer before I taste.
```

Figure 15-1. Poems produced by the Poetry Generator, using the words and verse formats of William Shakespeare, Emily Dickinson, and Robert Frost

—How the Poetry Generator Works

Two data structures determine the type of poems produced: the vocabulary, which is entered from the keyboard or loaded from a disk file, and the poem format, which is entered from the keyboard.

Along with each word in the vocabulary, you must indicate the part of speech, using the eight abbreviations: \nn (noun), \aj (adjective), \av (adverb), \pr (preposition), \vt (verb transitive), \vi (verb intransitive),

\sc (subordinate conjunction), \cc (coordinate conjunction). Given this information, the program determines which words satisfy the specifications given in the poem format you design.

—Designing a Poem Format

First write down a one- to six-line poem model. Then replace each part of speech (except for articles and other words that you want to appear verbatim) with the corresponding part of speech. As an example, suppose you take this verse as a model:

> The dewdrop hangs from a twig
> in late winter—
> a window into spring.

The grammatical structure for that verse is:

> The *noun verb-intransitive preposition* a *noun*
> *preposition adjective noun*—
> a *noun preposition noun.*

Notice we've replaced most words with their grammatical descriptions but have left the articles and punctuation intact. Now replace the part of speech with the appropriate abbreviation:

```
The \nn \vi \pr a \nn
\pr \aj \nn—
a \nn \pr \nn.
```

Now you have a generalized, abbreviated poem format that the computer can understand.

As the computer reads the poem format, it replaces each code with a randomly chosen word taken from the appropriate grammatical category.

The Poetry Generator does not check for subject and verb agreement, the proper spelling of articles ("a" before consonants, "an" before vowels), the proper spelling of plural nouns, and so forth. If your vocabulary list includes verbs in the third person singular and your format includes a plural subject, you will end up with incorrect results like

> The glum bull and the blue moon
> stalks the rebellious highway.

Don't hesitate to edit your Macintosh's poems for grammatical correctness. After all, even real poets occasionally need a little help.

—The Program

The first three program blocks contain descriptive data for the windows, buttons, and edit fields, in the same sequence used in most of the previous chapters. Here is the data for the windows:

```
REM Window descriptors
DATA 2
DATA 3.25, 4.125, 0.125, 0.375
DATA 3.25, 4.125, 3.625, 0.375
```

The left margin, top margin, width, and length are given in inches. The first window is used for all of the dialogs; the second window is used for displaying the poems.

The next lines describe the program's 14 dialog buttons:

```
REM Button descriptors
DATA 14
DATA Set poem format, 2.000, 0.208, 0.125, 0.250, 3
DATA Load words, 2.000, 0.208, 0.125, 0.333, 3
DATA Type in new words, 2.000, 0.208, 0.125, 0.417, 3
DATA Save words, 2.000, 0.208, 0.125, 0.500, 3
DATA Edit/review words, 2.000, 0.208, 0.125, 0.583, 3
DATA Make poems, 2.000, 0.208, 0.125, 0.666, 3
DATA QUIT, 2.000, 0.208, 0.125, 0.833, 3
DATA MENU, 1.000, 0.333, 0.500, 0.958, 1
DATA Screen, 1.000, 0.208, 0.500, 0.417, 2
DATA Printer, 1.000, 0.208, 0.500, 0.500, 2
DATA Disk file, 1.000, 0.208, 0.500, 0.583, 2
DATA POEM, 1.000, 0.333, 0.500, 0.813, 1
DATA BACK, 1.000, 0.333, 0.250, 0.750, 1
DATA FWD, 1.000, 0.333, 0.750, 0.750, 1
```

Here are the edit field definitions:

```
REM Field descriptors
DATA 10
DATA 0.500, 0.208, 0.875, 0.125
DATA 2.750, 0.208, 0.667, 0.208
DATA 2.750, 0.208, 0.667, 0.292
DATA 2.750, 0.208, 0.667, 0.375
DATA 2.750, 0.208, 0.667, 0.458
DATA 2.750, 0.208, 0.667, 0.542
DATA 2.750, 0.208, 0.667, 0.625
DATA 1.000, 0.208, 0.667, 0.333
DATA 1.000, 0.208, 0.500, 0.250
DATA 2.000, 0.208, 0.500, 0.333
```

To find each of the buttons and edit fields described in these lines, scan through the sample screens shown in Figures 15-2 through 15-11.

The next three blocks read in the data, converting inches to display units (pixels) where necessary. First the windows:

```
READ nw%
DIM ww%(nw%),wl%(nw%),wx%(nw%),wy%(nw%),wx1%(nw%),wy1%(nw%)
FOR n%=1 TO nw%
READ inches.wide, inches.long, ulcx, ulcy
LET ww%(n%)=inches.wide*72
LET wl%(n%)=inches.long*72
LET wx%(n%)=ulcx*72
LET wy%(n%)=ulcy*72
LET wx1%(n%)=wx%(n%)+ww%(n%)
LET wy1%(n%)=wy%(n%)+wl%(n%)
NEXT n%
```

Now the buttons:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
READ bl$(n%),inches.wide, inches.long, hzone, vzone, bt%(n%)
LET bx%(n%)=(ww%(1)-inches.wide*72)*hzone
LET by%(n%)=(wl%(1)-inches.long*72)*vzone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```

And finally the edit fields:

```
READ nf%
DIM fx%(nf%),fy%(nf%),fx1%(nf%),fy1%(nf%)
FOR n%=1 TO nf%
```

```
READ inches.wide, inches.long, hzone, vzone
LET fx%(n%)=(ww%(1)-inches.wide*72)*hzone
LET fy%(n%)=(wl%(1)-inches.long*72)*vzone
LET fx1%(n%)=fx%(n%)+inches.wide*72
LET fy1%(n%)=fy%(n%)+inches.long*72
NEXT n%
```

Program Constants and Word List Arrays

This next block sets up constant values and arrays for storing information about the vocabulary:

```
DIM w.type$(8),kn%(8),pn%(8),device$(3)
FOR t%=1 TO 8
READ w.type$(t%)
NEXT t%
DATA nn, aj, av, pr, vt, vi, sc, cc
LET b.wid%=.5*72   :REM width of each word-type button
LET b.len%=.208*72   :REM length of each word-type button
LET valid.codes$="\NN\AJ\AV\PR\VT\VI\SC\CC"   :REM no spaces inside
     quotes
LET nu$=""   :REM no spaces inside quotes
LET device$(1)="SCRN:"
LET device$(2)="LPT1:DIRECT"
LET device$(3)="DISK"
LET mark$="\"                                         :REM no spaces
LET yes%=(1=1)
LET no%=(1=0)
LET nw%=0   :REM number of words
LET nl%=0   :REM number of lines per poem
LET wd.limit%=500
LET max.poems%=100
DIM wl$(0),wt%(0),nn%(0),aj%(0),pr%(0),av%(0),vt%(0),vi%(0),sc%(0),cc%(0)
```

W.type$( ) stores the abbreviated codes for each word type. Kn%( ) keeps a count of the number of words in each grammatical category. Pn%( ) is used while sorting the words into categories. Device$( ) contains the names of the three output devices (SCRN: for the screen, LPT 1: for the printer, and DISK for disk files).

Wl$( ) and wt%( ) store the individual words and word type (1=noun, 2=adjective, and so forth). Nn%( ) stores pointers to each noun, aj%( ) stores pointers to each adjective, and so forth. For instance, if the

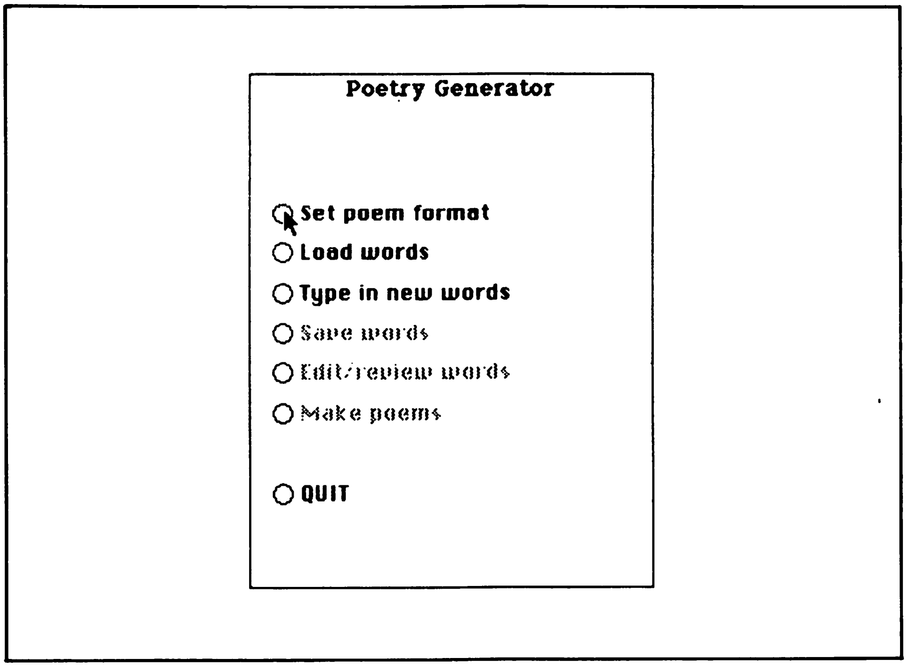


Figure 15-2. Main menu of the Poetry Generator

second noun is stored in w1$(3), then wt%(3)=1, indicating a noun, and nn%(2)=3.

The Main Menu

The following lines create the dialog box shown in Figure 15-2:

```
WINDOW 1,,(wx%(1),wy%(1))-(wx1%(1),wy1%(1)),3
CALL TEXTMODE(0)   :REM overprint text
CALL TEXTFONT(2)   :REM New York
main.menu:
CALL TEXTFACE(1)   :REM bold
CALL TEXTSIZE(12)
LET title.main$="Poetry Generator"
LET title.main.tab%=(ww%(1)-WIDTH(title.main$))/2
CLS
PRINT PTAB(title.main.tab%); title.main$
CALL TEXTFACE(0)
CALL TEXTSIZE(12)
BUTTON 1,1,bl$(1),(bx%(1),by%(1))-(bx1%(1),by1%(1)),bt%(1)
BUTTON 2,1,bl$(2),(bx%(2),by%(2))-(bx1%(2),by1%(2)),bt%(2)
BUTTON 3,1,bl$(3),(bx%(3),by%(3))-(bx1%(3),by1%(3)),bt%(3)
BUTTON 4,-(nw%>0),bl$(4),(bx%(4),by%(4))-(bx1%(4),by1%(4)),bt%(4)
BUTTON 5,-(nw%>0),bl$(5),(bx%(5),by%(5))-(bx1%(5),by1%(5)),bt%(5)
BUTTON 6, -(nw%>0 AND nl%>0), bl$(6), (bx%(6), by%(6))- (bx1%(6),
    by1%(6)), bt%(6)
BUTTON 7, 1,bl$(7),(bx%(7),by%(7))-(bx1%(7),by1%(7)),bt%(7)
```

Notice from the figure that three of the commands are disabled: Save words, Edit/Review words, and Make poems. These commands are not enabled until you have typed in or loaded a word list and provided a poem format.

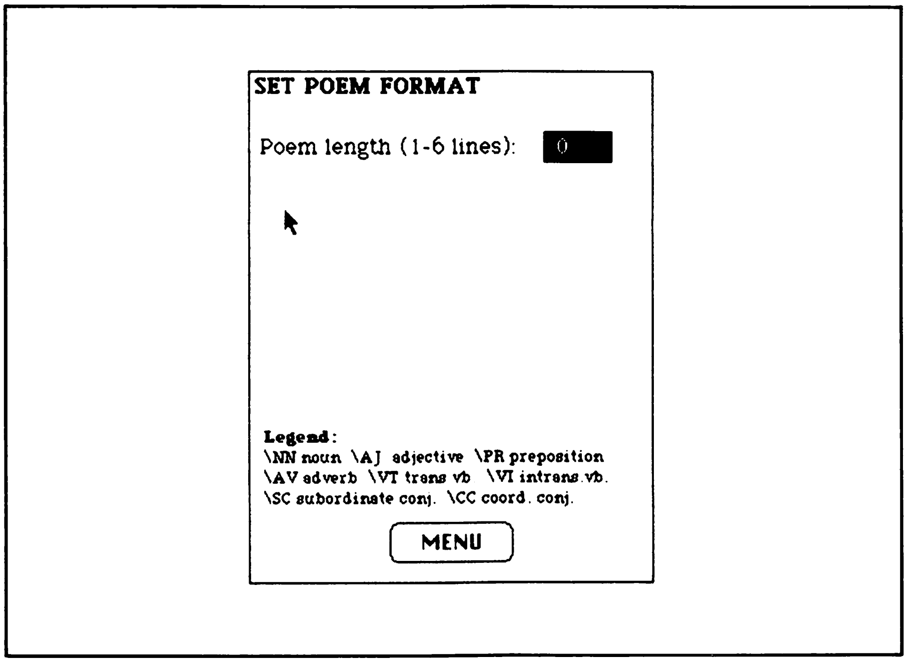
The next block waits for you to press one of the dialog buttons and responds accordingly:

```
LET event%=0
WHILE event%<>1
LET event%=DIALOG(0)
WEND
LET btn%=DIALOG(1)
FOR b%=1 TO 7
BUTTON CLOSE b%
NEXT b%
IF btn%=7 THEN quit
ON btn% GOSUB set.fmt, load.wds, type.in.wds, save.wds, edit.wds,
    make.poems
GOTO main.menu
quit:
WINDOW CLOSE 1
WINDOW CLOSE 2
END
```

Each of the major commands is handled by a subroutine. Upon returning from a subroutine, the program starts over and displays the main menu.

Test Point 1

To test your work thus far, try to run the program. You should be able to get the screen shown in Figure 15-2. However, pressing any of the buttons will cause an "undefined label" error at this point.


SET POEM FORMAT

Poem length (1-6 lines): 0

Legend:
\NN noun \AJ adjective \PR preposition
\AV adverb \VT trans vb \VI intrans.vb.
\SC subordinate conj. \CC coord. conj.

Figure 15-3. Dialog box for setting the poem format

Setting Up the Poem Format

The next block starts the poem-formatting process by presenting the screen shown in Figure 15-3:

```
set.fmt:
CLS
PRINT "Set Poem Format"
CALL MOVETO(6,fy%(1)+12)
PRINT "Poem length (1-6 lines):"
CALL TEXTSIZE(9)
CALL MOVETO(0,fy1%(7)+24)
CALL TEXTFACE(1)   :REM bold
PRINT " Legend:"
CALL TEXTFACE(0)   :REM light
REM   1234567890124356789012345678901 2435
PRINT " \NN noun \AJ adjective \PR preposition"
PRINT " \AV adverb \VT trans.vb. \VI intrans.vb."
PRINT " \SC subordinate conj. \CC coord. conj."
```

```
CALL TEXTSIZE(12)
EDIT FIELD 1,STR$(nl%),(fx%(1),fy%(1))-(fx1%(1),fy1%(1))
BUTTON 1, 1, bl$(8),(bx%(8),by%(8))-(bx1%(8),by1%(8)),bt%(8)
GOSUB show.format
LET fld%=1
LET new.fld%=1
```

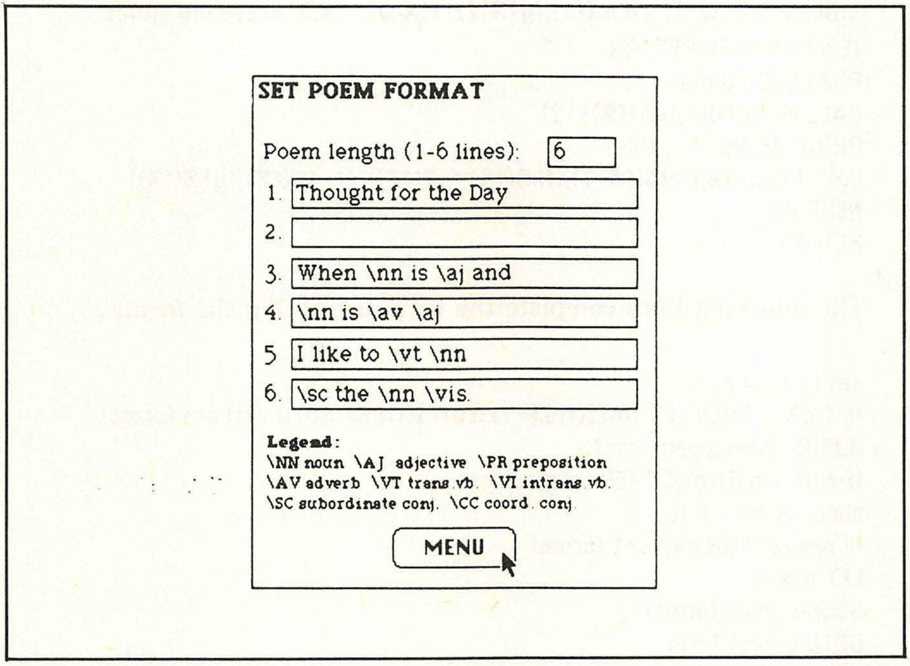
When typing in the "legend" data (\NN noun...) use the REM 1234... line as a guide for lining up the letters properly.

The next block comprises a dialog monitor loop to handle changes to the poem format and requests to return to the menu:

```
set.fmt.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=1 THEN set.fmt.done   :REM Pressed MENU
IF event%=6 OR event%=7 THEN next.field   :REM Enter, Return, or Tab
IF event%<>2 THEN set.fmt.loop   :REM ignore unless a field was selected
LET new.fld%=DIALOG(2)   :REM Which field?
IF fld%=new.fld% THEN set.fmt.loop   :REM Didn't change fields
IF fld%<>1 THEN select.field   :REM Didn't change the poem length
GOSUB check.poem.length
IF NOT length.ok% THEN length.err
```

The following lines handle changes to the poem length as well as requests to edit a different field:

```
check.format.list:
IF nl%=x THEN select.field
LET nl%=x
GOSUB show.format
select.field:
IF fld%>1 THEN LET fmt$(fld%-1)=EDIT$(fld%)
LET fld%=new.fld%
EDIT FIELD fld%
GOTO set.fmt.loop
next.field:
IF nl%=0 THEN LET new.fld%=2 ELSE LET new.fld%=(fld% MOD (nl%+1))+1
IF fld%<>1 THEN select.field
GOSUB check.poem.length
```


SET POEM FORMAT

Poem length (1-6 lines): 6

1. Thought for the Day
2.
3. When \nn is \aj and
4. \nn is \av \aj
5 I like to \vt \nn
6. \sc the \nn \vis.

Legend:
\NN noun \AJ adjective \PR preposition
\AV adverb \VT trans.vb. \VI intrans.vb.
\SC subordinate conj. \CC coord. conj.

Figure 15-4. A completed poem format

```
IF length.ok% THEN check.format.list
length.err:
BEEP
LET new.fld%=1
GOTO select.field
```

The check.format.list routine takes over after you change the poem length. Select.field and next.field respond to requests to change fields (pressing ENTER or RETURN or clicking on an inactive edit field).

The next block contains subroutines that check the length of a newly entered poem and create the necessary number of edit fields to accommodate the poem, as shown in Figure 15-4:

```
check.poem.length:
LET x=VAL(EDIT$(1))
LET length.ok%=(x=INT(x) AND x>=1 AND x<=6)
RETURN
show.format:
```

```
LINE (0,fy%(2)-1)-(ww%(1),fy1%(7)+1),0,bf    :REM Erase old fields
IF nl%=0 THEN RETURN
FOR f%=2 TO nl%+1
CALL MOVETO(6,fy%(f%)+12)
PRINT USING "#."; f%-1
EDIT FIELD f%,fmt$(f%-1),(fx%(f%),fy%(f%))-(fx1%(f%),fy1%(f%))
NEXT f%
RETURN
```

The following lines complete the logic for setting the format:

```
set.fmt.done:
IF fld%>1 THEN LET fmt$(fld%-1)=EDIT$(fld%): GOTO exit.set.format
GOSUB check.poem.length
IF NOT length.ok% THEN length.err
check.format.list:
IF nl%=x THEN exit.set.format
LET nl%=x
GOSUB show.format
GOTO select.field
exit.set.format:
FOR f%=1 TO 7
EDIT FIELD CLOSE f%
NEXT f%
BUTTON CLOSE 1
RETURN
```

These lines are executed when you press the MENU button. If one of the poem format fields was active, the program saves its latest value in the format list (IF fld%>1 THEN LET fmt$(fld%−1)=EDIT$(fld%)...).

Then the exit.set.format procedure closes the edit fields and buttons and returns to the main menu.

Entering a New Word List

The next block creates the screen shown in Figure 15-5:

```
type.in.wds:
CLS
CALL TEXTFACE(1)    :REM bold
PRINT "TYPE IN NEW WORDS"
CALL TEXTFACE(0)    :REM regular
CALL MOVETO(6,fy%(8)-4)
```

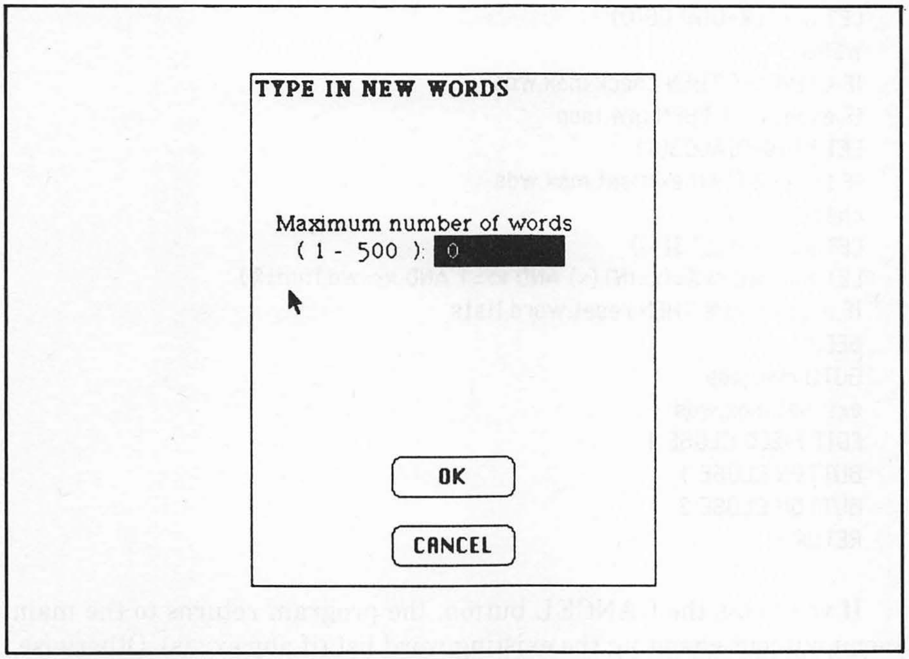


Figure 15-5. Dialog box for typing in new words

```
PRINT " Maximum number of words"
PRINT "      ( 1 -"; wd.limit%;"):"
EDIT FIELD 1,STR$(nw%),(fx%(8),fy%(8))-(fx1%(8),fy1%(8))
BUTTON 1,1,"OK",(bx%(12),by%(12))-(bx1%(12),by1%(12)),bt%(12)
BUTTON 2,1,"CANCEL",(bx%(8),by%(8))-(bx1%(8),by1%(8)),bt%(8)
```

You may specify a maximum vocabulary size up to the value of wd.limit%, which is arbitrarily set to 500 earlier in the program. However, the maximum vocabulary size depends on how much memory your Mac has and how long your words are.

When specifying the maximum vocabulary size, it's a good idea to enter a generous amount (more than you need at that moment), so you can add words to the list later.

The next block waits for you to enter the desired maximum or press the CANCEL button (leaving the word list in its previous form):

```
mw.loop:
LET event%=0
WHILE event%=0
```

```
LET event%=DIALOG(0)
WEND
IF event%=6 THEN check.max.wds
IF event%<>1 THEN mw.loop
LET btn%=DIALOG(1)
IF btn%=2 THEN exit.set.max.wds
check.max.wds:
LET x=VAL(EDIT$(1))
LET max.wd.ok%=(x=INT(x) AND x>=1 AND x<=wd.limit%)
IF max.wd.ok% THEN reset.word.lists
BEEP
GOTO mw.loop
exit.set.max.wds:
EDIT FIELD CLOSE 1
BUTTON CLOSE 1
BUTTON CLOSE 2
RETURN
```

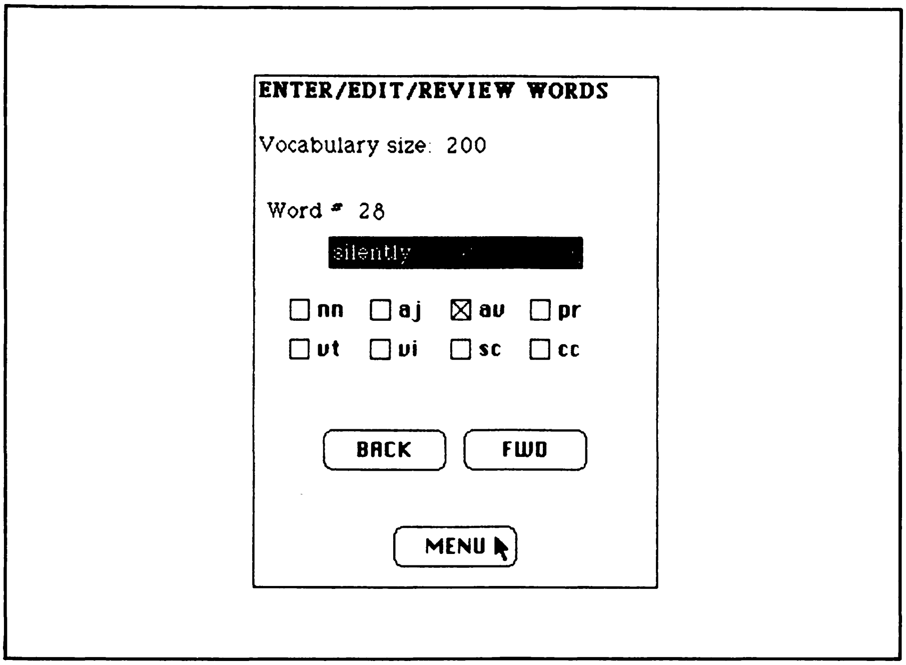
If you press the CANCEL button, the program returns to the main menu without changing the existing word list (if any exists). Otherwise, the following block resets the word list arrays to accommodate the newly specified size:

```
reset.word.lists:
EDIT FIELD CLOSE 1
BUTTON CLOSE 1
BUTTON CLOSE 2
LET nw%=x
ERASE wl$,wt%
DIM wl$(nw%),wt%(nw%)
```

Entering, Editing, and Reviewing Words

The next block presents the screen shown in Figure 15-6:

```
edit.wds:
CLS
CALL TEXTFACE(1)    :REM bold
PRINT "ENTER/EDIT/REVIEW WORDS"
CALL TEXTFACE(0)    :REM regular
PRINT
PRINT "Vocabulary size:"; nw%
FOR b%=1 TO 8
```



ENTER/EDIT/REVIEW WORDS

Vocabulary size: 200

Word # 28

silently

Figure 15-6. Dialog box for entering, editing, and reviewing words

```
LET b.x%=18+((b%-1) MOD 4)*1.3*b.wid%
LET b.y%=fy1%(10)+18+((b%-1)\4)*1.5*b.len%
BUTTON b%,1,w.type$(b%),(b.x%,b.y%)-(b.x%+b.wid%,b.y%+b.len%),2
NEXT b%
FOR b%=13 TO 14
BUTTON b%-4,1,bl$(b%),(bx%(b%),by%(b%))-(bx1%(b%),by1%(b%)),bt%(b%)
NEXT b%
BUTTON 11,1,bl$(8),(bx%(8),by%(8))-(bx1%(8),by1%(8)),bt%(8)
LET cw%=1   :REM start with first word in the list
last.wt%=0
```

Cw% points to the current position in the word list. Last.wt% keeps track of the word type of the most recently edited word.

The following lines complete the screen shown in Figure 15-6:

```
edit.loop:
CALL MOVETO(6,fy%(10)-12)
PRINT "Word #"; cw%
```

```
EDIT FIELD 1,wl$(cw%),(fx%(10),fy%(10))-(fx1%(10),fy1%(10))
FOR b%=1 TO 8
IF wt%(cw%)=0 THEN LET wt%(cw%)=last.wt%
LET b.on%=(b%=wt%(cw%))
BUTTON b%,1-b.on%
NEXT b%
```

The variable b.on% determines which one of the eight word-type boxes is selected, depending on the word type of the current word, wt%(cw%).

The following block waits for a dialog event:

```
wait.edit.event:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=6 THEN forward
IF event%<>1 THEN wait.edit.event
LET edit.btn%=DIALOG(1)
IF edit.btn%<9 THEN change.word.type
ON edit.btn%-8 GOTO back,forward,exit.edit
change.word.type:
IF wt%(cw%)<>0 THEN BUTTON wt%(cw%),1    :REM Deselect old type
LET wt%(cw%)=edit.btn%
BUTTON wt%(cw%),2
GOTO wait.edit.event
```

Pressing one of the eight check-boxes activates the change.word.type routine. Pressing ENTER or RETURN is equivalent to pressing the Forward (FWD) button, which displays the next word in the list. The next block accomplishes this.

```
forward:
LET wl$(cw%)=EDIT$(1)
IF wl$(cw%)=nu$ THEN LET wt%(cw%)=0
LET last.wt%=wt%(cw%)    :REM save latest word type
LET cw%=(cw% MOD nw%)+1    :REM move to next word
GOTO edit.loop
back:
LET wl$(cw%)=EDIT$(1)
IF wl$(cw%)=nu$ THEN LET wt%(cw%)=0
```

```
LET last.wt%=wt%(cw%)   :REM save latest word type
IF cw%=1 THEN LET cw%=nw% ELSE LET cw%=cw%-1   :REM preceding word
GOTO edit.loop
exit.edit:
LET wl$(cw%)=EDIT$(1)
IF wl$(cw%)=nu$ THEN LET wt%(cw%)=0
IF wl$(cw%)<>nu$ AND wt%(cw%)=0 THEN BEEP: GOTO wait.edit.event
EDIT FIELD CLOSE 1
FOR b%=1 TO 11
BUTTON CLOSE b%
NEXT b%
```

If you skip to the next field while leaving the previous word field empty, the program ensures that the word type is 0 (undefined). If you enter a word, the program will not let you move to the next word until the word type is also specified (IF wl$(cw%)< >nu$ AND wt%(cw%) =0...).

Sorting the New Word List

Each time you enter a new word list or edit an existing one, the program must sort the list again. The words are sorted according to the eight grammatical categories:

```
CLS
PRINT "Sorting the words..."
ERASE kn%,pn%
DIM kn%(8),pn%(8)
FOR cw%=1 TO nw%   :REM count how many words of each type
IF wt%(cw%)=0 THEN next.word.a   :REM skip empty entries
LET kn%(wt%(cw%))=kn%(wt%(cw%))+1
next.word.a:
NEXT cw%
ERASE nn%,aj%,av%,pr%,vt%,vi%,sc%,cc%
DIM nn%(kn%(1)), aj%(kn%(2)),av%(kn%(3)),pr%(kn%(4))
DIM vt%(kn%(5)),vi%(kn%(6)),sc%(kn%(7)),cc%(kn%(8))
```

First the counter and pointer arrays kn%( ) and pn%( ) are set to zero. Then the program counts the number of nouns, adjectives, and so forth. Next the program creates a suitably sized array for each word type (DIM nn%(kn%(1))...).

The following lines read through the word list wl$( ) to set up the proper values in the index arrays like nn%( ):

```
FOR cw%=1 TO nw%
IF wt%(cw%)=0 THEN next.word.b
LET pn%(wt%(cw%))=pn%(wt%(cw%))+1
ON wt%(cw%) GOTO noun,adj,adv,prep,verbt,verbi,subj,conj
noun:
nn%(pn%(1))=cw%
GOTO next.word.b
adj:
aj%(pn%(2))=cw%
GOTO next.word.b
adv:
av%(pn%(3))=cw%
GOTO next.word.b
prep:
pr%(pn%(4))=cw%
GOTO next.word.b
verbt:
vt%(pn%(5))=cw%
GOTO next.word.b
verbi:
vi%(pn%(6))=cw%
GOTO next.word.b
subj:
sc%(pn%(7))=cw%
GOTO next.word.b
conj:
cc%(pn%(8))=cw%
next.word.b:
NEXT cw%
RETURN
```

Upon completion of these lines, nn%(i) gives the location of the *i*th noun in the word list wl$( ), aj%(k) gives the location of the *k*th adjective, and so forth.

After this categorization process, the program returns to the main menu.

Making Poems

The Make poems command activates these lines:

```
make.poems:
LET np%=1
LET device%=1    :REM output to screen
make.poems.again:
CLS
CALL TEXTFACE(1)
PRINT "MAKE POEMS"
CALL TEXTFACE(0)
CALL MOVETO(6,fy%(9)-12)
PRINT "How many poems ( 1 -"; max.poems%; "):"
EDIT FIELD 1,STR$(np%),(fx%(9),fy%(9))-(fx1%(9),fy1%(9))
CALL MOVETO(6,by%(9)-4)
PRINT "Output to:"
FOR b%=9 TO 11
LET b.on%=((b%-8)=device%)
BUTTON
     b%-8,1-b.on%,bl$(b%),(bx%(b%),by%(b%))-(bx1%(b%),by1%(b%)),bt%(b%)
NEXT b%
BUTTON 4,1,bl$(12),(bx%(12),by%(12))-(bx1%(12),by1%(12)),bt%(12)   :REM
     poem
BUTTON 5,1,bl$(8),(bx%(8),by%(8))-(bx1%(8),by1%(8)),bt%(8)   :REM menu
```

These lines create the screen shown in Figure 15-7, prompting you to specify the number of poems to be generated and the output device.

The next block waits for your command:

```
mp.loop:
LET event%=0
WHILE event%=0
LET event%=DIALOG(0)
WEND
IF event%=6 THEN GOSUB check.mp: IF mp.ok% THEN mp.ready ELSE mp.loop
IF event%<>1 THEN mp.loop
LET btn%=DIALOG(1)
IF btn%<4 THEN change.device
IF btn%=5 THEN exit.mp
GOSUB check.mp
IF mp.ok% THEN mp.ready ELSE mp.loop
```

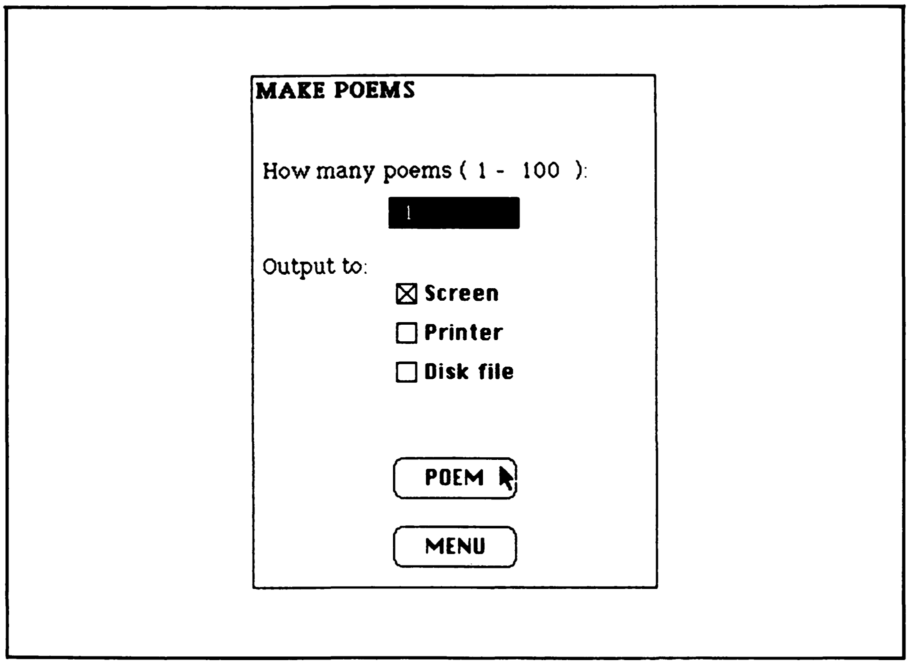


Figure 15-7. This dialog box lets you specify the poem quantity and output device

The following lines respond to requests to change the output device, return to the menu, and change the number of poems to generate.

```
change.device:
BUTTON device%,1
LET device%=btn%
BUTTON device%,2
GOTO mp.loop
exit.mp:
EDIT FIELD CLOSE 1
FOR b%=1 TO 5
BUTTON CLOSE b%
NEXT b%
RETURN
check.mp:
LET x=VAL(EDIT$(1))
LET mp.ok%=(x=INT(x)) AND (x>0) AND (x<=max.poems%)
IF NOT mp.ok% THEN BEEP ELSE LET np%=x
RETURN
```

Once the poem quantity and output device are set, the following lines close the edit field and buttons and open the output device:

```
mp.ready:
EDIT FIELD CLOSE 1
FOR b%=1 TO 5
BUTTON CLOSE b%
NEXT b%
CLS
LET fo$=device$(device%)
CALL TEXTFACE(1)
PRINT "OUTPUT"; np%; "TO "; FO$
CALL TEXTFACE(0)
IF device%<3 THEN device.ready
LET fo$=FILES$(0,"Name the output file")
IF fo$=nu$ THEN make.poems
```

The disk is handled specially, since you must specify the output file name. The FILES$ function waits for you to enter a valid file name.

Poem Generation

The next lines read in the lines of the poem format and locate every abbreviated grammatical code:

```
device.ready:
WINDOW 2,,(wx%(2),wy%(2))-(wx1%(2),wy1%(2)),3
WINDOW 1
OPEN fo$ FOR OUTPUT AS 1
BUTTON 1,1,"CANCEL",(bx%(12),by%(12))-(bx1%(12),by1%(12)),bt%(12)
WINDOW OUTPUT 2
FOR poem%=1 TO np%
FOR lin%=1 TO nl%
LET clin$=fmt$(lin%)
LET llen%=LEN(clin$)
LET c.pos%=1
WHILE c.pos%<=llen%
LET mark.pos%=INSTR(c.pos%,clin$,mark$)
IF mark.pos%=0 THEN finish.line
IF device%<>1 THEN PRINT MID$(clin$,c.pos%,mark.pos%-c.pos%);
PRINT #1, MID$(clin$,c.pos%,mark.pos%-c.pos%);
LET code$=UCASE$(MID$(clin$,mark.pos%,3))
LET which%=INT((INSTR(1,valid.codes$,code$)-1)/3)+1
```

```
IF which%=0 THEN literal.characters
LET c.pos%=mark.pos%+3
IF kn%(which%)=0 THEN none.available
LET word.ptr%=INT(RND*kn%(which%))+1
```

Llen% is the length of the current format line clin$. C.pos% is the current position within clin$. To locate the grammatical codes, the program searches for occurrences of the "\" mark. If there are no occurrences to the right of c.pos%, the program prints out the rest of the line from c.pos onward and proceeds to the next line.

After finding a "\" mark, the program examines the three-character sequence beginning with "\" to see if it matches one of the grammatical abbreviations contained in valid.codes$. If it does match, the variable which% indicates the word type (1=noun, 2=adjective, and so forth). In that case, the computer randomly selects a word pointer word.ptr%.

The next lines read a word from the appropriate category:

```
ON which% GOTO get.nn,get.aj,get.av,get.pr,get.vt,get.vi,get.sc,get.cc
get.nn:
LET random.word$=wl$(nn%(word.ptr%))
GOTO print.random.word
get.aj:
LET random.word$=wl$(aj%(word.ptr%))
GOTO print.random.word
get.av:
LET random.word$=wl$(av%(word.ptr%))
GOTO print.random.word
get.pr:
LET random.word$=wl$(pr%(word.ptr%))
GOTO print.random.word
get.vt:
LET random.word$=wl$(vt%(word.ptr%))
GOTO print.random.word
get.vi:
LET random.word$=wl$(vi%(word.ptr%))
GOTO print.random.word
get.sc:
LET random.word$=wl$(sc%(word.ptr%))
GOTO print.random.word
get.cc:
LET random.word$=wl$(cc%(word.ptr%))
print.random.word:
IF device%<>1 THEN PRINT random.word$;
```

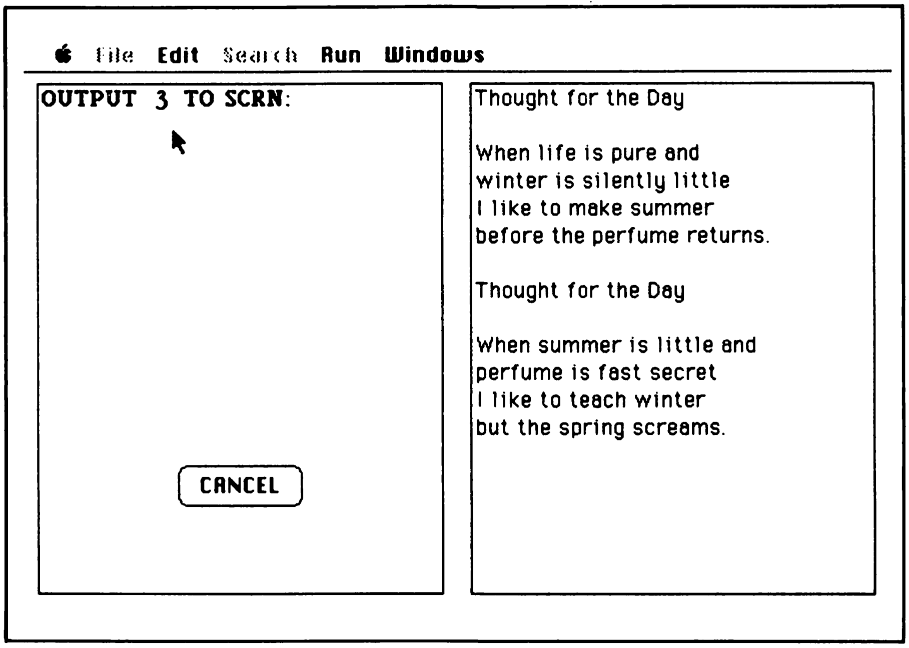
```
PRINT #1, random.word$;
GOTO next.position
none.available:
IF device%<>1 THEN PRINT code$;
PRINT #1, code$;
GOTO next.position
```

If the word category is empty, the none.available routine simply prints out the three-letter code in place of a randomly chosen word.

Here's the routine that prints the remainder of a line containing no more codes and the routine that prints a literal character (that is, any character that is not part of a code):

```
finish.line:
IF device%<>1 THEN PRINT RIGHT$(clin$,llen%-c.pos%+1);
PRINT #1,RIGHT$(clin$,llen%-c.pos%+1);
LET c.pos%=llen%+1
GOTO next.position
literal.characters:
IF device%<>1 THEN PRINT MID$(clin$,mark.pos%,1);
PRINT #1,MID$(clin$,mark.pos%,1);
LET c.pos%=c.pos%+1
next.position:
WEND
IF device%<>1 THEN PRINT
PRINT#1,
NEXT lin%
IF device%<>1 THEN PRINT
PRINT#1,
IF DIALOG(0)=1 THEN LET poem%=np%
NEXT poem%
WINDOW 1
CLOSE 1
CLS
PRINT "Finished."
GOSUB wait.ok
WINDOW CLOSE 2
GOTO make.poems.again
```

As each poem is completed, the computer checks to see whether the CANCEL button was pressed. If not, the program continues with the next poem until all the requested poems have been output.



```
Thought for the Day

When life is pure and
winter is silently little
I like to make summer
before the perfume returns.

Thought for the Day

When summer is little and
perfume is fast secret
I like to teach winter
but the spring screams.
```

Figure 15-8. Sample poem output to the screen

Figure 15-8 shows the screen appearance after a poem has been written to the screen.

Loading a Word List

Pressing the Load words button on the main menu activates these lines:

```
load.wds:
CLS
CALL TEXTFACE(1)   :REM bold
PRINT "LOAD WORDS"
CALL TEXTFACE(0)   :REM regular
PRINT "Name the input file:"
LET fi$=FILES$(1,"TEXT")
IF fi$=nu$ THEN RETURN
ON ERROR GOTO load.err
OPEN fi$ FOR INPUT AS 1
INPUT#1, x
```

```
IF x<>INT(x) OR x<1 OR x>wd.limit% THEN file.err.nw
LET nw%=x
ERASE wl$,wt%
DIM wl$(nw%),wt%(nw%)
FOR cw%=1 TO nw%
LINE INPUT#1, wl$(cw%)
INPUT#1, wt%(cw%)
NEXT cw%
CLOSE
ON ERROR GOTO 0
GOTO edit.wds
```

This routine creates a window similar to that shown in Figure 15-9. The foreground window is provided by the FILES$ function. The program opens the specified file fi$, reads in a value for nw%, the number of words in the file, and reads in the vocabulary words and word types.

In case of an error during loading, the following lines take over:

```
load.err:
IF ERR>=50 THEN load.disk.related
IF ERR<>6 AND ERR<>13 AND ERR<>23 THEN ON ERROR GOTO 0
load.disk.related:
CLOSE
```

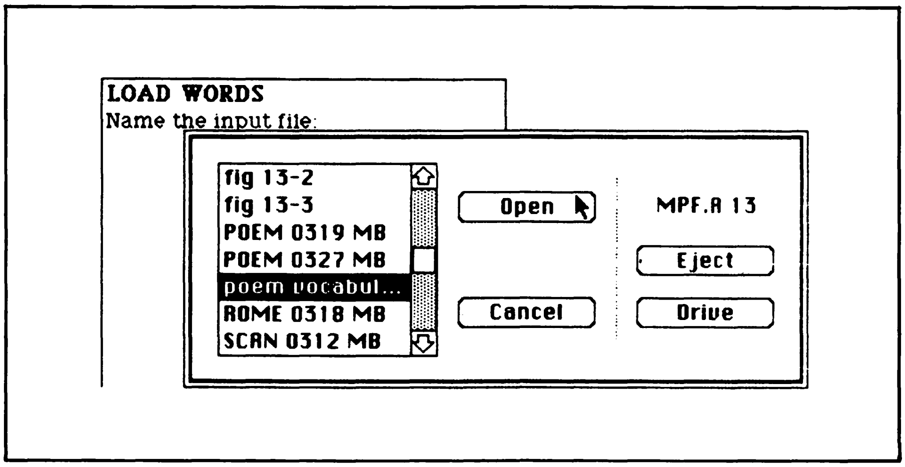

LOAD WORDS
Name the input file:

Figure 15-9. Screen appearance when you select the Load words command

```
BEEP
PRINT
PRINT "Can't load data from"
PRINT fi$
PRINT "Use Macwrite to check"
PRINT "the file data and format."
GOSUB wait.ok
RESUME err.exit
file.err.nw:
CLOSE
BEEP
PRINT
PRINT fi$
PRINT "contains"; x; "words."
PRINT "The range is 1 -"; wd.limit%
PRINT "Use Macwrite to check the file."
err.exit:
ON ERROR GOTO 0
GOSUB wait.ok
RETURN
```

There are two error-handling routines, one for disk-related errors (load.disk.related) and another for the case of nw% being out of range (file.err.nw).

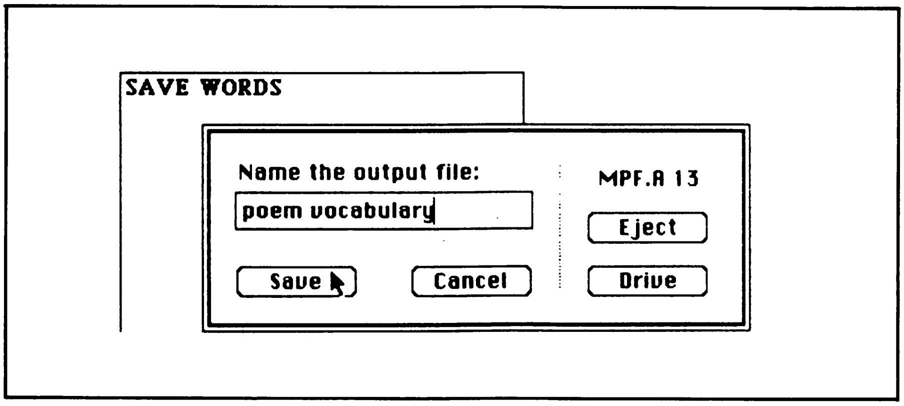
The following subroutine provides a pause in program operation until you press the OK button:

```
wait.ok:
BUTTON 1,1,"OK", (bx%(8),by%(8))-(bx1%(8),by1%(8)),bt%(8)
LET event%=0
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
BUTTON CLOSE 1
RETURN
```

Saving a Word List

Pressing the Save words button on the main menu activates these lines:

```
save.wds:
CLS
CALL TEXTFACE(1)    :REM bold
```


SAVE WORDS

Name the output file:

poem vocabulary

Figure 15-10. Screen appearance when you select the Save words command

```
PRINT "SAVE WORDS"
CALL TEXTFACE(0)    :REM regular
LET fo$=FILES$(0,"Name the output file:")
IF fo$=nu$ THEN RETURN
ON ERROR GOTO save.err:
OPEN fo$ FOR OUTPUT AS 1
PRINT#1, nw%
FOR cw%=1 TO nw%
PRINT#1, wl$(cw%)
PRINT#1, wt%(cw%)
NEXT cw%
CLOSE
ON ERROR GOTO 0
RETURN
```

The program presents a screen like that shown in Figure 15-10. Pressing Cancel returns you to the main menu without saving the vocabulary; pressing ENTER, RETURN, or Save causes the words to be saved in the specified file, as shown in Figure 15-11.

In case of an error, the following routine (the last block of our program) takes over:

```
save.err:
CLOSE
IF ERR<50 THEN ON ERROR GOTO 0
BEEP
```

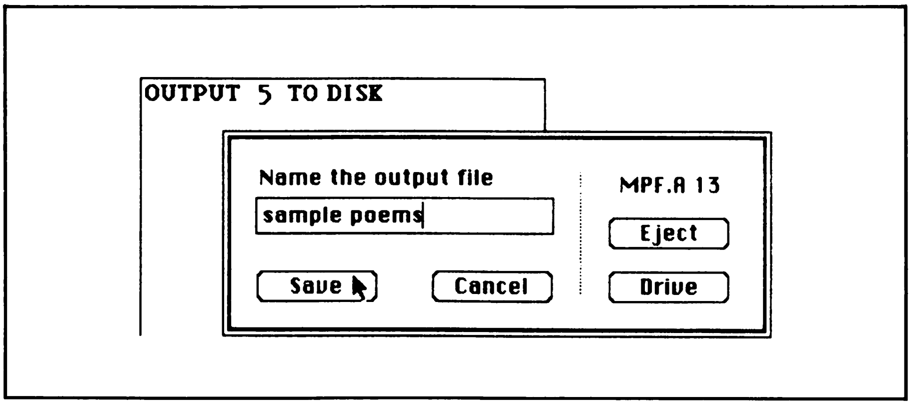


Figure 15-11. Dialog box to let you specify the output file name for saving poems to disk

```
PRINT
PRINT "Can't save the data in"
PRINT fo$
GOTO err.exit
```

If the error is not disk-related, the program lets Microsoft BASIC handle the error in its normal manner (IF ERR<50 THEN ON ERROR GOTO 0). If it is a disk error, the program informs you and returns to the main menu. In that case, your disk probably has a file containing part of the word list.

—Putting the Program to Work

Now the research begins. Select an assortment of words—take them at random from a book of poems or any other source. Type them in using the Type in new words command. You may type the words in any sequence; however, it is a little faster if you have already sorted the words into categories, because you won't have to change the word-type button settings so often. Save the list on disk.

Experiment with various formats. Try including prefixes, suffixes, and inflectional endings for special effects. For instance, the format line:

```
\viing the \nns of your \nn
```

might generate:

walking the rivers of your mind

On the other hand, it might also produce:

breaksing the dresss of your lawn

depending on how well your vocabulary is suited to the poem format.

This chapter is adapted from "Roll Over, Robert Frost" by George Stewart, appearing in the February 1983 issue of *Popular Computing* magazine. Copyright 1983 Byte Publications, Inc. Used with the permission of Byte Publications, Inc.

Chapter 16

Designs in a Circle

In this chapter we turn the Macintosh into a design toy that lets you create fascinating geometric patterns on the screen. Patterns may be saved on disk and loaded into MacPaint or printed out on the Imagewriter printer.

The program simulates the mechanism shown in Figure 16-1. A real-world version is available at toy stores under the name Spirograph.

To draw a design, you select a cog wheel and place it inside the track. You then place a pen into one of several holes in the wheel; using the pen as a handle, you rotate the wheel along the inside circumference of the track. As the wheel rotates, the pen creates a design on the paper. By varying the wheel size, pen location, and the size of the cogs, you can generate a wide variety of designs. For further elaboration, you can superimpose one curve on another by changing the wheel and pen setup and using the same sheet of paper (512K Macintoshes only).

Figure 16-2 shows a few of the countless designs you can make with this program.

The simulated version of the design tool operates quite similarly to

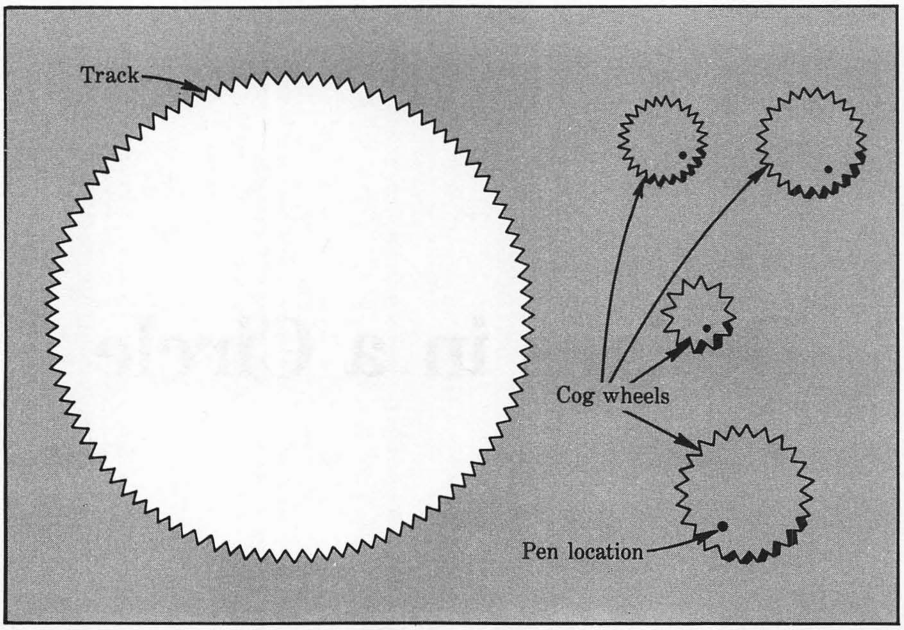


Figure 16-1. The design toy that the program simulates

the real thing, except that you can change the track size, wheel size, pen location, and cog size simply by clicking the mouse.

Figure 16-3 presents the tool-setup menu, and Figure 16-4 shows the various instruction screens explaining how you change the tool settings.

—The Program

The first program block sets up the window parameters:

```
REM Window descriptors
DATA 7.0,4.4, 0.04, 0.32
READ inches.wide, inches.long, ulcx,ulcy
LET ww%=INT(inches.wide*72)
LET wl%=INT(inches.long*72)
LET wx%=INT(ulcx*72)
LET wy%=INT(ulcy*72)
LET wx1%=wx%+ww%
LET wy1%=wy%+wl%
```

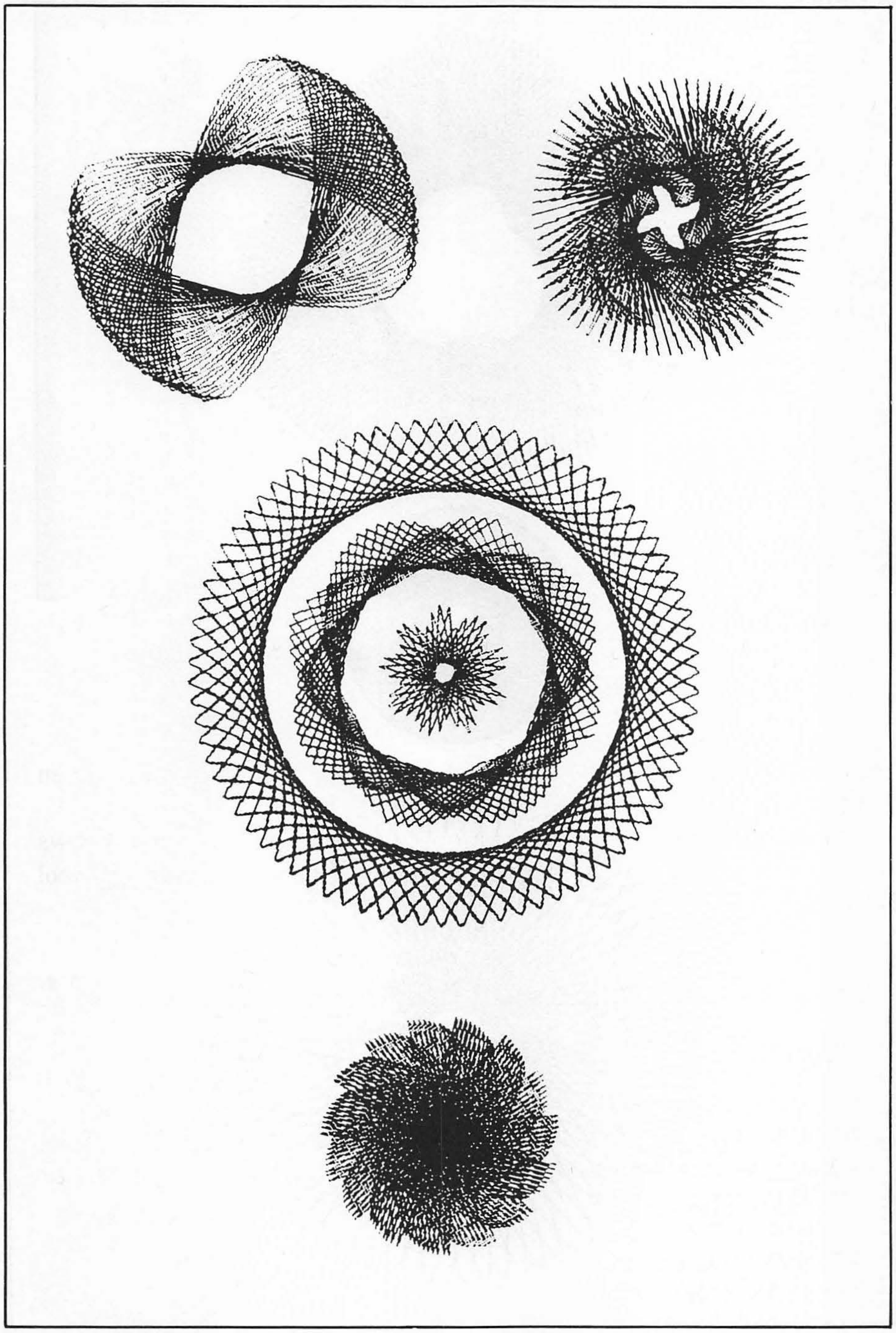


Figure 16-2. Sample designs created by the program

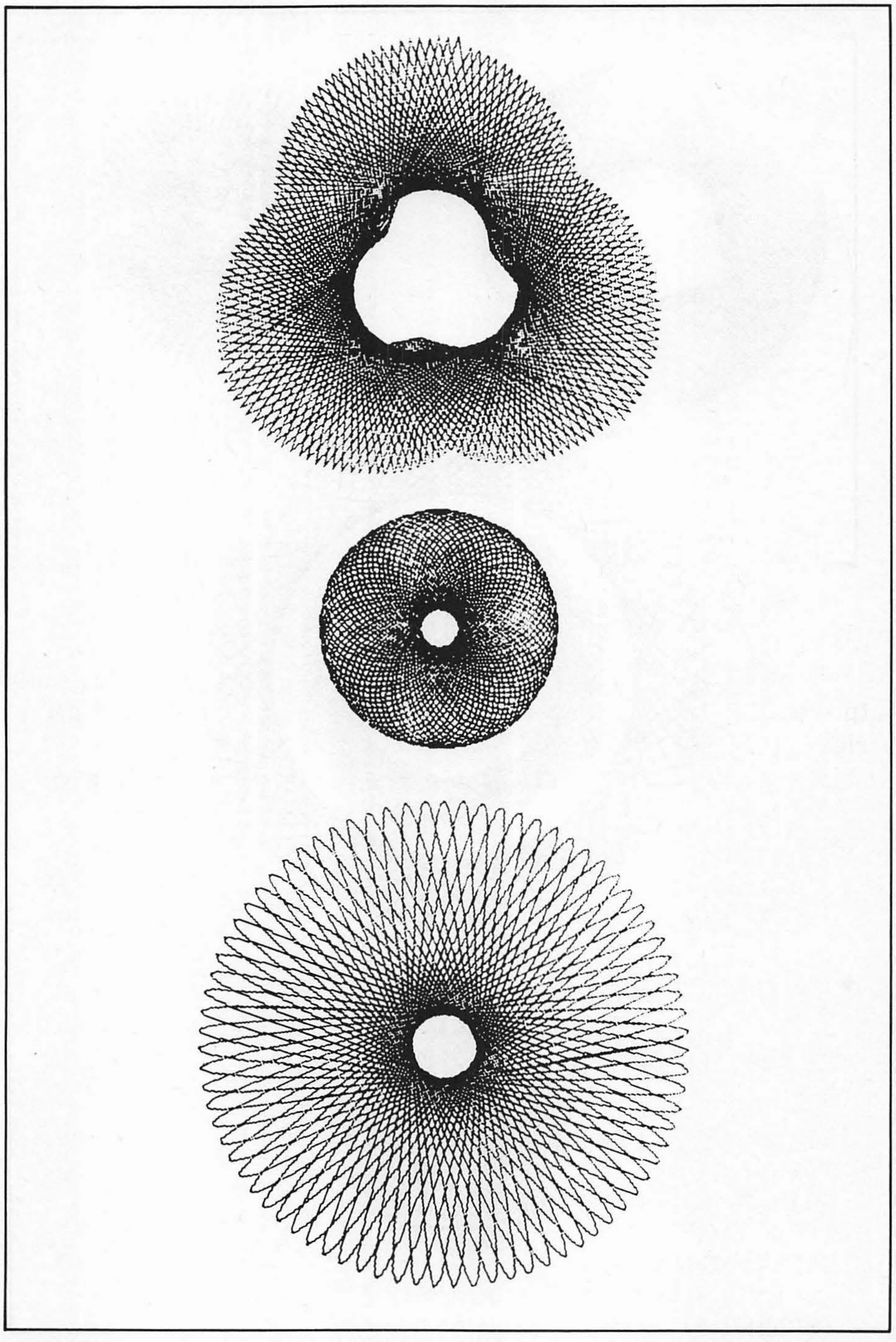


Figure 16-2. Sample designs created by the program (*continued*)

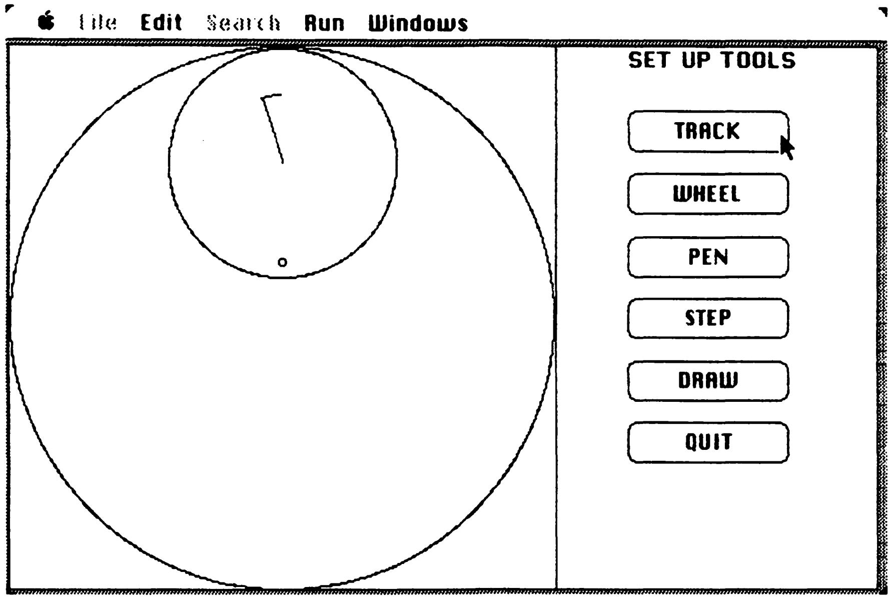
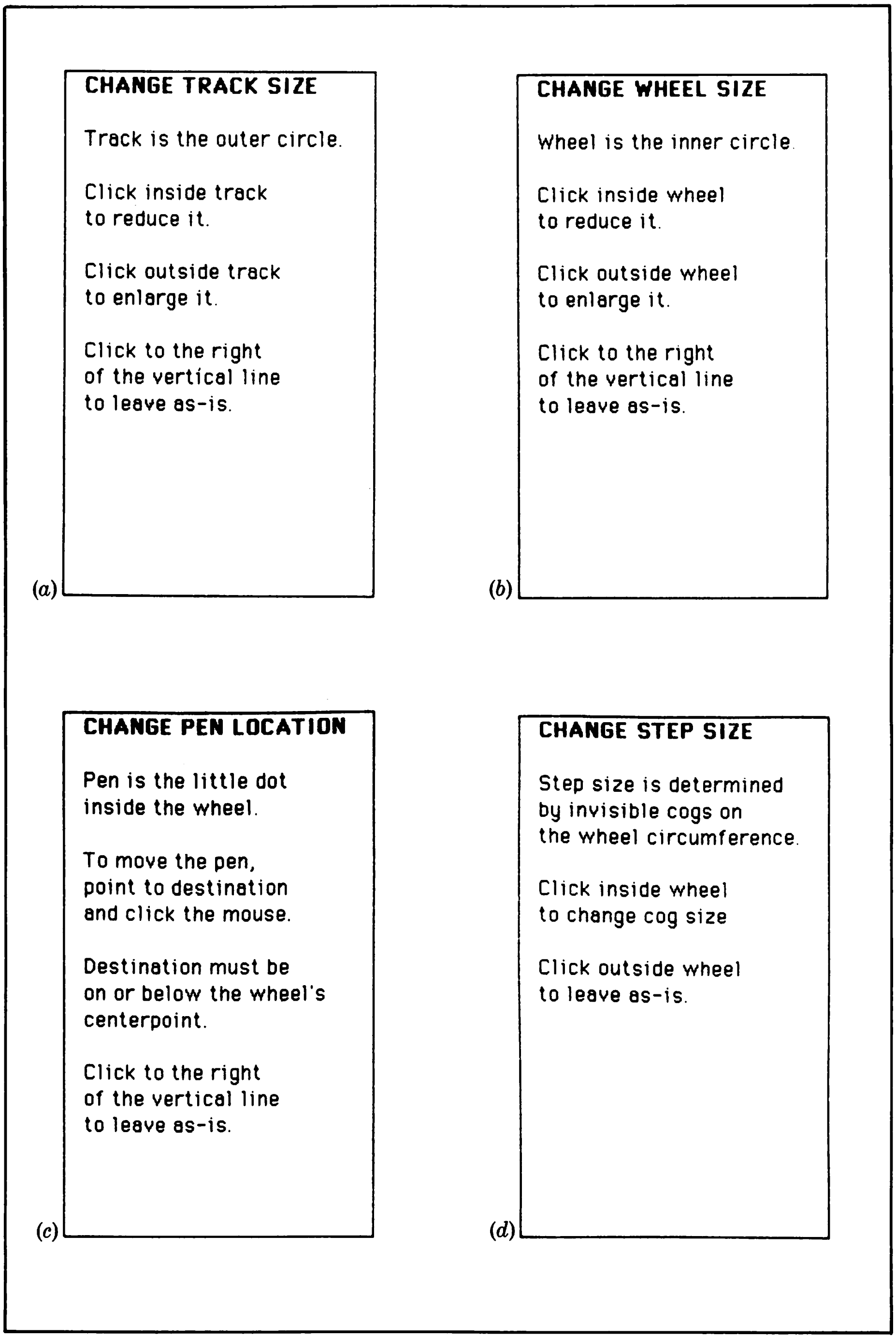


Figure 16-3. The Set Up Tools Menu

Ww% and wl% are the window's width and length in display units (pixels). Wx%,wy% and wx1%,wy1% specify the upper-left and lower-right corners of the window.

The next block describes the dialog buttons used in the program.

```
REM Button descriptors
DATA 7
DATA DRAW, 1.3, 0.333, 0.875, 0.125
DATA CHANGE, 1.3, 0.333, 0.875, 0.250
DATA QUIT, 1.3, 0.333, 0.875, 0.375
DATA STOP, 1.3, 0.333, 0.875, 0.500
DATA OK, 1.3, 0.333, 0.875, 0.625
DATA QUIT,1.3, 0.333, 0.875, 0.750
DATA RECALL,1.3, 0.333, 0.875, 0.875
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%)
FOR n%=1 TO nb%
READ bl$(n%),inches.wide, inches.long, hzone, vzone
LET bx%(n%)=(ww%-inches.wide*72)*hzone
LET by%(n%)=(wl%-inches.long*72)*vzone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```



(a)

CHANGE TRACK SIZE

Track is the outer circle.

Click inside track
to reduce it.

Click outside track
to enlarge it.

Click to the right
of the vertical line
to leave as-is.

(b)

CHANGE WHEEL SIZE

Wheel is the inner circle.

Click inside wheel
to reduce it.

Click outside wheel
to enlarge it.

Click to the right
of the vertical line
to leave as-is.

(c)

CHANGE PEN LOCATION

Pen is the little dot
inside the wheel.

To move the pen,
point to destination
and click the mouse.

Destination must be
on or below the wheel's
centerpoint.

Click to the right
of the vertical line
to leave as-is.

(d)

CHANGE STEP SIZE

Step size is determined
by invisible cogs on
the wheel circumference.

Click inside wheel
to change cog size

Click outside wheel
to leave as-is.

Figure 16-4. Instructions for changing (a) the track size, (b) the wheel size, (c) the pen location, and (d) the step size

Each button is described in terms of its label, width, length, horizontal zone, and vertical zone. Look at Figures 16-3, 16-5, and 16-6 to see where each button is used. Some of the buttons are displayed with labels different from those given in the DATA statements.

Setting Up Design Constants and Parameters

The next block sets up certain values that remain constant throughout the program:

```
LET yes%=(1=1)
LET no%=(1=0)
LET pi=4*ATN(1)
DEF FNdist(x1,y1,x2,y2)=SQR((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))
LET max.t.dia%=w1%
LET max.t.rad%=max.t.dia%\2
LET t.scale%=12
LET w.scale%=12
LET cog.scale%=20
LET min.t.rtio%=3
LET max.w.rtio%=w.scale%-1
LET min.w.rtio%=1
LET tc.x%=max.t.rad%
LET tc.y%=max.t.rad%
LET wc.x%=tc.x%
```

Pi is the ratio of a circle's circumference to its diameter. Max.t.dia%, the maximum track diameter, is set so that the largest track completely fills the window from top to bottom. T.scale%, w.scale% , and cog.scale% determine the number of different track, wheel, and cog sizes available. Min.t.rtio% specifies the smallest track size; the actual minimum track and wheel radii are given by

smallest track radius = max.t.rad% * min.t.rtio% / t.scale%

smallest wheel radius = max.t.rad% * min.w.rtio% / w.scale%

The following lines set up the initial values for the track, wheel, pen location, and cog ratio:

```
LET t.rtio%=t.scale%
LET w.rtio%=5
LET p.rtio=.875   :REM ratio of pen-rad to wheel-rad
LET cog.size%=1
LET pen.x%=wc.x%
LET mem.req%=0
```

```
IF FRE(1)<15000 THEN nsf.memory
LET mem.req%=1+(INT((max.t.dia%+1)/16)+1)*(max.t.dia%+1)
DIM curve%(mem.req%)
nsf.memory:
DIM previous%(6),pause.btn$(1)
LET pause.btn$(0)="PAUSE"
LET pause.btn$(1)="CONTINUE"
WINDOW 1,,(wx%,wy%)-(wx1%,wy1%),3
GOSUB calculate.params
```

T.rtio% is the current track ratio; the corresponding radius is

t.rad%=t.rtio% / t.scale%

W.rtio% is the current wheel ratio; the corresponding radius is

w.rad%=t.rad% * w.rtio% / w.scale%

Cog.size% sets the numerator of the cog ratio (the denominator is always cog.scale%). With 512K Macintoshes only, mem.req% stores the number of bytes needed to store an image of the graphics design when you press the STORE button. The array curve%( ) is set up to hold this information. The array previous%( ) stores the previous contents of the small area surrounding the pen before the pen is drawn. This allows the area to be restored when the pen marker is moved.

The Set Up Tools Menu

The following block creates the display shown in Figure 16-3:

```
change.params:
CLS
LINE (max.t.dia%+1,0)-STEP(0,max.t.dia%)
GOSUB show.params
cp.loop:
CALL MOVETO(bx%(1),12)
CALL TEXTFACE(1)
PRINT "SET UP TOOLS"
CALL TEXTFACE(0)
BUTTON 1,1,"TRACK",(bx%(1),by%(1))-(bx1%(1),by1%(1)),1
BUTTON 2,1,"WHEEL",(bx%(2),by%(2))-(bx1%(2),by1%(2)),1
BUTTON 3,1,"PEN",(bx%(3),by%(3))-(bx1%(3),by1%(3)),1
BUTTON 4,1,"STEP",(bx%(4),by%(4))-(bx1%(4),by1%(4)),1
BUTTON 5, 1, "DRAW",(bx%(5),by%(5))-(bx1%(5),by1%(5)),1
BUTTON 6, 1, "QUIT",(bx%(6),by%(6))-(bx1%(6),by1%(6)),1
```

The first four buttons activate routines to change the design parameters. DRAW starts the drawing, and QUIT ends the program.

The next line waits for you to press a button:

```
WHILE DIALOG(0)<>1
WEND
FOR b%=1 TO 6
BUTTON CLOSE b%
NEXT b%
LET btn%=DIALOG(1)
IF btn%=6 THEN END
IF btn%=5 THEN draw.curve
ON btn% GOSUB resize.the.track,resize.the.wheel,set.pen.radius,set.step
GOTO cp.loop
```

Pressing button 6 ends the program, and button 5 transfers control to the draw.curve routine. Pressing one of the other buttons activates a corresponding tool-setup subroutine.

Changing the Track Size

The following lines print the instructions shown in Figure 16-4a.

```
resize.the.track:
LET event%=MOUSE(0)    :REM clear previous mouse-clicks
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
CALL MOVETO(bx%(1)-30,12)
CALL TEXTFACE(1)
PRINT "CHANGE TRACK SIZE"
CALL TEXTFACE(0)
PRINT
PRINT PTAB(bx%(1)-30); "Track is the outer circle."
PRINT
PRINT PTAB(bx%(1)-30); "Click inside track"
PRINT PTAB(bx%(1)-30); "to reduce it."
PRINT
PRINT PTAB(bx%(1)-30); "Click outside track"
PRINT PTAB(bx%(1)-30); "to enlarge it."
PRINT
PRINT PTAB(bx%(1)-30); "Click to the right"
PRINT PTAB(bx%(1)-30); "of the vertical line"
PRINT PTAB(bx%(1)-30); "to leave as-is."
```

The next block responds to the changes that you make:

```
rtt.loop:
WHILE MOUSE(0)<=0
WEND
LET mx%=MOUSE(1)
LET my%=MOUSE(2)
LET outside.field%=mx%>max.t.dia%+1
LET outside.track%=FNdist(mx%,my%,tc.x%,tc.y%)>t.rad%
IF outside.field% THEN rtt.done
IF outside.track% THEN enlarge.track
IF t.rtio%=min.t.rtio% THEN rtt.loop
LET t.rtio%=t.rtio%-1
GOTO t.rtio.ready
enlarge.track:
IF t.rtio%=t.scale% THEN rtt.loop
LET t.rtio%=t.rtio%+1
t.rtio.ready:
GOSUB erase.params
GOSUB calculate.params
GOSUB show.params
GOTO rtt.loop
rtt.done:
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
RETURN
```

Clicking the mouse button to the right of the vertical line causes the program to return to the main setup menu (IF outside.field% THEN...). Clicking outside the track (but to the right of the vertical line) enlarges the track; clicking inside the track reduces its size.

Each time you enlarge or reduce the track, the program recalculates the sizes for everything that is inside the track and then redraws all these objects (GOSUB erase.params, GOSUB calculate.params, and GOSUB show.params).

Changing the Wheel Size

Here are the lines that print the instructions shown in Figure 16-4b:

```
resize.the.wheel:
LET event%=MOUSE(0)   :REM clear previous mouse-clicks
```

```
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
CALL MOVETO(bx%(1)-30,12)
CALL TEXTFACE(1)
PRINT "CHANGE WHEEL SIZE"
CALL TEXTFACE(0)
PRINT
PRINT PTAB(bx%(1)-30); "Wheel is the inner circle."
PRINT
PRINT PTAB(bx%(1)-30); "Click inside wheel"
PRINT PTAB(bx%(1)-30); "to reduce it."
PRINT
PRINT PTAB(bx%(1)-30); "Click outside wheel"
PRINT PTAB(bx%(1)-30); "to enlarge it."
PRINT
PRINT PTAB(bx%(1)-30); "Click to the right"
PRINT PTAB(bx%(1)-30); "of the vertical line"
PRINT PTAB(bx%(1)-30); "to leave as-is."
```

The next lines monitor your requests to change the wheel size:

```
rtw.loop:
WHILE MOUSE(0)<=0
WEND
LET mx%=MOUSE(1)
LET my%=MOUSE(2)
LET outside.wheel%=FNdist(mx%,my%,wc.x%,wc.y%)>w.rad%
LET outside.field%=mx%>max.t.dia%+1
IF outside.field% THEN rtw.done
IF outside.wheel% THEN enlarge.wheel
IF w.rtio%=min.w.rtio% THEN rtw.loop
LET w.rtio%=w.rtio%-1    :REM reduce wheel
GOTO w.rtio.ready
enlarge.wheel:
IF w.rtio%=max.w.rtio% THEN rtw.loop
LET w.rtio%=w.rtio%+1
w.rtio.ready:
GOSUB erase.wheel
GOSUB new.wheel
GOSUB show.wheel
GOTO rtw.loop
rtw.done:
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
RETURN
```

These lines function similarly to the corresponding lines from the track-size routine.

Changing the Cog Size

Pressing the STEP button activates the following lines, which print the instructions shown in Figure 16-4d.

```
set.step:
LET event%=MOUSE(0)   :REM clear previous mouse-clicks
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
CALL MOVETO(bx%(1)-30,12)
CALL TEXTFACE(1)
PRINT "CHANGE STEP SIZE"
CALL TEXTFACE(0)
PRINT
PRINT PTAB(bx%(1)-30); "Step size is determined"
PRINT PTAB(bx%(1)-30); "by invisible cogs on"
PRINT PTAB(bx%(1)-30); "the wheel circumference."
PRINT
PRINT PTAB(bx%(1)-30); "Click inside wheel"
PRINT PTAB(bx%(1)-30); "to change cog size."
PRINT
PRINT PTAB(bx%(1)-30); "Click outside wheel"
PRINT PTAB(bx%(1)-30); "to leave as-is."
```

The following lines monitor your changes to the step size (the size of the invisible cogs):

```
ss.loop:
WHILE MOUSE(0)<=0
WEND
LET outside.wheel%=FNdist(MOUSE(1),MOUSE(2),wc.x%,wc.y%)>w.rad%
IF outside.wheel% THEN ss.done:
LET cog.size%=(cog.size% MOD cog.scale%)+1
GOSUB erase.step
GOSUB new.step
GOSUB show.step
GOTO ss.loop
ss.done:
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
RETURN
```

Each time you click the mouse button inside the wheel, the cog size increases by 2**pi*/cog.scale% (in degrees, by 360/cog.scale%). Cog sizes that do not divide evenly into 2**pi* (360 degrees) have no real-world counterpart, but they occasionally produce interesting patterns.

The maximum cog size is 2**pi* (360). At this setting, the wheel makes a complete rotation before the program draws the next point on the curve.

Setting the Pen Location

Pressing the PEN button activates the following lines, which print the instructions shown in Figure 16-4c:

```
set.pen.radius:
LET event%=MOUSE(0)   :REM clear previous mouse-clicks
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
CALL MOVETO(bx%(1)-30,12)
CALL TEXTFACE(1)
PRINT "CHANGE PEN LOCATION"
CALL TEXTFACE(0)
PRINT
PRINT PTAB(bx%(1)-30); "Pen is the little dot"
PRINT PTAB(bx%(1)-30); "inside the wheel."
PRINT
PRINT PTAB(bx%(1)-30); "To move the pen,"
PRINT PTAB(bx%(1)-30); "point to destination"
PRINT PTAB(bx%(1)-30); "and click the mouse."
PRINT
PRINT PTAB(bx%(1)-30); "Destination must be"
PRINT PTAB(bx%(1)-30); "on or below the wheel's"
PRINT PTAB(bx%(1)-30); "centerpoint."
PRINT
PRINT PTAB(bx%(1)-30); "Click to the right"
PRINT PTAB(bx%(1)-30); "of the vertical line"
PRINT PTAB(bx%(1)-30); "to leave as-is."
```

The following lines monitor your requests for changes to the pen location:

```
spr.loop:
WHILE MOUSE(0)<=0
WEND
```

```
LET mx%=MOUSE(1)
LET my%=MOUSE(2)
IF mx%>max.t.dia% THEN spr.done
IF my%<wc.y% THEN spr.loop
IF my%>wc.y%+w.rad% THEN my%=wc.y%+w.rad%
IF my%=pen.y% THEN spr.loop
PUT (pen.x%-2,pen.y%-2)-(pen.x%+2,pen.y%+2),previous%,PSET
LET pen.y%=my%
LET pen.rad%=ABS(pen.y%-wc.y%)
LET p.rtio=pen.rad%/w.rad%
GET (pen.x%-2,pen.y%-2)-(pen.x%+2,pen.y%+2),previous%
CIRCLE (pen.x%,pen.y%),2
GOTO spr.loop
spr.done:
LINE (max.t.dia%+2,0)-(ww%,wl%),0,bf
RETURN
```

Calculating the Drawing Parameters

Each time you change one of the tool sizes or locations, the following lines make the necessary recalculations of all dependent values:

```
calculate.params:
LET t.rad%=max.t.rad%*t.rtio%/t.scale%
new.wheel:
LET w.rad%=t.rad%*w.rtio%/w.scale%
LET wc.y%=tc.y%-t.rad%+w.rad%+1
new.pen:
LET pen.rad%=INT(w.rad%*p.rtio)
LET pen.y%=wc.y%+pen.rad%
new.step:
LET arc.rad%=w.rad%*3\5
LET end.arc=pi*cog.size%/cog.scale%*2
LET step.ccf%=INT(pi*cog.size%/cog.scale%*2*w.rad%)
LET step.t.angle=step.ccf%/t.rad%
RETURN
```

Changing the track size causes a recalculation of all four values: track radius, wheel radius, pen radius (location), and step size (cog size). Changing one of the other parameters changes only those values that depend on the changed parameter, hence the separate entry points for new.wheel, new.pen, and new.step.

The following block erases the previous track, wheel, pen, and step indicator from the display:

```
erase.params:
CIRCLE (tc.x%,tc.y%),t.rad%,0
erase.wheel:
CIRCLE (wc.x%,wc.y%),w.rad%,0
erase.step:
CIRCLE (wc.x%,wc.y%),arc.rad%,0,pi/2,-(end.arc+pi/2)
erase.pen:
CIRCLE (pen.x%,pen.y%),2,0
RETURN
```

Again, erasing an object requires erasing all interior objects as well.

The next block draws the track, wheel, pen, and step size in their current sizes:

```
show.params:
CIRCLE (tc.x%,tc.y%),t.rad%
show.wheel:
CIRCLE (wc.x%,wc.y%),w.rad%
show.step:
CIRCLE (wc.x%,wc.y%),arc.rad%,1,pi/2,-(end.arc+pi/2)
show.pen:
GET (pen.x%-2,pen.y%-2)-(pen.x%+2,pen.y%+2),previous%
CIRCLE (pen.x%,pen.y%),2
RETURN
```

Test Point 1

You can now test the lines you've entered so far. First compare a listing of your program with the original program lines in this chapter. After correcting any typing errors, run the program. You should see the screen shown in Figure 16-3. Try using each of the first four buttons; you cannot use the DRAW button, since that logic isn't included yet.

Press TRACK and try making the track as small as possible, then as large as possible. Click to the right of the vertical line, then press the WHEEL button. Try to reset the wheel size.

Return to the main setup menu (click on the right of the vertical line) and try setting the pen location anywhere between the wheel's centerpoint and the circumference below the centerpoint.

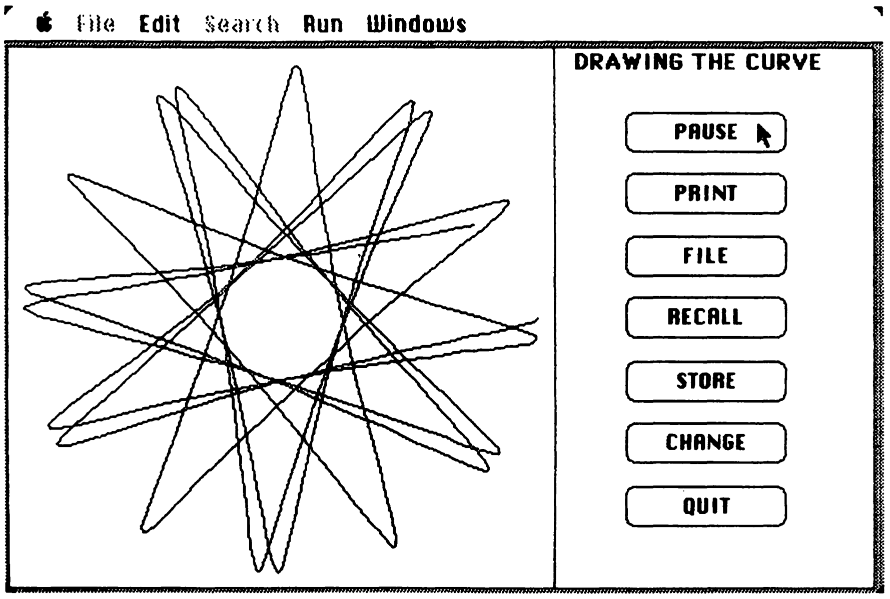


Figure 16-5. Drawing a curve

Return to the main setup menu and press the STEP button. Click the mouse button inside the wheel and the arc should advance counter-clockwise. If you continue to click inside the wheel, eventually the arc will appear as a straight line, representing the maximum step size (2**pi* or 360 degrees). Click again and the step size will start over at its minimum value.

Drawing the Curve

Now add the following lines to the end of the program. These lines produce a screen similar to that shown in Figure 16-5:

```
draw.curve:
CLS
LINE (max.t.dia%+1,0)-STEP(0,max.t.dia%)
CALL TEXTFACE(1)
PRINT PTAB(bx%(1)-30); "DRAWING THE CURVE"
CALL TEXTFACE(0)
LET t.circ%=INT(2*pi*t.rad%)
LET w.circ%=INT(2*pi*w.rad%)
LET u%=t.circ%
LET v%=w.circ%
WHILE v%>0
LET r%=u% MOD v%
LET u%=v%
LET v%=r%
WEND
LET lcm=t.circ%*w.circ%/u%
LET last.a=lcm/t.circ%
IF last.a=1 THEN last.a=2*pi
LET tw.dif%=t.rad%-w.rad%
LET tw.wr.quo=tw.dif%/w.rad%
LET paused%=no%
LET quit%=no%
LET change%=no%
LET curve.done%=no%
GOSUB set.up.buttons
DIALOG ON
ON DIALOG GOSUB dc.interrupt
LET a=0
GOSUB calculate.points
PSET(px%,py%)
```

T.circ% and w.circ% are the circumferences of the track and wheel. Lcm is the least common multiple of the two (the smallest number into which they both divide evenly). From lcm, the program determines last.a, the number of radians through which the wheel must rotate before the curve begins to repeat.

Pause%, quit%, change%, and curve.done% are status variables used to terminate the drawing procedure when you press PAUSE, QUIT, or CHANGE, or when the curve begins to repeat itself. The set.up.buttons subroutine prepares the buttons shown in Figure 16-5. Dc.interrupt handles button events during the drawing procedure.

The program starts by setting the angle to 0 and plotting the first point on the curve (PSET (px%,py%)).

The next block constitutes a repetitive procedure for calculating the next point on the curve and connecting it to the previously drawn point:

```
dc.loop:
WHILE paused% AND NOT change% AND NOT quit%
WEND
IF change% THEN dc.change
IF quit% THEN dc.quit
IF curve.done% THEN dc.loop
LET a=a+step.t.angle
GOSUB calculate.points
LINE -(px%,py%)
LET curve.done%=(a>=last.a)
IF NOT curve.done% THEN dc.loop
LET paused%=no%
BUTTON CLOSE 1
LINE (max.t.dia%+2,0)-(ww%,by%(1)-1),0,bf
CALL MOVETO(bx%(1)-30,12)
CALL TEXTFACE(1)
PRINT "CURVE IS COMPLETE"
CALL TEXTFACE(0)
GOTO dc.loop
dc.change:
DIALOG STOP
FOR b%=1 TO 7
BUTTON CLOSE b%
NEXT b%
GOTO change.params
dc.quit:
FOR b%=1 TO 7
BUTTON CLOSE b%
NEXT b%
END
```

If you press PAUSE, the program sets the paused% flag to yes% and changes the button label from PAUSE to CONTINUE. Under that condition, the program waits in the first WHILE/WEND loop until you press CONTINUE, CHANGE, or QUIT.

When paused%=no%, the program advances beyond this loop and calculates the next curve position (LET a=a+step.t.angle, GOSUB calculate.points, and LINE −(px%,py%)).

Here's the subroutine that calculates the coordinates of the new pen location for angle a:

```
calculate.points:
LET x=(tw.dif%)*COS(a)+pen.rad%*COS(tw.wr.quo*a)
LET y=(tw.dif%)*SIN(a)-pen.rad%*SIN(tw.wr.quo*a)
LET px%=INT(ABS(x)+.5)*SGN(x)+tc.x%
LET py%=INT(ABS(y)+.5)*SGN(y)+tc.y%
RETURN
```

The first two program lines correspond to parametric equations for a family of curves known as hypocycloids (the type of curves drawn by this program). The next two lines convert x and y to point addresses around a circle with center at tc.x%, tc.y%.

Interrupt Handlers

The next few program blocks take care of button interrupts that take place during the curve-drawing procedure. Pressing a button while the computer is drawing a curve (as shown in Figure 16-5) activates this routine:

```
dc.interrupt:
LET event%=DIALOG(0)
IF event%<>1 THEN RETURN
LET d.interrupt%=DIALOG(1)
ON d.interrupt% GOTO
        pause.switch,dc.print,dc.file,dc.recall,dc.store,dc.rq.change,dc.rq.quit
```

The routine transfers to the appropriate button routine, depending on which button was pressed. First we present the PAUSE and PRINT handlers:

```
pause.switch:
LET paused%=paused% XOR yes%
BUTTON 1,1,pause.btn$(-paused%),(bx%(1),by%(1))-(bx1%(1),by1%(1)),1
RETURN
dc.print:
CALL HIDECURSOR
LCOPY
CALL SHOWCURSOR
RETURN
```

For the PAUSE button, the program switches the status of the paused% flag and the label that appears on button 1. For the PRINT button, the program hides the cursor and copies the screen to the printer.

Here are the lines that handle the CHANGE, STORE, and RECALL buttons:

```
dc.rq.change:
LET change%=yes%
```

```
RETURN
dc.store:
GET (0,0)-(max.t.dia%,max.t.dia%),curve%
PSET(px%,py%)   :REM return to latest point on curve
RETURN
dc.recall:
PUT (0,0)-(max.t.dia%,max.t.dia%),curve%,OR
PSET(px%,py%) ' :REM return to latest point on curve
RETURN
dc.rq.quit:
LET quit%=yes%
RETURN
```

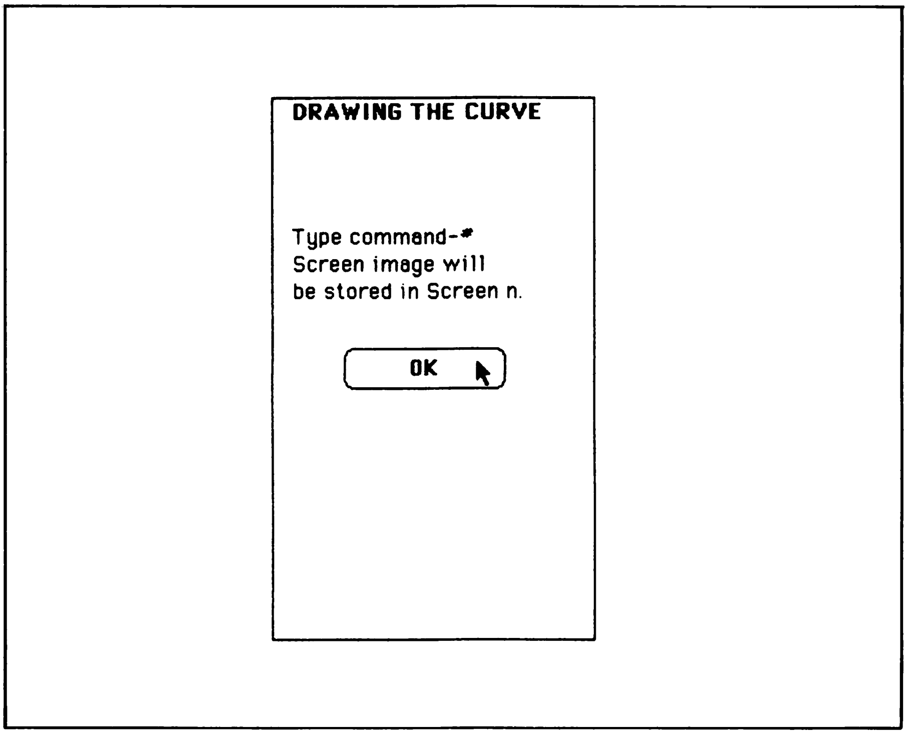
If you have a 128K Mac, the STORE and RECALL buttons will always be disabled (ghosted appearance).

If you have a 512K Mac and you press STORE, the program copies everything to the left of the vertical line into the array curve%( ). Pressing RECALL transfers the contents of curve%( ) back onto the screen, but doesn't erase any lines that are already drawn. That enables you to superimpose one curve on top of a previously drawn one.

The next lines handle the FILE button:

```
dc.file:
FOR b%=1 TO 7
BUTTON CLOSE b%
NEXT b%
BUTTON 1,1,"OK",(bx%(4),by%(4))-(bx1%(4),by1%(4)),1
CALL MOVETO(bx%(2)-30,by%(2)+12)
PRINT "Type command-*."
PRINT PTAB(bx%(2)-30); "Screen image will"
PRINT PTAB(bx%(2)-30); "be stored in Screen n."
WHILE DIALOG(0)<>1
WEND
BUTTON CLOSE 1
LINE (max.t.dia%+2,by%(1))-(ww%,w1%),0,bf
PSET(px%,py%)   :REM return to last point on the curve
GOSUB set.up.buttons
RETURN
```

The program prints the instructions shown in Figure 16-6 and waits for you to press the OK button. Note that to save a copy of the design,


DRAWING THE CURVE

Type command-*
Screen image will
be stored in Screen n.

Figure 16-6. Instructions for saving a copy of the screen

you must press COMMAND-SHIFT-3 (COMMAND-#) prior to clicking OK.

The final subroutine in the program sets up the buttons that are shown in Figure 16-5.

```
set.up.buttons:
IF NOT curve.done% THEN BUTTON
        1,1,"PAUSE",(bx%(1),by%(1))-(bx1%(1),by1%(1)),1
BUTTON 2,1,"PRINT",(bx%(2),by%(2))-(bx1%(2),by1%(2)),1
BUTTON 3,1,"FILE",(bx%(3),by%(3))-(bx1%(3),by1%(3)),1
BUTTON 4,SGN(mem.req%),"RECALL",(bx%(4),by%(4))-(bx1%(4),by1%(4)),1
BUTTON 5, SGN(mem.req%), "STORE",(bx%(5),by%(5))-(bx1%(5),by1%(5)),1
BUTTON 6, 1, "CHANGE",(bx%(6),by%(6))-(bx1%(6),by1%(6)),1
BUTTON 7, 1, "QUIT",(bx%(7),by%(7))-(bx1%(7),by1%(7)),1
RETURN
```

—Testing and Using the Program

After carefully checking your work, run the program. By experimenting with various combinations of wheel size, step size, and pen location, you should be able to get results similar to those shown in Figure 16-2.

To superimpose one drawing on top of another (512K Macs only), press STORE while the first curve is displayed. Then press CHANGE and reset the tool setup. Then draw the new curve. While the new curve is being drawn, press RECALL, and your previous curve will be superimposed on the current one.

To place one curve completely inside another one, reduce the track size to draw the smaller figure, then store it. You can then superimpose it on a larger figure using the RECALL button.

Chapter 17

Secret Messages

Cryptography, or the art of coding and deciphering messages, has been in use for almost 4000 years. Diplomats, soldiers, popes, and furtive lovers have all used it to send private messages through public channels. With this program, you can practice it just for fun.

The Secret Message Processor program presented in this chapter turns your Macintosh into a code machine. The program converts English or any other language into apparent gibberish, and vice versa. To decipher a secret message, you must know the code that was used to encipher the original message.

In a typical use of the program, you and a friend both have access to a Macintosh computer. The two of you agree on a *key* value prior to sending the message. You run the program, input the key value, and type in the original message (cryptographers call it the *plaintext*). The program outputs the enciphered version (the *ciphertext*) onto paper or into a disk file. You send your friend the printout or the disk.

When your friend receives the ciphertext, he repeats the process: running the program, inputting the identical key value, and typing in the ciphertext or loading it from the disk file. The program deciphers the message, outputting the results onto the screen and, optionally, onto paper or another disk file.

Figures 17-1 through 17-5 show various stages of the program's operation.

—Crash Course in Cryptography

We'll start this introduction to cryptography with a few definitions.

A cipher is a process that converts plaintext into ciphertext or vice versa. The two general categories of ciphers are transposition and substitution. Transposition ciphers rearrange the letters according to a definite set of rules. The resultant letter-frequency distribution (the number of times each letter occurs) remains the same; only the sequence is changed.

Substitution ciphers replace each letter of the plaintext with another letter by using a replacement table. The letter-frequency distribution is different in the plaintext and ciphertext, but the sequence of letters is the same—that is, the nth letter in the plaintext produces or corresponds to the nth letter in the ciphertext.

Table 17-1 shows examples of each type of cipher.

The cryptographic method employed by the Secret Message Processor is a form of substitution cipher.

The program has a list of 64 characters (the cipher list) that can be processed. Any characters that aren't in the list are left as is (not processed). Cipherable characters are uppercase and lowercase letters, the ten decimal digits, and the plus and minus signs.

Table 17-1. Transposition and Substitution Ciphers

TRANSPOSITION: Write down the message one line at a time, five columns to a line. Read off the ciphertext one column at a time.

```
T H E   N
E W   P A
S S W O R
D   I S
C R A B T
R E E . .
```

Plaintext: THE NEW PASSWORD IS CRABTREE.

Ciphertext: TESDCRHWS REE WIAE POSB.NAR T.

SUBSTITUTION: Replace each letter with its third successor in the alphabet:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
```

Plaintext: THE NEW PASSWORD IS CRABTREE.

Ciphertext: WKH QHZ SDVVZRUG LV FUDEWUHH.

The Secret Message Processor also has a list of numbers known as a *key stream.* Each cipherable character of the plaintext is paired with a number taken from the key stream. For each character-number pair, the program derives a corresponding ciphertext character, as illustrated:

Plaintext:	M	e	e	t	m	e	a	t	7	p	m
Key stream:	47	17	19	34	56	3	4	57	58	34	36
Ciphertext:	S	X	D	c	t	–	P	a	X	m	W

The Secret Message Processor can generate a very large number of different key streams, and each key stream produces a different ciphertext. To decipher a message, you must use the same key stream that was used to encipher it. The *key* or secret code number determines which key stream is used.

Enciphering a Character

Given a character-number pair *c-n,* the Secret Message Processor follows these steps to derive ciphertext character *d*:

1. Find the position of character *c* within the 64-character cipher list. By convention, the first position is numbered 0. Thus the position of character *c* (referred to as p(*c*) is a value from 0 to 63.
2. Take the number *n* (also a value between 0 and 63) and calculate *n* XOR p(*c*). (The XOR operation is explained below.) The result is a value from 0 to 63, referred to as p(*d*).
3. Locate the character within the cipher list at position p(*d*). Call that character *d*; *d* is the ciphertext character corresponding to plaintext character *c*.

The XOR Operator

XOR is a binary logical operator. Given two numbers A and B, XOR compares their binary representations one bit at a time to produce a result C. The outcome of each bit-to-bit comparison determines the value of the corresponding bit in the result C. Here are the rules for comparing bits from A and B:

A:	0	0	1	1
B:	0	1	0	1
A XOR B:	0	1	1	0

As an example, 174 XOR 119 = 217, as explained below:

$$\begin{array}{r} 174 = 10101110 \text{ binary} \\ \text{XOR } 119 = 01110111 \text{ binary} \\ \hline 217 = 11011001 \text{ binary} \end{array}$$

XOR has a special property: if C=A XOR B, then A=C XOR B and B=C XOR A. For instance:

$$\begin{array}{r} 174 = 10101110 \text{ binary} \\ \text{XOR } 217 = 11011001 \text{ binary} \\ \hline 119 = 01110111 \text{ binary} \end{array}$$

In short, the same function that generates C can be used to regenerate either of the original operands when the other operand is known. That's why the Secret Message Processor is able to encipher or decipher a message using the identical key and the same program logic.

The following two equations summarize the ciphering process used by the Secret Message Processor:

To encipher c:

$$\mathrm{p}(d)=\mathrm{p}(c) \text{ XOR } n$$

To decipher *d*:

$$\mathrm{p}(c)=\mathrm{p}(d) \text{ XOR } n$$

The variable *n* is the number from the key stream. P(*c*) refers to the position of character c in the cipher list. Knowing p(*c*), you can find the corresponding character *c*, and vice versa.

Source of the Key Stream

To encipher or decipher a message containing *m* characters, you need a list (key stream) of at least *m* random numbers. When this method is used manually, both parties (sender and receiver) have a printed copy of the key stream. They may even have a book of key streams and a prior agreement about which key stream to use on a given day.

The key stream we'll use is built right into the Macintosh. It's more commonly known as the random number generator, or the RND function in BASIC.

The RND function returns a pseudo-random value greater than or equal to 0 and less than 1. The value is not really random; it is determined by a "seed" value hidden in the Macintosh's memory. Each time the Macintosh executes the RND function, the seed value changes, so that the next time RND is used, it generates a different value. After a very large number of uses, RND completes the sequence and starts over.

Our key stream must consist of numbers between 0 and 63. To scale the result of RND into the range 0-63, we multiply by 64 and take the integer portion of the result.

We must also be able to generate the same sequence of numbers for enciphering and deciphering. To do this, we use the RANDOMIZE function, which sets the random number seed and thus determines the sequence of numbers that will be produced by subsequent uses of RND. For instance, the function RANDOMIZE 4 starts the following sequence: .2761034369468689, .8757113218307495, .7579789757728577.... To scale these values into the 0-63 range, we multiply by 64 and take the integer portion of the result, getting this sequence: 56, 32, 25....

—The Program

The first block describes the windows, buttons, and the edit field used in the program:

```
REM Window descriptors
DATA 4
DATA 6.5, 4.15, 0.25, 0.375
DATA 6.5, 0.75, 0.25, 0.375
DATA 3.2, 3.125,0.25, 1.5
DATA 3.2, 3.125, 3.55, 1.5
REM Button descriptors
DATA 9
DATA Keyboard, 1.2, 0.208, 0.4, 0.333, 3
DATA Disk file, 1.2, 0.208, 0.65, 0.333, 3
DATA Screen, 1.2, 0.208, 0.4, 0.5, 3
DATA Printer, 1.2, 0.208, 0.65, 0.5, 3
DATA Disk file, 1.2, 0.208, 0.9, 0.5, 3
DATA BEGIN, 1.0, 0.333, 0.333, 0.917, 1
DATA QUIT, 1.0, 0.333, 0.667, 0.917, 1
DATA PAUSE, 1.0, 0.333, 0.333, 0.875, 1
DATA STOP, 1.0, 0.333, 0.667, 0.875, 1
```

```
REM Edit field descriptors
DATA 0.75, 0.208, 0.8, 0.667
```

The second block reads in the window descriptors:

```
READ nw%
DIM ww%(nw%),wl%(nw%),wx%(nw%),wy%(nw%),wx1%(nw%),wy1%(nw%)
FOR n%=1 TO nw%
READ inches.wide, inches.long, ulcx, ulcy
LET ww%(n%)=inches.wide*72
LET wl%(n%)=inches.long*72
LET wx%(n%)=ulcx*72
LET wy%(n%)=ulcy*72
LET wx1%(n%)=wx%(n%)+ww%(n%)
LET wy1%(n%)=wy%(n%)+wl%(n%)
NEXT n%
```

The third block reads in the button descriptors.

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
IF n%<8 THEN LET w%=1 ELSE LET w%=2
READ bl$(n%),inches.wide, inches.long, hzone, vzone, bt%(n%)
LET bx%(n%)=(ww%(w%)-inches.wide*72)*hzone
LET by%(n%)=(wl%(w%)-inches.long*72)*vzone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
DIM pause$(1)
LET pause$(0)="PAUSE"
LET pause$(1)="CONTINUE"
```

And the fourth block reads in the edit field descriptors.

```
READ inches.wide, inches.long, hzone, vzone
LET fx%=(ww%(1)-inches.wide*72)*hzone
LET fy%=(wl%(1)-inches.long*72)*vzone
LET fx1%=fx%+inches.wide*72
LET fy1%=fy%+inches.long*72
```

Program Constants

The following lines set up certain constants and initial values for parameters:

```
LET yes%=(1=1)
LET no%=(1=0)
LET lower.case$="abcdefghijklmnopqrstuvwxyz"
LET upper.case$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LET numbers$="+-0123456789"
LET char.set$=numbers$+upper.case$+lower.case$
IF LEN(char.set$)<>64 THEN CLS: PRINT "Invalid character table.": STOP
LET source%=1    :REM keyboard
LET destination%=1    :REM screen
LET secret%=0    :REM initial key value
```

Char.set$ contains the cipher list. It is important to type the values for lowercase$, uppercase$, and number$ exactly as shown. The program checks to ensure that the list contains 64 characters.

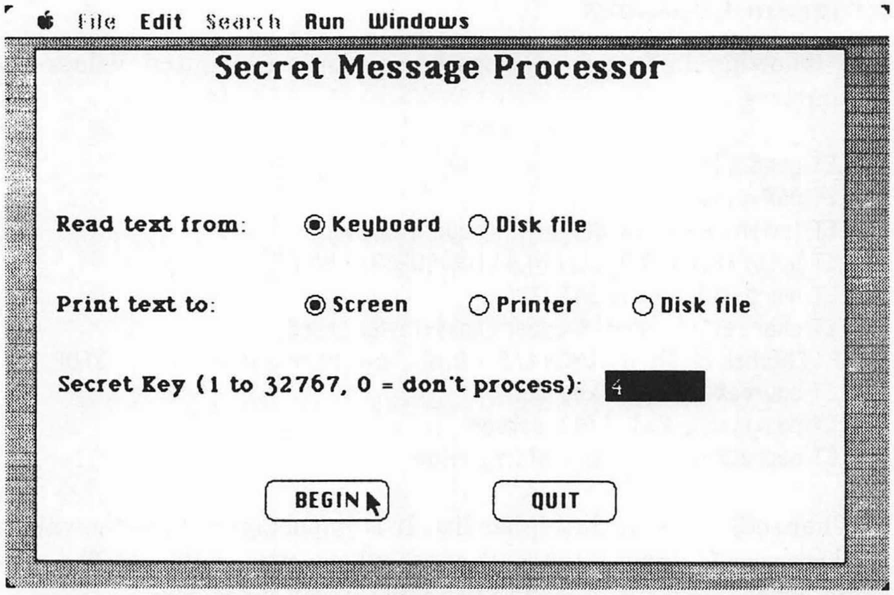
Main Menu

The next lines create the main menu window and print the title as shown in Figure 17-1.

```
main.menu:
WINDOW 1,,(wx%(1),wy%(1))-(wx1%(1),wy1%(1)),3
WIDTH 255
CALL TEXTFONT(2)
CALL TEXTSIZE(18)
CALL TEXTFACE(1)
LET title$="Secret Message Processor"
LET title.w%=WIDTH(title$)
CLS
PRINT PTAB((ww%(1)-title.w%)/2); title$
```

The preceding lines print the title only; the following lines put in the buttons and other information:

```
CALL TEXTSIZE(12)
CALL MOVETO(12,by%(1)+12)
PRINT "Read text from:"
```



Secret Message Processor

Read text from: [✓ Keyboard] Disk file

Print text to: [✓ Screen] Printer Disk file

Secret Key (1 to 32767, 0 = don't process): 4

Figure 17-1. The title screen and main menu

```
FOR n%=1 TO 2
BUTTON n%,1-(n%=source%), bl$(n%), (bx%(n%), by%(n%))- (bx1%(n%),
      by1%(n%)),bt%(n%)
NEXT n%
CALL MOVETO(12,by%(3)+12)
PRINT "Print text to:"
FOR n%=3 TO 5
BUTTON n%,1-(n%-2=destination%), bl$(n%), (bx%(n%), by%(n%))- (bx1%(n%),
      by1%(n%)),bt%(n%)
NEXT n%
FOR n%=6 TO 7
BUTTON n%,1, bl$(n%), (bx%(n%), by%(n%))- (bx1%(n%), by1%(n%)),bt%(n%)
NEXT n%
CALL MOVETO (12,fy%+12)
PRINT "Secret Key (0 to 32767, 0 = don't process):"
CALL TEXTFACE(0)
EDIT FIELD 1,STR$(secret%),(fx%,fy%)-(fx1%,fy1%)
```

The five radio-style buttons (buttons 1 through 5) indicate the current input/output device settings.

The program executes the following lines repeatedly until you select a button:

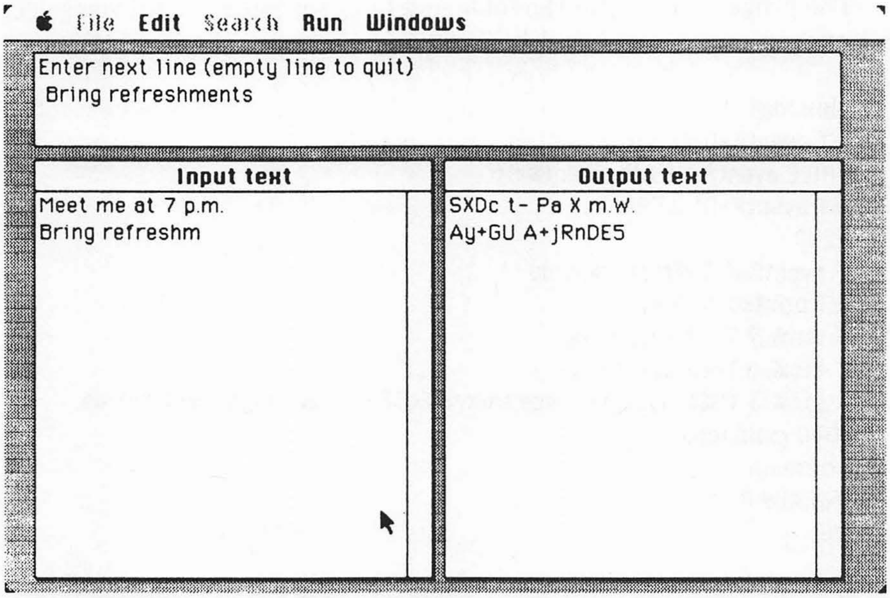
```
main.loop:
LET event%=DIALOG(0)
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
IF event%=6 THEN check.code
LET btn%=DIALOG(1)
IF btn%=7 THEN exit.main
IF btn%=6 THEN check.code
IF btn%<3 THEN GOSUB change.source ELSE GOSUB change.destination
GOTO main.loop
exit.main:
WINDOW CLOSE 1
END
```

Pressing the ENTER or RETURN key or clicking the BEGIN dialog button signals to the program that you are ready to begin the ciphering process. In that case, the program first checks the secret key to ensure that it is within range (check.code routine). Clicking one of buttons 1 through 5 signals a change to the input or output setting; the following lines make the change:

```
change.source:
BUTTON source%,1
LET source%=btn%
BUTTON source%,2
RETURN
change.destination:
BUTTON destination%+2,1
LET destination%=btn%-2
BUTTON destination%+2,2
RETURN
```

Here are the lines that ensure you have entered a valid secret key:

```
check.code:
LET x=VAL(EDIT$(1))
IF x<>INT(x) OR x< 0 OR x> +32767 THEN BEEP: GOTO main.loop
LET secret%=x
```



```
Enter next line (empty line to quit)
 Bring refreshments
```

```
Meet me at 7 p.m.
Bring refreshm
```

```
SXDc t- Pa X m.W.
Ay+GU A+jRnDE5
```

Figure 17-2. Enciphering text with key=4

```
LET xyz=RND(-secret%)    :REM Select pseudorandom sequence
WINDOW CLOSE 1
```

Setting Up the Coding Windows

The following lines initialize the three windows shown in Figure 17-2:

```
begin.coding:
LET termination.code%=0    :REM no error
WINDOW 1,,(wx%(2),wy%(2))-(wx1%(2),wy1%(2)),3
WINDOW 2,"Input text",(wx%(3),wy%(3))-(wx1%(3),wy1%(3)),1
WINDOW 3,"Output text",(wx%(4),wy%(4))-(wx1%(4),wy1%(4)),1
WINDOW 1
WIDTH 255
IF source%=1 THEN source.ok
```

Termination.code% keeps track of why the program stopped processing text; reasons include input and output file errors and clicking the QUIT button. Termination.code%=0 indicates no error.

Window 1 is a status indicator and dialog box that allows you to quit or pause the program. Window 2 repeats the text as it is typed from the keyboard or input from a disk file; window 3 shows the processed text (enciphered or deciphered).

The next lines prompt you to specify the name of a disk file for input to the program.

```
CLS
CALL TEXTFACE(1)
PRINT "READ TEXT FROM A DISK FILE"
CALL TEXTFACE(0)
LET fi$=FILES$(1,"TEXT")
IF fi$=nu$ THEN LET termination.code%=3: GOTO exit.coding
ON ERROR GOTO fi.err
OPEN fi$ FOR INPUT AS 1
ON ERROR GOTO 0
```

These lines set up the output device (display, printer, or disk file):

```
source.ok:
IF destination%=1 THEN destination.ok
IF destination%=2 THEN set.up.printer
CLS
CALL TEXTFACE(1)
PRINT "OUTPUT PROCESSED TEXT TO A DISK FILE"
CALL TEXTFACE(0)
LET fo$=FILES$(0,"Name the output file:")   :REM set up disk file
IF fo$=nu$ THEN LET termination.code%=3: GOTO exit.coding
ON ERROR GOTO fo.err
OPEN fo$ FOR OUTPUT AS 2
ON ERROR GOTO 0
GOTO destination.ok
set.up.printer:
OPEN "LPT1:DIRECT" FOR OUTPUT AS 2
```

Once all the input/output routing has been taken care of, the following lines put PAUSE and QUIT buttons into window 1 (except when you are inputting from the keyboard, when no buttons are needed). During disk input, the coding.interrupt routine is activated to handle requests to pause or quit the program.

```
destination.ok:
IF source%=1 THEN coding.loop
```

```
CLS
PRINT "Coding in progress"
FOR b%=8 TO 9
BUTTON b%-7,1,bl$(b%),(bx%(b%),by%(b%))-(bx1%(b%),by1%(b%)),bt%(b%)
NEXT b%
LET paused%=no%
LET rq.quit%=no%
ON DIALOG GOSUB coding.interrupt
DIALOG ON
```

The Coding Process

The following lines are executed repeatedly until there is no more text in the file (or you type an empty line during keyboard input) or until an input/output error occurs:

```
coding.loop:
WHILE paused% AND NOT rq.quit%
WEND
IF rq.quit% THEN exit.coding
IF source%=2 THEN disk.input
WINDOW OUTPUT 1
WIDTH 255
CLS
PRINT "Enter next line (empty line to quit)"
LINE INPUT text$
IF text$=nu$ THEN exit.coding
GOTO process.text
disk.input:
IF EOF(1) THEN exit.coding
ON ERROR GOTO fi.err
LINE INPUT#1, text$
ON ERROR GOTO 0
```

In the case of keyboard input, entering an empty line signals that no more text remains. In the case of disk input, the process continues until it reaches the end of file or an error occurs.

Given a line of text text$, the following routine reads every character and enciphers those that are contained in its cipher list.

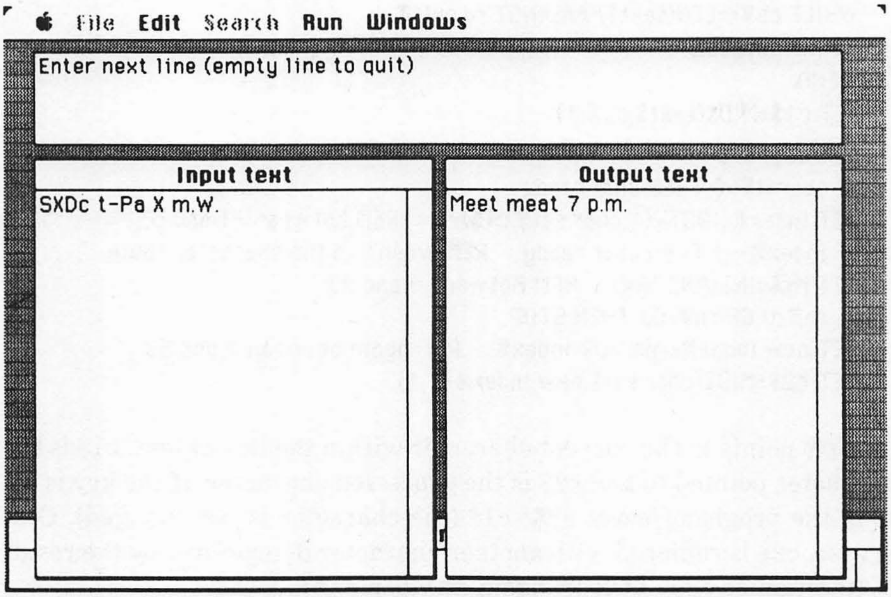
```
process.text:
LET cp%=1
```

```
WHILE cp%<=LEN(text$) AND NOT rq.quit%
WHILE paused%
WEND
LET c1$=MID$(text$,cp%,1)
LET c2$=c1$
IF secret%=0 THEN char.ready
LET index%=INSTR(1,char.set$,c1$)-1    :REM between -1 and 63
IF index%=-1 THEN char.ready   :REM wasn't in the character table
LET rn%=INT(RND*64)    :REM Between 0 and 63
IF rn%<0 OR rn%>63 THEN STOP
LET new.index%=rn% XOR index%    :REM again between 0 and 63
LET c2$=MID$(char.set$,new.index%+1,1)
```

Cp% points to the current character within the line of text. c1$ is the character pointed to and c2$ is the processed character. If the key is set to 0, the program leaves c2$=c1$ (the character is not changed). Otherwise, c2$ is replaced with another character determined by the result of (position of c2$) XOR (random number rn%).

When c2$ is ready, the following lines print the original character c1$ to window 2 and the processed character c2$ to window 3 (see Figure 17-3) and optionally to the printer or disk file:

```
char.ready:
WINDOW OUTPUT 2
WIDTH 25
PRINT c1$;
WINDOW OUTPUT 3
PRINT c2$;
ON ERROR GOTO fo.err
IF destination%<>1 THEN PRINT*2, c2$;
ON ERROR GOTO 0
LET cp%=cp%+1
WEND
WINDOW OUTPUT 2
WIDTH 25
PRINT
WINDOW OUTPUT 3
PRINT
ON ERROR GOTO fo.err
IF destination%<>1 THEN PRINT*2,
ON ERROR GOTO 0
GOTO coding.loop
```



Enter next line (empty line to quit)

SXDc t-Pa X m.W.

Meet meat 7 p.m.

Figure 17-3. Deciphering text with key=4

Windows 2 and 3 are set up with a maximum character width of 25, so that all the text will be displayed within the windows (WIDTH 25).

While the program is reading text from a disk file, the following lines handle button interrupts (requests to pause or quit):

```
coding.interrupt:
LET event%=DIALOG(0)
IF event%<>1 THEN RETURN
LET type.interrupt%=DIALOG(1)
ON type.interrupt% GOTO pause.switch,request.quit
pause.switch:
LET paused%=paused% XOR yes%
LET current.window%=WINDOW(1)   :REM remember current output window
WINDOW 1
BUTTON 1,1,pause$(-paused%),(bx%(8),by%(8))-(bx1%(8),by1%(8)),1
WINDOW OUTPUT current.window%   :REM restore current output window
RETURN
request.quit:
LET rq.quit%=yes%
RETURN
```

In case of a pause request, the program changes the PAUSE button to a CONTINUE button, sets the paused% flag to yes%, and returns. If the program is already paused, pressing CONTINUE changes the button from CONTINUE to PAUSE and sets the paused% flag to no%.

In case of a quit request, the program sets the rq.quit% flag to yes% and returns.

The next lines take over when coding is terminated for any reason:

```
exit.coding:
ON ERROR GOTO 0
DIALOG STOP
CLOSE
WINDOW 1
WIDTH 255
CLS
IF termination.code%>0 THEN input.output.error
IF rq.quit% THEN PRINT "Processing terminated." ELSE PRINT "Processing
      complete."
GOTO wait.ok
```

Termination.code%>0 indicates that an input or output error caused the processing to stop; otherwise the processing ended because you pressed the QUIT button or the program reached the end of the text. The program prints an appropriate message on the screen depending on which condition is true.

The following lines warn you that an error occurred:

```
input.output.error:
BEEP
ON termination.code% GOTO input.error,output.error,io.cancelled
input.error:
PRINT "Error while reading from:"
PRINT fi$
GOTO wait.ok
output.error:
PRINT "Error while outputting to:"
PRINT fo$
GOTO wait.ok
io.cancelled:
PRINT "Cancelled processing."
```

Before returning to the main menu, the program creates an OK button and waits for you to press it.

```
wait.ok:
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON 1,1,"OK",(bx%(9),by%(9))-(bx1%(9),by1%(9)),bt%(9)
LET event%=DIALOG(0)
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
FOR w%=1 TO 3
WINDOW CLOSE w%
NEXT w%
GOTO main.menu
```

The last program block is executed when an error occurs:

```
fi.err:
LET termination.code%=1
RESUME exit.coding
fo.err:
LET termination.code%=2
RESUME exit.coding
```

The fi.err routine is enabled immediately before and disabled immediately after each file input operation. The same is true of the fo.err routine with respect to file output operations.

—Testing and Using the Program

First carefully check a printout of the program against the program lines in this chapter. Then run the program. Set the key to 0, specify input from the keyboard and output to the screen. Then press BEGIN. The program should prompt you to enter text, as shown in Figure 17-1. Since you selected a key of 0, the program should not change your text—the Input Text (plaintext) should be the same as the Output Text (ciphertext).

To stop entering text, press ENTER on an empty line. Then select a key of 4 and repeat the test. Now type the same text that is shown in Figure 17-2. You should get the same results that are shown in that figure.

To test the deciphering capability, go back to the main menu (press ENTER on an empty line), leave the key set to 4, and press BEGIN again.

Now type in the ciphertext as shown in Figure 17-3. The original plaintext should be printed in the Output Text window.

Test the program's ability to output to the printer and to a disk file, and to input text from a file (must be a text file). Figures 17-4 and 17-5 show the screen appearance after selection of disk input/output from the main menu.

—Tips for Processing Lengthy Texts

If you are enciphering or deciphering a lengthy text, you may not want to sit at the keyboard waiting for the computer to process one line at a time. Using the disk-to-disk option (input from one disk file, output to another) can free you to do other things while the computer processes the entire text.

Suppose you want to send a lengthy document to a friend. Run the program, specifying the keyboard as the input device and a disk file PLAINTEXT as the output device. Enter a key of 0 (no processing).

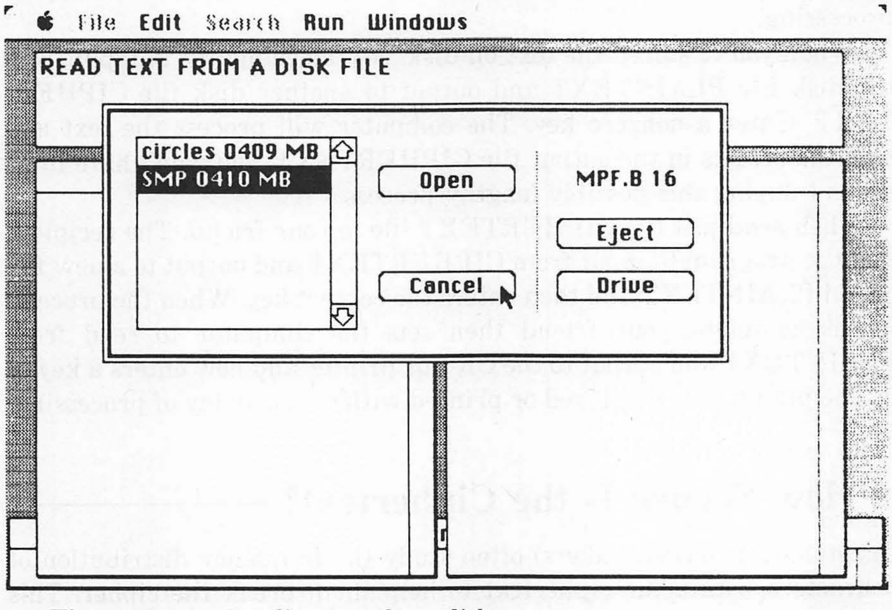


Figure 17-4. Loading text from disk

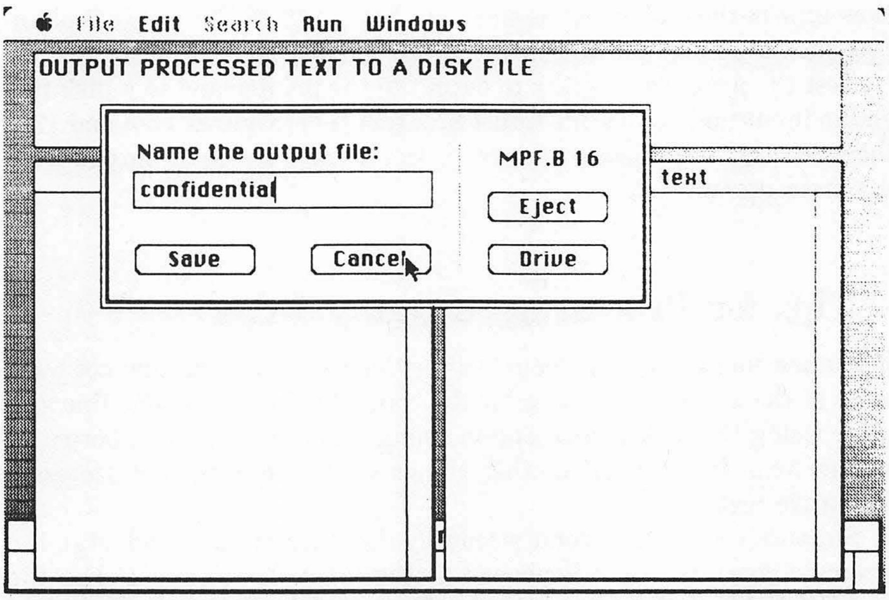


Figure 17-5. Saving text to disk

Type in the text, which will be stored on disk without the delay of processing.

When you've stored the text on disk, set the computer to input from the disk file PLAINTEXT and output to another disk file CIPHERTEXT. Enter a nonzero key. The computer will process the text and save the results in the output file CIPHERTEXT; you won't have to be around during this possibly lengthy process.

Then send just the CIPHERTEXT file to your friend. The recipient sets the program to input from CIPHERTEXT and output to a new file called PLAINTEXT and then enters the correct key. When the processing is complete, your friend then sets the computer to read from PLAINTEXT and output to the CRT or printer and now enters a key of 0. The plaintext is displayed or printed without the delay of processing.

—How Secure Is the Ciphertext?

Cryptanalysts (codebreakers) often study the frequency distribution of characters within the ciphertext to help them break the cipher. This technique is of little use with ciphertext from the Secret Message pro-

Table 17-2. Frequency Distribution of Characters in the Ciphertext

Plaintext	Key	Ciphertext
AAAAAAAAA	32050	F1soJBEuM
111111111	12345	IcLPJQKOn
Joe Joe Joe	41200	7Ge −91 w7i

gram because the distribution of letters in its ciphertext is almost uniform. (See Table 17-2.)

The very fact of uniform frequency distribution might lead a cryptanalyst to suspect the use of a key stream substitution cipher. However, breaking such a cipher is difficult and time-consuming.

If a cryptanalyst can obtain a large sample of ciphertext, he may eventually break the code. The cryptanalyst starts by assuming that certain words occur in the text ("the," for example) and then applies various mathematical operations to the ciphertext, trying to obtain "the." Once he has recovered a single word of plaintext, he may be able to infer the nature of the key stream since it is not truly random, only pseudo-random. (If it were a truly random key stream, the cipher would be virtually unbreakable without prior knowledge of the key stream.)

The only way for a person who is not a cryptanalyst to break the code is by trial and error, assuming the person has a copy of the Secret Messages program. This time-consuming method requires the would-be codebreaker systematically to try different keys and see the results on the ciphertext.

In summary, the Secret Messages program produces ciphertext that is secure against attack by nonexperts. However, don't expect it to fool the National Security Administration!

This chapter has been adapted from "Secret Messages" by George Stewart, appearing in the April 1983 issue of *Popular Computing* magazine. Copyright © 1983 BYTE Publications, Inc. Used with the permission of BYTE Publications, Inc.

Chapter 18

Blazing Telephones

Harry was plain old 273-2255 until he found out about *ape-call.* Sue suffered along with 468-5477 until she discovered *hot-lips.* And Frank never really appreciated his 683-4323 until he noticed *mud-head.*

How about your telephone number? Would you like to add a little "ring" to it? The Blazing Telephones program will help you find out what words (if any) are hidden in those seven digits.

The technique of replacing digits with letters is often used by businesses. A barbecue stand, for example, may ask the local telephone company for the number 737-3744 (*pure pig*) or 255-2333 (*all beef*), depending on its culinary persuasion. Although telephone companies are not obligated to honor such requests, most of them will try to do so if it is possible.

The situation facing the private individual is less encouraging. The telephone company cannot comply with all personal requests for a specific number. Furthermore, you probably already have a telephone number that is widely known by friends and associates.

But serendipity is on your side. By conducting an exhaustive search through all 2187 possible letter combinations, chances are good that you'll find a viable alternative to the plain numeric sequence. But exhaustive searches tend to be exhausting. That's where Blazing Telephones comes in.

—The Method

Any person who uses a phone will recognize the two objects portrayed in Figure 18-1. They are reproduced here to emphasize the correspondence between the digits 0-9 and the letters A-P and R-Y (the letters Q and Z are omitted on the dials).

For each digit in your phone number, three different letter replacements are possible. The numbers 0 and 1 are exceptions: the telephone dial offers no replacements for them. Thus, for a seven-digit number, the total number of distinct letter combinations is 3^7 or 2187, and fewer if the number includes 1's or 0's.

The combinatorial problem is solved by a simple exercise in counting. The trick is to count in base 3. All base 3 numbers are composed of three distinct symbols: 0, 1, and 2. For example, the decimal or base 10 number 19 is represented in base 3 as 201 ($2 \times 3^2 + 0 \times 3^1 + 1 \times 3^0$).

For seven-digit telephone numbers, the program counts from 0 to 2186 in base 3. (If your telephone number contains more or fewer than seven digits, the program automatically adjusts the base 3 counter to match the number of possibilities for that number.) Each base 3 number acts as a mask or key for generating the 2187 possible alphabetic sequences.

Consider the phone number 352-5562. The first digit is a 3. According to the telephone dial layout, 3 corresponds to the letter triplet D,E,F.

Which letter is chosen? Here's where the key comes in. Each digit of the key is either 0, 1, or 2. In the case of a 0, the first letter in the triplet is used; in the case of a 1, the second letter; and in the case of a 2, the third letter is used.

The first base 3 number generated is 0000000 (seven digits are required since the phone number contains seven digits). The first digit in the key is 0, so D is taken, which is the "0th" letter in the triplet D,E,F. The second digit in the phone number is a 5, which corresponds to the triplet J,K,L. The key has a 0 in the second position, so the 0th letter, J, is selected.

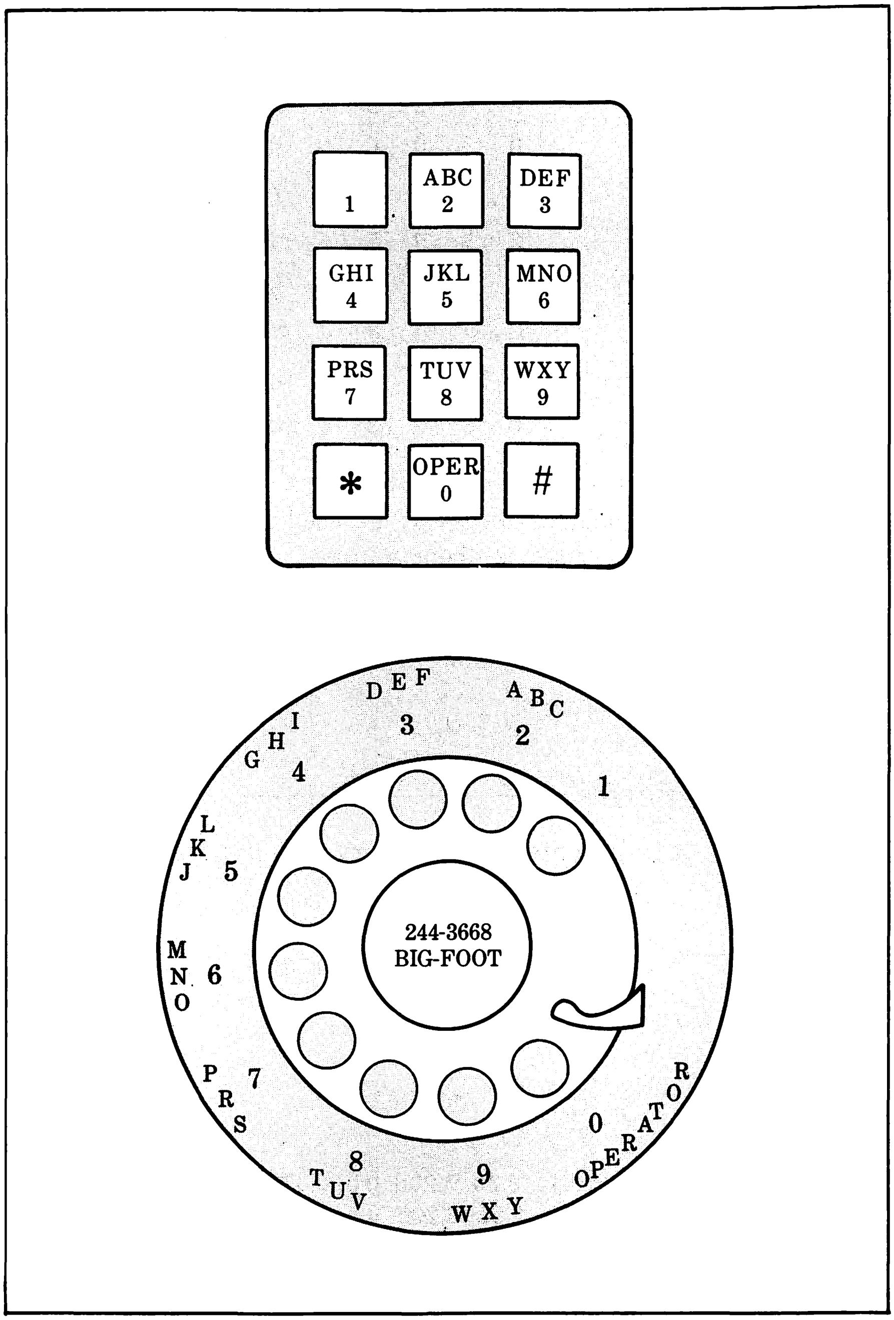


Figure 18-1. Pushbutton and rotary dial telephone faces

The following table shows letter replacements for the phone number 352-5562 using the three keys 0000000, 0000001, and 0002100:

Phone number:	3	5	2	5	5	6	2
Key:	0	0	0	0	0	0	0
Letter sequence:	D	J	A	J	J	M	A
Phone number:	3	5	2	5	5	6	2
Key:	0	0	0	0	0	0	1
Letter sequence:	D	J	A	J	J	M	B
Phone number:	3	5	2	5	5	6	2
Key:	0	0	0	2	1	0	0
Letter sequence:	D	J	A	L	K	O	A

In a similar manner, all 2187 keys can be used to generate a total of 2187 distinct names for this one phone number.

To be sure you understand the method, compute the resultant letter sequence for the phone number 266-7883 and the key 2020101.

Figures 18-2 through 18-9 show the screen appearance when you run the program.

—The Program

The first block of lines contains descriptors for the windows, buttons, and edit fields:

```
REM Windows
DATA 3
DATA 4, 3, 1.5, 0.5
DATA 4,1,1.5,0.375
DATA 7,3,0.05,1.5
REM Buttons
DATA 12
DATA Alphabetic to numeric, 2.25, 0.208, 0.5, 0.375, 2
DATA Numeric to alphabetic, 2.25, 0.208, 0.5, 0.5, 2
DATA OK, 0.75, 0.333, 0.333, 0.875, 1
DATA QUIT, 0.75, 0.333, 0.667, 0.875, 1
DATA AGAIN, 1, 0.333, 0.25, 0.9, 1
DATA MENU, 1, 0.333, 0.75, 0.9, 1
DATA Screen, 1, 0.208, 0.5, 0.25, 3
DATA Printer, 1, 0.208, 0.5, 0.375, 3
DATA Disk File, 1, 0.208, 0.5, 0.5,3
DATA PAUSE, 1.0, 0.333, 0.25, 0.75, 1
DATA STOP, 1.0, 0.333, 0.75, 0.75,1
DATA OK,1,0.333, 0.5, 0.75, 1
REM Edit fields
DATA 2.5, 0.208, 0.5, 0.4
```

The first window is shown in Figure 18-2; the second and third windows, in Figure 18-7; buttons are shown in Figures 18-2, 18-4, 18-6, 18-7, 18-8, and 18-9; and edit fields in Figures 18-3 and 18-5.

Next come three blocks of lines to read in the preceding data. The first block reads in the window data:

```
READ nw%
DIM ww%(nw%),wl%(nw%),wx%(nw%),wy%(nw%),wx1%(nw%),wy1%(nw%)
FOR n%=1 TO nw%
READ inches.wide, inches.long, ulcx, ulcy
LET ww%(n%)=inches.wide*72
LET wl%(n%)=inches.long*72
LET wx%(n%)=ulcx*72
LET wy%(n%)=ulcy*72
LET wx1%(n%)=wx%(n%)+ww%(n%)
LET wy1%(n%)=wy%(n%)+wl%(n%)
NEXT n%
```

The second block reads in the data for the buttons:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
IF n%<10 THEN LET w%=1 ELSE LET w%=2
READ bl$(n%),inches.wide, inches.long, hzone, vzone, bt%(n%)
LET bx%(n%)=(ww%(w%)-inches.wide*72)*hzone
LET by%(n%)=(wl%(w%)-inches.long*72)*vzone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
DIM pause$(1)
LET pause$(0)="PAUSE"
LET pause$(1)="CONTINUE"
```

Buttons 1 through 9 appear in window 1; buttons 10 and 12 appear in window 2 (IF n%<10 THEN LET w%=1 ELSE LET w%=2).

The following lines read in the field descriptors and set up certain arrays and constants:

```
READ inches.wide, inches.long, hzone, vzone
LET fx%=(ww%(1)-inches.wide*72)*hzone
LET fy%=(wl%(1)-inches.long*72)*vzone
LET fx1%=fx%+inches.wide*72
LET fy1%=fy%+inches.long*72
LET md%=15  :REM maximum digits in a phone number
DIM k%(md%),fo$(3)
LET fo$(1)="SCRN:"
LET fo$(2)="LPT1:DIRECT"
LET fo$(3)="DISK"
LET p$="000111ABCDEFGHIJKLMNOPRSTUVWXY"  :REM Alpha table
LET num$="0123456789"  :REM Numeric table
LET col.spaces%=2  :REM spaces between columns
LET yes%=(1=1)
LET no%=(1=0)
LET cv.type%=1  :REM 1=alpha to numeric, 2=numeric to alpha
LET output.to%=1  :REM 1=screen, 2=printer, 3=disk
LET lw%=70  :REM maximum characters per line
```

Array k%( ) holds the key (described previously). Fo$( ) holds the name of the output device used for printing the sometimes voluminous list of names.

The variable p$ is a table of the 30 characters used to replace the numerals. Note that there are 3 zeros and 3 ones followed by the complete alphabet *except* for the letters Q and Z.

Num$ is the corresponding table of digits used to convert letters back into numbers (alphabetic to numeric option).

Main Menu

The next block of lines creates the screen shown in Figure 18-2:

```
main.menu:
WINDOW 1,,(wx%(1),wy%(1))-(wx1%(1),wy1%(1)),3
CALL TEXTFONT(1)
CALL TEXTSIZE(18)
```

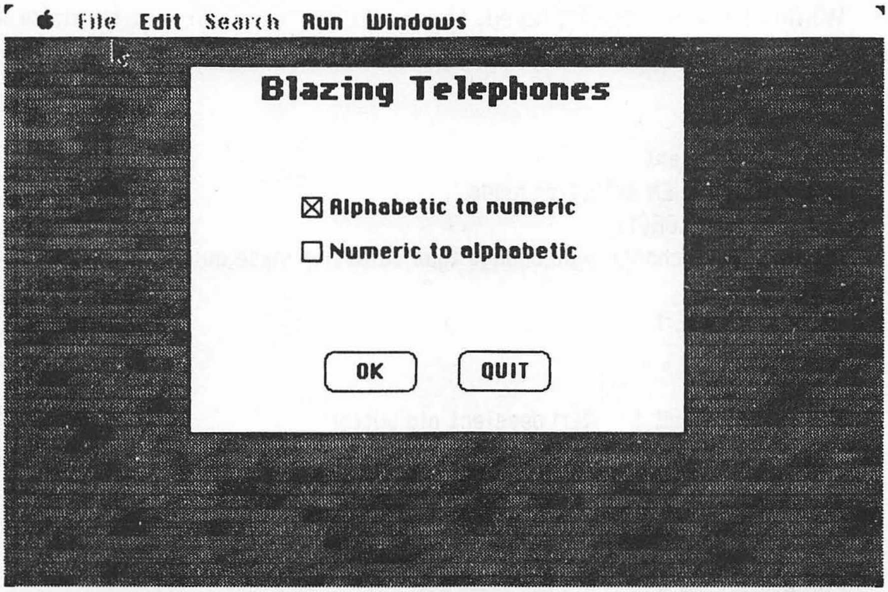


Figure 18-2. Main menu

```
CALL TEXTFACE(1)
CLS
LET title$="Blazing Telephones"
LET ti.tab%=(ww%(1)-WIDTH(title$))/2
PRINT PTAB(ti.tab%); title$
CALL TEXTFACE(0)
CALL TEXTSIZE(12)
FOR n%=1 TO 2
BUTTON n%,1-(n%=cv.type%),bl$(n%),(bx%(n%),by%(n%))
      -(bx1%(n%),by1%(n%)), bt%(n%)
NEXT n%
FOR n%=3 TO 4
BUTTON n%,1,bl$(n%),(bx%(n%),by%(n%))-(bx1%(n%),by1%(n%)),bt%(n%)
NEXT n%
```

The value of cv.type% determines which of the check-box buttons is selected.

While the menu is displayed, the program waits for you to make a selection:

```
mm.loop:
GOSUB wait.event
IF event%=6 THEN selection.made
LET btn%=DIALOG(1)
ON btn% GOTO change.type,change.type,selection.made,quit
quit:
WINDOW CLOSE 1
END
change.type:
BUTTON cv.type%,1   :REM deselect old button
LET cv.type%=btn%
BUTTON cv.type%,2   :REM select new button
GOTO mm.loop
selection.made:
FOR btn%=1 TO 4
BUTTON CLOSE btn%
NEXT btn%
ON cv.type% GOTO alpha.to.numeric, numeric.to.alpha
```

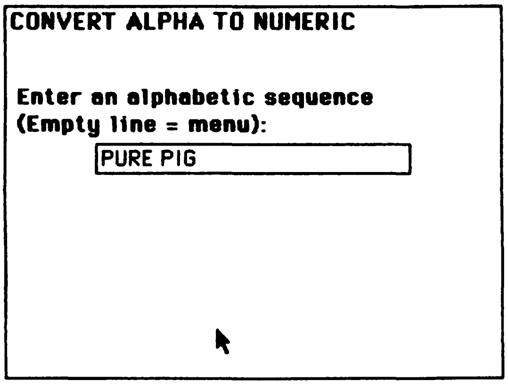
The wait.event subroutine waits for you to click a button or press ENTER or RETURN. If you press ENTER, RETURN, or the OK button, the program closes the buttons and jumps to one of the conversion menus, as determined by cv.type% (ON cv.type% GOTO)...).

If you click one of the check-box buttons, the change.type routine updates the button status, and the program stays inside the idle loop mm.loop.

Alpha-to-Numeric Conversions

The next part of the program handles the alpha-to-numeric option, starting with the block that creates the screen shown in Figure 18-3:

```
alpha.to.numeric:
LET sequence$=nu$
atn.loop:
CLS
CALL TEXTFACE(1)
PRINT "CONVERT ALPHA TO NUMERIC"
CALL MOVETO(6,fy%-24)
```


CONVERT ALPHA TO NUMERIC

Enter an alphabetic sequence
(Empty line = menu):

PURE PIG

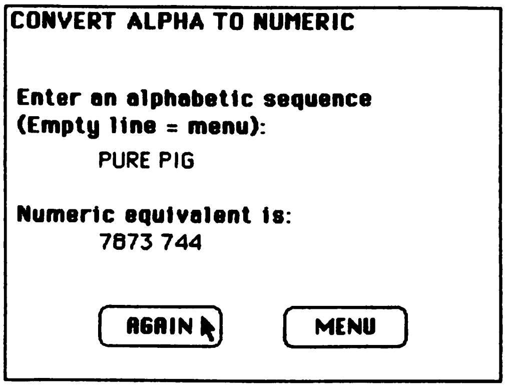
Figure 18-3. Alpha-to-numeric entry screen

```
PRINT "Enter an alphabetic sequence"
PRINT PTAB(6); "(Empty line = menu):"
CALL TEXTFACE(0)
GOSUB get.sequence
IF sequence$=nu$ THEN main.menu
CALL TEXTFACE(1)
PRINT PTAB(6); "Numeric equivalent is:"
CALL TEXTFACE(0)
PRINT PTAB(fx%);
```

The get.sequence subroutine inputs a character sequence from the keyboard by means of an edit field like that shown in Figure 18-3. The same subroutine is used in the numeric-to-alpha subroutine presented later. If you enter a null (by pressing ENTER or RETURN in an empty field), the program returns to the main menu (IF sequence$=nu$...).

Otherwise, the program continues with the following block, which does the alpha-to-numeric conversion:

```
FOR cn%=1 TO LEN(sequence$)
LET c$=MID$(sequence$,cn%,1)
LET ps%=INSTR(1,p$,c$)
IF ps%=0 THEN char.ready
LET pd%=(ps%-1)\3
LET c$=MID$(num$,pd%+1,1)
char.ready:
```


CONVERT ALPHA TO NUMERIC

Enter an alphabetic sequence
(Empty line = menu):
PURE PIG

Numeric equivalent is:
7873 744

Figure 18-4. Alpha-to-numeric output screen

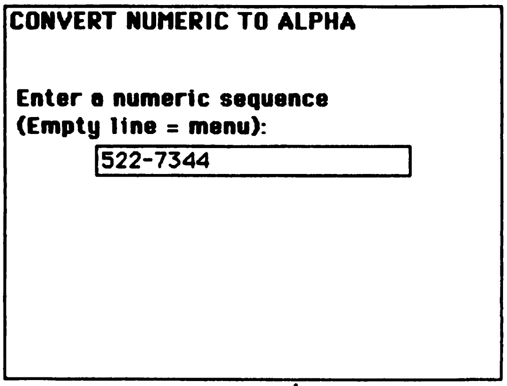
```
PRINT c$;
NEXT cn%
FOR n%=5 TO 6
BUTTON n%-4,1,bl$(n%),(bx%(n%),by%(n%))-(bx1%(n%),by1%(n%)),bt%(n%)
NEXT n%
GOSUB wait.event
BUTTON CLOSE 1
BUTTON CLOSE 2
IF event%=6 OR DIALOG(1)=1 THEN atn.loop
GOTO main.menu
```

The program examines each character c$ of the sequence. If the character occurs in the alpha list p$, it is changed into the corresponding numeric digit in the number list num$. Otherwise it is left unchanged.

The resultant alphabetic sequence is printed underneath the original sequence, and the program lets you continue with the same type of conversion or return to the menu (see Figure 18-4).

Numeric-to-Alpha Conversions

The logic for converting from numeric to alphabetic equivalents is more complicated. First the screen shown in Figure 18-5 is created by the following block:



```
CONVERT NUMERIC TO ALPHA

 Enter a numeric sequence
 (Empty line = menu):
       522-7344
```

Figure 18-5. Numeric-to-alpha entry screen

```
numeric.to.alpha:
LET sequence$=nu$
nta.loop:
CLS
CALL TEXTFACE(1)
PRINT "CONVERT NUMERIC TO ALPHA"
CALL MOVETO(6,fy%-24)
PRINT "Enter a numeric sequence"
PRINT PTAB(6); "(Empty line = menu):"
CALL TEXTFACE(0)
GOSUB get.sequence
IF sequence$=nu$ THEN main.menu
```

Again, the get.sequence subroutine inputs the string value sequence$. If it is null, the program returns to the main menu; otherwise, the following block is executed:

```
LET pl%=LEN(sequence$)
LET nd%=0
FOR cn%=1 TO pl%
LET c$=MID$(sequence$,cn%,1)
IF INSTR(1,num$,c$)>2 THEN LET nd%=nd%+1
NEXT cn%
IF nd% >0 THEN x.able.digits.found
CALL TEXTFACE(1)
```

```
PRINT "No translatable digits found."
CALL TEXTFACE(0)
GOSUB wait.ok
IF event%=6 OR btn%=1 THEN nta.loop ELSE main.menu
x.able.digits.found:
IF nd%<=md% THEN length.ok
CALL TEXTFACE(1)
PRINT "Too many digits. Max is"; md%
CALL TEXTFACE(0)
GOSUB wait.ok
IF event%=6 OR btn%=1 THEN nta.loop ELSE main.menu
```

These lines count the number nd% of translatable digits in sequence$ (numbers 2 through 9). If there are none or too many, the program displays the corresponding error messages (shown in Figures 18-8 and 18-9) and the continuation buttons (AGAIN and MENU).

If the number of translatable digits is between 1 and md%, the program continues with the following block:

```
length.ok:
LET how.many=3*nd%
CLS
CALL TEXTFACE(1)
PRINT "SELECT OUTPUT DEVICE:"
CALL TEXTFACE(0)
FOR n%=7 TO 9
BUTTON n%-6,1-(n%-6=output.to%), bl$(n%),(bx%(n%),by%(n%))- (bx1%(n%),
      by1%(n%)), bt%(n%)
NEXT n%
FOR n%=3 TO 4
BUTTON n%+1,1,bl$(n%),(bx%(n%),by%(n%))-(bx1%(n%),by1%(n%)),bt%(n%)
NEXT n%
```

These lines prompt you to select the output device, as shown in Figure 18-6. The variable output.to% indicates the current output device: 1=screen, 2=printer, 3=disk.

The following block waits for you to press ENTER or RETURN or one of the dialog buttons:

```
lo.loop:
GOSUB wait.event
IF event%=6 THEN open.od
```

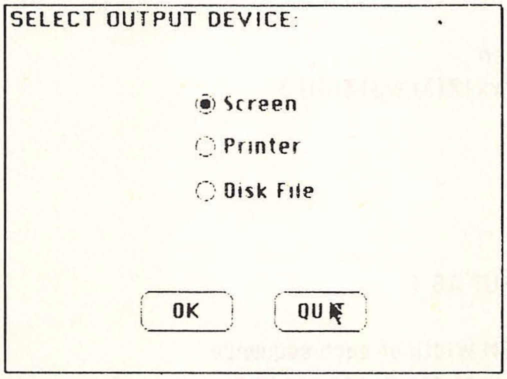


Figure 18-6. Output device selection

```
LET btn%=DIALOG(1)
ON btn% GOTO set.od,set.od,set.od,open.od,main.menu
set.od:
BUTTON output.to%,1
LET output.to%=btn%
BUTTON output.to%,2
GOTO lo.loop
open.od:
WINDOW 1,,(wx%(2),wy%(2))-(wx1%(2),wy1%(2)),3
CLS
CALL TEXTFACE(1)
PRINT "LIST"; how.many; "NAMES FOR ";sequence$
CALL TEXTFACE(0)
IF output.to%<>3 THEN name.ready
LET fo$(3)=FILES$(0,"Name the output file:")
IF fo$(3)=nu$ THEN main.menu
```

Pressing one of the three output-device buttons activates the set.od routine and changes the output device. Pressing ENTER or RETURN is equivalent to pressing the OK button; the program skips the open.od routine.

The open.od routine creates the smaller window shown in Figure 18-7, indicates the number of names to be output, and opens the output device (screen, printer, or disk).

The following lines start the output process.

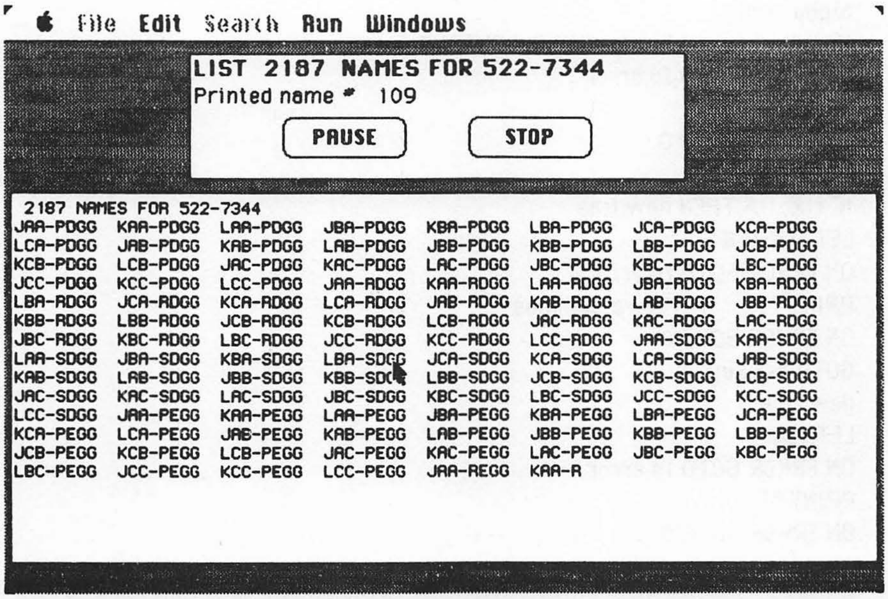
```
name.ready:
IF output.to%<>1 THEN not.screen
WINDOW 2,,(wx%(3),wy%(3))-(wx1%(3),wy1%(3)),3
WINDOW OUTPUT 2
CALL TEXTFONT(4)
CALL TEXTSIZE(9)
not.screen:
ON ERROR GOTO fo.error
OPEN fo$(output.to%) FOR OUTPUT AS 1
ON ERROR GOTO 0
LET iw%=pl%+col.spaces%   :REM Width of each sequence
LET il%=lw%\iw%   :REM Number of sequences per line
LET it%=1
WIDTH #1,255,iw%
IF output.to%=1 THEN WINDOW OUTPUT 2
ON ERROR GOTO fo.error
PRINT#1, 3^nd%; "NAMES FOR ";sequence$
ON ERROR GOTO 0
WINDOW 1
FOR n%=10 TO 11
BUTTON n%-9,1,bl$(n%),(bx%(n%),by%(n%))-(bx1%(n%),by1%(n%)),bt%(n%)
NEXT n%
```

In the case of screen output, the program creates another window, as shown in Figure 18-7.

It% counts the number of names printed on the current output line. When it%=il%, the program starts a new line of output. The program also creates two buttons, PAUSE and STOP, that let you interrupt the list output process.

The next block of lines resets the interrupt status variables and the key k%( ):

```
LET paused%=no%
LET rq.quit%=no%
LET printed=0
LOCATE 2,1
PRINT "Printed name #"
ON DIALOG GOSUB nta.interrupt
DIALOG ON
FOR td%=1 TO nd%
LET k%(td%)=0
NEXT td%
```



```
LIST 2187 NAMES FOR 522-7344
Printed name #  109
```

```
 2187 NAMES FOR 522-7344
JAA-PDGG  KAA-PDGG  LAA-PDGG  JBA-PDGG  KBA-PDGG  LBA-PDGG  JCA-PDGG  KCA-PDGG
LCA-PDGG  JAB-PDGG  KAB-PDGG  LAB-PDGG  JBB-PDGG  KBB-PDGG  LBB-PDGG  JCB-PDGG
KCB-PDGG  LCB-PDGG  JAC-PDGG  KAC-PDGG  LAC-PDGG  JBC-PDGG  KBC-PDGG  LBC-PDGG
JCC-PDGG  KCC-PDGG  LCC-PDGG  JAA-RDGG  KAA-RDGG  LAA-RDGG  JBA-RDGG  KBA-RDGG
LBA-RDGG  JCA-RDGG  KCA-RDGG  LCA-RDGG  JAB-RDGG  KAB-RDGG  LAB-RDGG  JBB-RDGG
KBB-RDGG  LBB-RDGG  JCB-RDGG  KCB-RDGG  LCB-RDGG  JAC-RDGG  KAC-RDGG  LAC-RDGG
JBC-RDGG  KBC-RDGG  LBC-RDGG  JCC-RDGG  KCC-RDGG  LCC-RDGG  JAA-SDGG  KAA-SDGG
LAA-SDGG  JBA-SDGG  KBA-SDGG  LBA-SDGG  JCA-SDGG  KCA-SDGG  LCA-SDGG  JAB-SDGG
KAB-SDGG  LAB-SDGG  JBB-SDGG  KBB-SDGG  LBB-SDGG  JCB-SDGG  KCB-SDGG  LCB-SDGG
JAC-SDGG  KAC-SDGG  LAC-SDGG  JBC-SDGG  KBC-SDGG  LBC-SDGG  JCC-SDGG  KCC-SDGG
LCC-SDGG  JAA-PEGG  KAA-PEGG  LAA-PEGG  JBA-PEGG  KBA-PEGG  LBA-PEGG  JCA-PEGG
KCA-PEGG  LCA-PEGG  JAB-PEGG  KAB-PEGG  LAB-PEGG  JBB-PEGG  KBB-PEGG  LBB-PEGG
JCB-PEGG  KCB-PEGG  LCB-PEGG  JAC-PEGG  KAC-PEGG  LAC-PEGG  JBC-PEGG  KBC-PEGG
LBC-PEGG  JCC-PEGG  KCC-PEGG  LCC-PEGG  JAA-REGG  KAA-R
```

Figure 18-7. Numeric-to-alpha screen output

The total number of names printed is stored in a variable named printed and is displayed in the smaller window shown in Figure 18-7.

Using a Key

The following lines use the current key to produce an alphabetic sequence:

```
key.loop:
WHILE paused% AND NOT rq.quit%
WEND
IF rq.quit% THEN exit.nta
LET d%=1
FOR cn%=1 TO pl%
LET c$=MID$(sequence$,cn%,1)
LET pd%=INSTR(1,num$,c$)
IF pd%<3 THEN alpha.ready
LET c$=MID$(p$,(pd%-1)*3+1+k%(d%),1)
LET d%=d%+1
```

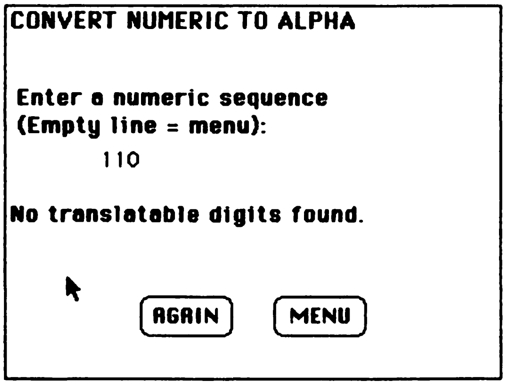
```
alpha.ready:
IF output.to%=1 THEN WINDOW OUTPUT 2
ON ERROR GOTO fo.error
PRINT #1, c$;
ON ERROR GOTO 0
NEXT cn%
IF it%>i1% THEN new.line
LET it%=it%+1
ON ERROR GOTO fo.error
PRINT#1,,   :REM two commas
ON ERROR GOTO 0
GOTO nxt.key
new.line:
LET it%=1
ON ERROR GOTO fo.error
PRINT#1,
ON ERROR GOTO 0
```

D% points to the digit being converted. For each translatable digit in the sequence, the program substitutes a letter from the corresponding triplet in alpha$; the key array k%( ) determines which of the three letters is substituted.

The resulting character c$ is output to the selected device, and the program continues with the next block, which prepares another key:

```
nxt.key:
LET printed=printed+1
IF output.to%=1 THEN WINDOW OUTPUT 1
LOCATE 2,1
PRINT PTAB(100); printed
LET dp%=1
bump.digit:
LET k%(dp%)=k%(dp%)+1
IF k%(dp%)<=2 THEN key.loop
LET k%(dp%)=0
IF dp%=nd% THEN exit.nta
LET dp%=dp%+1   :REM carry to next digit
GOTO bump.digit
```

After updating the number of items printed, the program generates the next key. When all the keys have been used, the program executes the following lines:


CONVERT NUMERIC TO ALPHA

Enter a numeric sequence
(Empty line = menu):
110

No translatable digits found.

Figure 18-8. Numeric-to-alpha error message (no translatable digits)

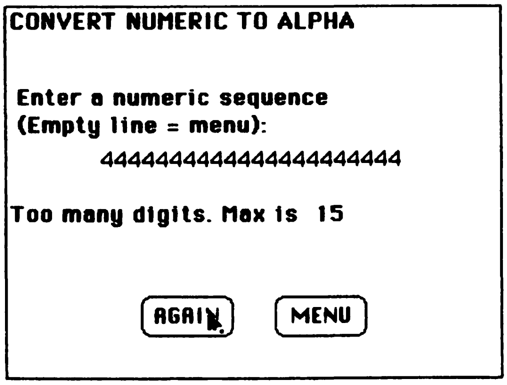
```
exit.nta:
DIALOG STOP
ON ERROR GOTO fo.error
PRINT#1,
CLOSE
err.exit.nta:
ON ERROR GOTO 0
WINDOW 1
BUTTON 1,1,"AGAIN",(bx%(10),by%(10))-(bx1%(10),by1%(10)),bt%(10)
BUTTON 2, 1,"MENU",(bx%(11),by%(11))-(bx1%(11),by1%(11)),bt%(11)
GOSUB wait.event
WINDOW CLOSE 2
IF event%<>6 AND DIALOG(1)<>1 THEN main.menu
WINDOW 1,,(wx%(1),wy%(1))-(wx1%(1),wy1%(1)),3
GOTO nta.loop
```

These lines put the continuation buttons AGAIN and MENU in the dialog box and wait for a selection.

Auxiliary Subroutines

While a list is being printed to the screen, printer, or disk file, you can interrupt the process by pressing the PAUSE or STOP button. The following lines handle the interrupt:

```
nta.interrupt:
LET event%=DIALOG(0)
```



```
CONVERT NUMERIC TO ALPHA

 Enter a numeric sequence
 (Empty line = menu):
        444444444444444444444

Too many digits. Max is  15
```

Figure 18-9. Numeric-to-alpha error message (too many digits)

```
IF event%<>1 THEN RETURN
LET btn%=DIALOG(1)
ON btn% GOTO pause.switch,request.quit
pause.switch:
LET paused%=paused% XOR yes%
LET current.window%=WINDOW(1)
WINDOW 1
BUTTON 1,1,pause$(-paused%), (bx%(10),by%(10))-(bx1%(10), by1%(10)),
      bt%(10)
WINDOW OUTPUT current.window%
RETURN
request.quit:
LET rq.quit%=yes%
RETURN
```

If you press PAUSE, the program sets the paused% flag to 1 and changes the button label to CONTINUE. (If you then press CONTINUE, the program resets the paused% flag and restores the PAUSE label.)

If you press STOP, the program sets the rq.quit% flag and returns to the listing procedure. The quit request is detected as soon as the current sequence is translated.

The next block takes care of output errors that may occur during the listing process.

```
fo.error:
CLOSE
WINDOW 1
BUTTON CLOSE 1
BUTTON CLOSE 2
CLS
IF ERR<50 THEN ON ERROR GOTO 0   :REM Not file-related
PRINT "Can't output to:"
PRINT fo$(output.to%)
BUTTON 1,1,bl$(12),(bx%(12),by%(12))-(bx1%(12),by1%(12)),bt%(12)
GOSUB wait.event
CLS
RESUME err.exit.nta
```

If the error number is less than 50, it is not a file-related error, so the program lets BASIC handle the error in the normal fashion. File-related errors cause the program to print an error notice and return to the main menu.

The following lines display the AGAIN and MENU buttons after completing the sequence conversion:

```
wait.ok:
BUTTON 1,1,"AGAIN",(bx%(3),by%(3))-(bx1%(3),by1%(3)),bt%(3)
BUTTON 2,1,"MENU",(bx%(4),by%(4))-(bx1%(4),by1%(4)),bt%(4)
GOSUB wait.event
IF event%=1 THEN LET btn%=DIALOG(1)
BUTTON CLOSE 1
BUTTON CLOSE 2
RETURN
```

The wait.event subroutine waits until you press ENTER or RETURN or one of the dialog buttons.

```
wait.event:
LET event%=DIALOG(0)
WHILE event%<>1 AND event%<>6
LET event%=DIALOG(0)
WEND
RETURN
```

The final block of the program creates an edit field and waits for you to input a value, which is then stored in sequence$:

```
get.sequence:
EDIT FIELD 1,sequence$,(fx%,fy%)-(fx1%,fy1%)
WHILE DIALOG(0)<>6
WEND
LET sequence$=UCASE$(EDIT$(1))
EDIT FIELD CLOSE 1
CALL MOVETO(fx%,fy%+12)
PRINT sequence$
PRINT
RETURN
```

—Testing and Using the Program

You should be able to duplicate the results shown in Figures 18-2 through 18-9. For numeric-to-alpha conversions, select the screen as the output device and verify that the conversion procedure works properly. Then test the printer and disk options. The disk option creates a file (named by you) that can be loaded into MacWrite.

When using the numeric-to-alpha converter, you don't have to process the entire number at once. You may find it helpful to enter only part of the number at a time—for example, the initial three digits of your telephone number. This reduces the output list to just 27 names. Once you have found a suitable name for part of the number, you can try the other portion.

If your number contains any 1's or 0's, it's a good idea to enter only the segment on either side of these digits. For example, given the number 665-8415, enter the number as 66584, which produces only 243 distinct names. Among them you'll find NOJUG, NOLUI, and OOLUH. Now combine the names with the last two digits to get NOJUG-15, NOLUI-15, and OOLUH-15.

Who knows what bright new name may be hiding in your telephone number?

Chapter **19**

Nutritional Advisor

A one-ounce bag of potato chips provides 150 calories, 3 grams of protein, 14 grams of carbohydrates, and 10 grams of fat. Two peanut butter cups give you 180 calories, 4 grams of protein, 17 grams of carbohydrates, and 11 grams of fat.

All this information (and quite a lot more) is printed on food packages for those who care to know. Almost all prepared foods include similar information.

But how does Grandmother's pineapple upside-down cake stack up? How nutritious is your favorite quiche recipe? When it comes to fresh foods or recipes that you prepare, analyzing your nutritional intake can be complicated.

The Nutritional Advisor program gives you the essential information —calories, carbohydrates, fats, and proteins—about the foods you prepare. Used in conjunction with standard nutritional requirement tables, the program will help you plan a balanced diet.

You may also find it interesting to do food cost/value studies. For example, ounce for ounce, which is a cheaper source of protein: potato chips or filet mignon? The program will help you make such comparisons.

Figures 19-1 through 19-5 summarize the operation of the program.

—The Program

The first block of the program contains descriptors for windows, buttons, and edit fields:

```
DATA 2
DATA 4.35, 4.2, 0.15, 0.375
DATA 2.35, 4.2, 4.6, 0.375
DATA 3
DATA UP, .4, 0.333, 0.1, 0.85,1
DATA DOWN, .6, 0.333, 0.5, 0.85,1
DATA OK, .4, 0.333, 0.9, 0.85,1
DATA 3
DATA 2.0, 0.208, 0.5, 0.04
DATA 0.75, 0.208, 0.8, 0.95
DATA 0.75, 0.208, 0.8, 0.55
```

The next block contains nutritional data:

```
DATA 48
DATA milk, cup, 165, 8, 12, 10
DATA whipping cream, cup, 860, 4, 6, 94
DATA cottage cheese, cup, 240, 30, 6, 11
DATA cheddar cheese, 1-inch cube, 70, 4, 0, 6
DATA cream cheese, oz, 105, 2, 1, 11
DATA eggs, egg, 75, 6, 0, 6
DATA butter, 1/4-lb stick, 800, 0, 0, 90
DATA margarine, 1/4-lb stick, 806, 0, 0, 91
DATA veg. oil, tbs, 125, 0, 0, 14
DATA ground beef, lb, 1307, 112, 0, 91
DATA chicken, lb, 1326, 114, 0, 91
DATA lamb, lb, 1675, 107, 0, 75
DATA ham, lb, 1547, 85, 0, 117
DATA cod, lb, 777, 128, 0, 23
DATA flounder, lb, 914, 137, 0, 37
DATA crabmeat, lb, 480, 75, 5, 11
DATA tuna, lb, 907, 133, 0, 37
DATA gr. snap beans, cup, 25, 1, 6, 0
DATA gr. lima beans, cup, 140, 8, 24, 0
DATA canned kidney beans, cup, 230, 15, 42, 0
DATA broccoli, cup, 45, 5, 8, 0
DATA cabbage, cup, 40, 2, 9, 0
DATA carrots, cup, 45, 1, 10, 0
DATA cauliflower, cup, 30, 3, 6, 0
DATA celery, cup, 20, 1, 4, 0
DATA corn, cup, 170, 5, 41, 0
DATA mushrooms, .5 c, 12, 2, 4, 0
DATA onions, cup, 80, 2, 18, 0
DATA canned gr. peas, cup, 68, 3, 13, 0
DATA potatoes, medium size, 100, 2, 22, 0
DATA canned tomatoes, cup, 50, 2, 9, 0
DATA spinach, cup, 26, 3, 3, 0
DATA apples, cup, 100, 0, 26, 0
DATA banana, medium size, 85, 0, 23, 0
DATA canned blueberries, cup, 245, 1, 2, 0
DATA canned peaches, cup, 200, 0, 52, 0
DATA canned pineapple, slice, 95, 0, 26, 0
DATA raisins, cup, 230, 2, 62, 0
DATA corn meal, cup, 360, 9, 74, 4
DATA white flour, cup, 400, 12, 84, 0
DATA whole wheat flour, cup, 390, 13, 79, 2
DATA brown rice, cup, 748, 15, 154, 3
DATA white rice, cup, 692, 14, 150, 0
DATA noodles, cup, 200, 7, 37, 2
DATA oatmeal, cup, 150, 5, 26, 3
DATA sugar, cup, 770, 0, 199, 0
DATA almonds, .5 cup, 425, 13, 13, 38
DATA walnuts, .5 cup, 325, 7, 8, 32
```

The program includes nutritional information for 48 common foods. You can add your own favorite ingredients to the list as well. To add a food, you need the following information:

- food name
- measurement unit
- calories per measurement unit
- protein (grams per measurement unit)
- carbohydrates (grams per measurement unit)
- fat (grams per measurement unit).

One handy compilation of nutritional data can be found in *Let's Get Well*, by Adelle Davis (New York: Harcourt Brace Jovanovich, Inc., 1965). The data in this program was selected from a much longer list of foods presented in that book.

If you add any foods to the list, you must change the number in the DATA line preceding the food list. The value is currently set to 48.

Reading the Data

The next lines read in the window descriptors:

```
READ nw%
DIM ww%(nw%),wl%(nw%),wx%(nw%),wy%(nw%),wx1%(nw%),wy1%(nw%)
FOR n%=1 TO nw%
READ inches.wide, inches.long, ulcx, ulcy
LET ww%(n%)=inches.wide*72
LET wl%(n%)=inches.long*72
LET wx%(n%)=ulcx*72
LET wy%(n%)=ulcy*72
LET wx1%(n%)=wx%(n%)+ww%(n%)
LET wy1%(n%)=wy%(n%)+wl%(n%)
NEXT n%
```

Here are the corresponding lines for the button descriptors:

```
READ nb%
DIM bl$(nb%),bx%(nb%),by%(nb%),bx1%(nb%),by1%(nb%),bt%(nb%)
FOR n%=1 TO nb%
LET w%=2   :REM buttons go in window 2
READ bl$(n%),inches.wide, inches.long, hzone, vzone, bt%(n%)
LET bx%(n%)=(ww%(w%)-inches.wide*72)*hzone
LET by%(n%)=(wl%(w%)-inches.long*72)*vzone
LET bx1%(n%)=bx%(n%)+inches.wide*72
LET by1%(n%)=by%(n%)+inches.long*72
NEXT n%
```

And here are the lines for the edit fields:

```
READ nf%
DIM fx%(nf%),fy%(nf%),fx1%(nf%),fy1%(nf%),ingred%(12),quantity%(12)
LET w%=1   :REM edit fields go in window 1
FOR n%=1 TO nf%
READ inches.wide, inches.long, hzone, vzone
LET fx%(n%)=(ww%(w%)-inches.wide*72)*hzone
LET fy%(n%)=(wl%(w%)-inches.long*72)*vzone
LET fx1%(n%)=fx%(n%)+inches.wide*72
LET fy1%(n%)=fy%(n%)+inches.long*72
NEXT n%
```

The next block of lines reads in the nutritional data and sets up certain functions and constants:

```
READ nf%
DIM food$(nf%),unit$(nf%),cal(nf%),protein(nf%),carbo(nf%),fat(nf%)
FOR n%=1 TO nf%
READ food$(n%),unit$(n%),cal(n%),protein(n%),carbo(n%),fat(n%)
NEXT n%
DIM rectangle%(3)
DEF FNmin%(a,b)=-(a<=b)*a-(b<a)*b :REM calculate minimum(a,b)
LET np%=14   :REM items per screen-page
LET yes%=(1=1)
LET no%=(1=0)
```

Nutritional Advisor Menu

The following lines create the pull-down menu:

```
MENU 6,0,1,"Nutritional Advisor"
MENU 6,1,1,"New recipe "
MENU 6,2,0,"Print "
MENU 6,3,1,"Quit "
MENU ON
ON MENU GOSUB change.modes
GOTO new.recipe
idle.loop:
MENU 6,1,1   :REM enable new.recipe option
MENU 6,2,-(nr.selected%>0)   :REM enable print option if non-empty
loop: GOTO loop
```

After setting up the pull-down menu, the computer automatically executes the new.recipe routine.

Selecting an item from the Nutritional Advisor menu activates the following subroutine:

```
change.modes:
IF MENU(0)<>6 THEN RETURN
CLOSE 1   :REM close printer channel
```

```
WINDOW CLOSE 2
ON MENU(1) GOTO rq.new.recipe,rq.print.analysis,rq.quit
rq.new.recipe:
MENU 6,2,0   :REM disable print option
RETURN new.recipe
rq.print.analysis:
MENU 6,1,0   :REM disable new.recipe option
MENU 6,2,0   :REM disable print option
RETURN print.analysis
rq.quit:
END
```

The new.recipe option is disabled during execution of the print option, and vice versa.

Print Analysis Option

The following lines output a complete recipe analysis to the printer.

```
print.analysis:
IF nr.selected%=0 THEN GOTO idle.loop
WINDOW 1
CLS
OPEN "LPT1:DIRECT" FOR OUTPUT AS 1
PRINT #1,title$
PRINT#1,
PRINT #1,"Ingredients:"
FOR ig%=1 TO nr.selected%
PRINT #1, USING "##.## "; quantity%(ig%);
PRINT#1, unit$(ingred%(ig%));SPC(2); food$(ingred%(ig%))
NEXT ig%
PRINT#1,
PRINT #1, "Serves"; servings%
IF servings%=0 THEN servings.done
PRINT#1,"Nutritional analysis per serving:"
PRINT #1, USING "calories: #####.#"; cal/servings%
PRINT #1, USING "protein: ####.# g"; protein/servings%
PRINT #1, USING "carbohydrate: ####.# g"; carbo/servings%
PRINT #1, USING "fat: ####.# g";fat/servings%
servings.done:
CLOSE 1
GOTO idle.loop
```

Upon completion of the printout, the computer waits in the idle.loop routine until you make another selection from the pull-down menu.

Starting a New Recipe

The following lines take over when you select the new.recipe option from the Nutritional Advisor menu:

```
new.recipe:
WINDOW 1,,(wx%(1),wy%(1))-(wx1%(1),wy1%(1)),3
CALL TEXTMODE(0)   :REM Erase-before-printing mode
CLS
EDIT FIELD 1,"Recipe Title",(fx%(1),fy%(1))-(fx1%(1),fy1%(1)),,2
WHILE DIALOG(0)<>6
WEND
LET title$=EDIT$(1)
EDIT FIELD CLOSE 1
LET center%=(ww%(1)-WIDTH(title$))/2
CALL MOVETO(center%,fy%(1)+12)
PRINT title$
LINE (0,wl%(1)\2)-(ww%(1),wl%(1)\2)
LET cal=0
LET protein=0
LET carbo=0
LET fat=0
```

The lines produce the screen shown in Figure 19-1. The program waits for you to type in the name of the recipe to be analyzed, after which the counters for cumulative protein, carbohydrates, and fat are cleared.

Displaying the Food List

The following block sets up the food list window:

```
WINDOW 2,,(wx%(2),wy%(2))-(wx1%(2),wy1%(2)),3
CALL TEXTMODE(1)   :REM overprinting mode
LET nr.selected%=0
LET fop%=1
LET hilited%=0
FOR n%=1 TO 3
```

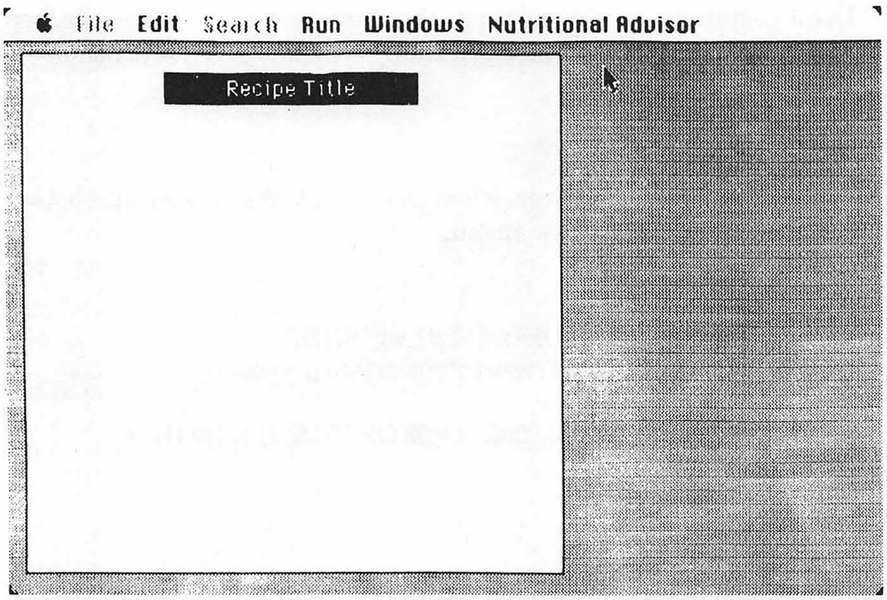


Figure 19-1. The initial screen appearance of Nutritional Advisor

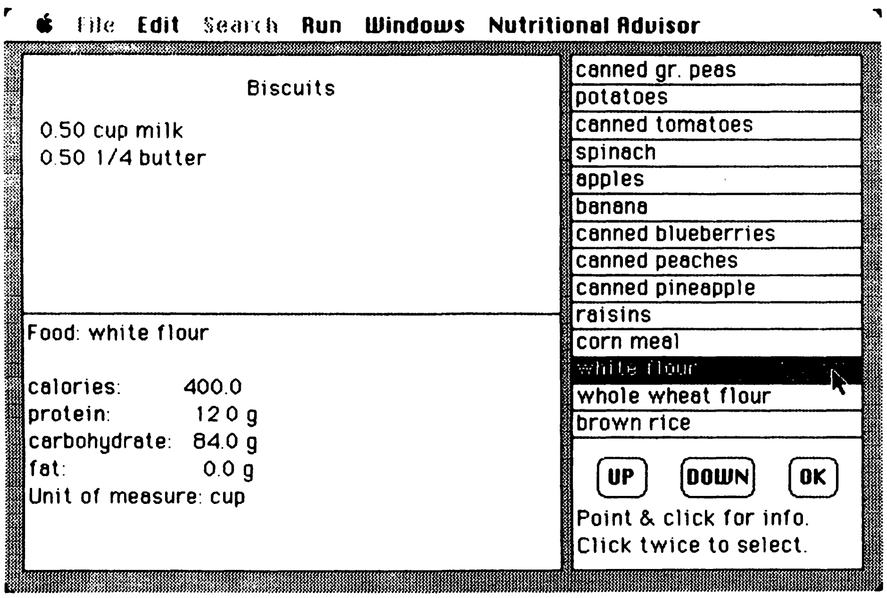
```
BUTTON n%, -(n%=3), bl$(n%), (bx%(n%), by%(n%))- (bx1%(n%), by1%(n%)),
      bt%(n%)
NEXT n%
CALL MOVETO(2,by1%(1)+16)
PRINT "Point & click for info."
PRINT "Click twice to select.";
GOSUB show.page
```

Window 2 is the smaller window on the right of the screen in Figure 19-2. Nr.selected% counts the number of ingredients selected so far. The program allows up to 12 ingredients in a single recipe.

Fop% indicates which food is first on the current page (the entire food list can't fit on the screen at once, so it is broken up into pages). Hilited% points to the currently highlighted food; hilited%=0 indicates that no item is highlighted.

The show.page subroutine displays the current page of the food list, as determined by the value of fop%.

The next block waits for you to select a food or to press OK:



```
Biscuits

0.50 cup milk
0.50 1/4 butter

Food: white flour

calories:       400.0
protein:         12.0 g
carbohydrate:   84.0 g
fat:             0.0 g
Unit of measure: cup
```

```
canned gr. peas
potatoes
canned tomatoes
spinach
apples
banana
canned blueberries
canned peaches
canned pineapple
raisins
corn meal
white flour
whole wheat flour
brown rice

Point & click for info.
Click twice to select.
```

Figure 19-2. Screen appearance with "white flour" highlighted. White flour information appears in the bottom half of the left-hand window

```
sf.loop:
LET d.clicked%=no%
WHILE DIALOG(0) <>1 AND NOT d.clicked%
LET mouse.event%=MOUSE(0)
IF mouse.event%<=0 THEN no.click
LET d.clicked%=(mouse.event%>1 AND hilited%>0)
IF d.clicked% THEN no.click
LET my%=MOUSE(2)
LET newlin.nr%=(my%-1)\16+1
IF newlin.nr%>lop%-fop%+1 OR newlin.nr%=hilited% THEN no.click
IF hilited%>0 THEN CALL INVERTRECT(VARPTR(rectangle%(0)))
LET hilited%=newlin.nr%
LET f.name%=fop%+hilited%-1
LET rectangle%(0)=(hilited%-1)*16
LET rectangle%(1)=0
LET rectangle%(2)=hilited%*16
LET rectangle%(3)=ww%(2)
CALL INVERTRECT(VARPTR(rectangle%(0)))   :REM hilite new item
```

Refer to Figure 19-2. Pressing UP moves the food list to the preceding screen page, if one exists. Pressing DOWN moves the food list to the following screen page. Pressing OK indicates that the recipe is complete.

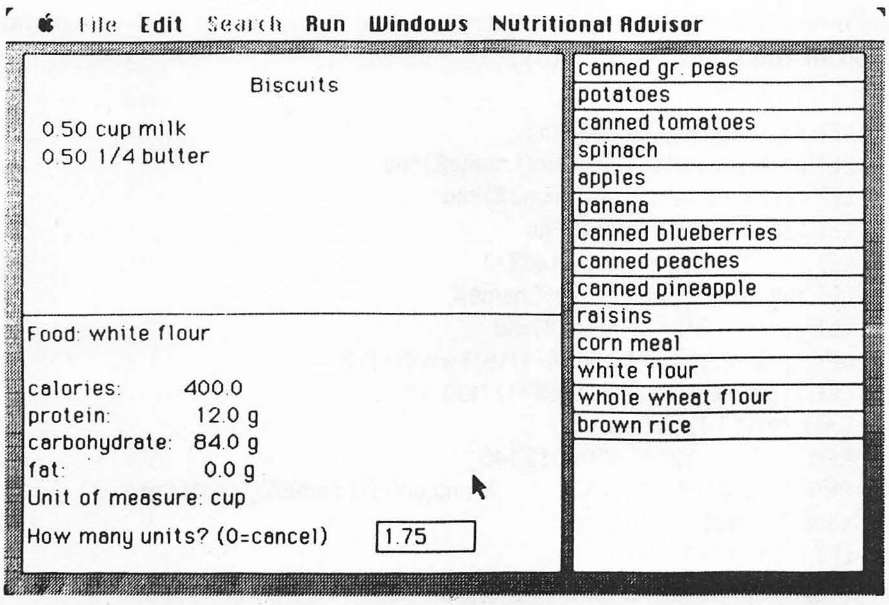
To obtain information about a food, you point to it with the mouse and click once; the item will be highlighted and the following block will provide descriptive information:

```
WINDOW 1
LINE (0,wl%(1)\2+1)-(ww%(1),wl%(1)),0,bf
CALL MOVETO(2,wl%(1)\2+16)
PRINT "Food: "; food$(f.name%)
PRINT
PRINT USING "calories:      ####.#"; cal(f.name%)
PRINT USING "protein:        ###.# g"; protein(f.name%)
PRINT USING "carbohydrate: ###.# g"; carbo(f.name%)
PRINT USING "fat:            ###.# g"; fat(f.name%)
PRINT "Unit of measure: "; unit$(f.name%)
```

While an item is highlighted, you may select it for the recipe by double-clicking the mouse anywhere in window 2. If you don't wish to use a highlighted item, simply click on another item or move to another page. The following lines respond to double-clicks and button presses:

```
WINDOW 2
no.click:
WEND
IF d.clicked% THEN selected
LET btn%=DIALOG(1)
ON btn% GOTO back,forward,recipe.done
selected:
CALL INVERTRECT(VARPTR(rectangle%(0)))   :REM cancel previous
       highlighting
BUTTON CLOSE 1
BUTTON CLOSE 2
BUTTON CLOSE 3
LINE (0,np%*16+1)-(ww%(2),wl%(2)),0,bf
```

The WEND statement causes the program to loop back to the sf.loop routine unless you have double-clicked or pressed a button. In the case of a double-click, the program goes to the routine called selected. In the



Biscuits

0.50 cup milk
0.50 1/4 butter

Food: white flour

```
calories:        400.0
protein:          12.0 g
carbohydrate:     84.0 g
fat:               0.0 g
Unit of measure: cup
How many units? (0=cancel)    1.75
```

canned gr. peas
potatoes
canned tomatoes
spinach
apples
banana
canned blueberries
canned peaches
canned pineapple
raisins
corn meal
white flour
whole wheat flour
brown rice

Figure 19-3. Screen appearance after double-clicking on "white flour"

case of a button press, the program goes to the corresponding button routine (ON btn% GOTO...).

Selected starts by closing the buttons and erasing the two lines at the bottom of window 2 (LINE (0,np%*16+1)...).

The following lines add an edit field to window 1 and prompt you to specify the quantity of the selected ingredient (see the bottom line of window 1 in Figure 19-3):

```
WINDOW 1
CALL MOVETO(2,fy%(2)+12)
PRINT "How many units? (0=cancel)";
EDIT FIELD 1," ",(fx%(2),fy%(2))-(fx1%(2),fy1%(2))
WHILE DIALOG(0)<>6
WEND
LET nu=VAL(EDIT$(1))
EDIT FIELD CLOSE 1
LINE (0,w1%(1)\2+1)-(ww%(1),w1%(1)),0,bf
IF nu=0 THEN cancel.select
```

Entering 0 for the amount cancels the selection. Otherwise, the following lines calculate the nutritional contribution of that particular food in the specified quantity:

```
LET cal=cal+cal(f.name%)*nu
LET protein=protein+protein(f.name%)*nu
LET carbo=carbo+carbo(f.name%)*nu
LET fat=fat+fat(f.name%)*nu
LET nr.selected%=nr.selected%+1
LET ingred%(nr.selected%)=f.name%
LET quantity%(nr.selected%)=nu
LET ig.x%=2+((nr.selected%-1)\6)*ww%(1)\2
LET ig.y%=48+((nr.selected%-1) MOD 6)*16
CALL MOVETO(ig.x%,ig.y%)
REM          123456789012345
PRINT USING "**.** \ \ \       \"; nu,unit$(f.name%),food$(f.name%)
cancel.select:
LET hilited%=0
IF nr.selected%=12 THEN recipe.done
```

Nu is the number of measurement units. Multiplying this value times the nutritional value per measurement unit gives the total nutritional value of that food in the recipe. For instance,

LET protein = protein + protein(f.name%) * nu

adds the current food's protein contribution to the cumulative protein total.

After all four food elements are calculated, the program prints the food in the ingredient list at the top of window 1. The text in this section will be partially obscured or truncated if it is too long to fit in the allotted space. However, the full text is printed when you select the Print option from the Nutritional Advisor menu.

Upon completion of the food-quantity dialog, the program continues with this block:

```
WINDOW 2
BUTTON 1,-(fop%>1),bl$(1),(bx%(1),by%(1))-(bx1%(1),by1%(1)),bt%(1)
BUTTON 2,-(lop%<nf%),bl$(2),(bx%(2),by%(2))-(bx1%(2),by1%(2)),bt%(2)
BUTTON 3,1,bl$(3),(bx%(3),by%(3))-(bx1%(3),by1%(3)),bt%(3)
CALL MOVETO(2,by1%(1)+16)
PRINT "Point & click for info."
PRINT "Click twice to select.";
GOTO sf.loop
```

Paging Up and Down

The next block includes routines to show the preceding (up) and following (down) screen pages of the food list.

```
back:
IF fop%<=np% THEN sf.loop
LET fop%=fop%-np%
GOSUB show.page
GOTO sf.loop
forward:
IF fop%>nf%-np%+1 THEN sf.loop
LET fop%=fop%+np%
GOSUB show.page
GOTO sf.loop
```

Here's the routine that displays the current page, starting with food number fop%:

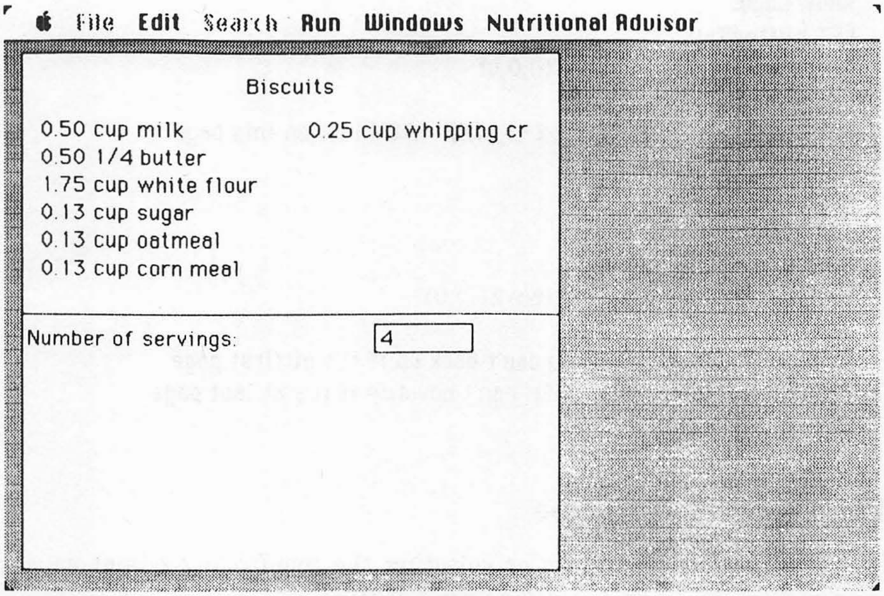
```
show.page:
LET hilited%=0
LINE (0,0)-(ww%(2),np%*16),0,bf
LOCATE 1,1
LET lop%=FN min%(np%+fop%-1,nf%)    :REM last on this page
FOR j%=fop% TO lop%
LET line.nr%=j%-fop%+1
LOCATE line.nr%,1
PRINT food$(j%)
LINE (0,line.nr%*16)-STEP(ww%(2),0)
NEXT j%
BUTTON 1,-(fop%>1)   :REM can't back up if it's at first page
BUTTON 2,-(lop%<nf%)   :REM can't advance if it's at last page
RETURN
```

The Completed Recipe

Pressing the DONE button or selecting the twelfth ingredient causes the program to begin the recipe.done routine.

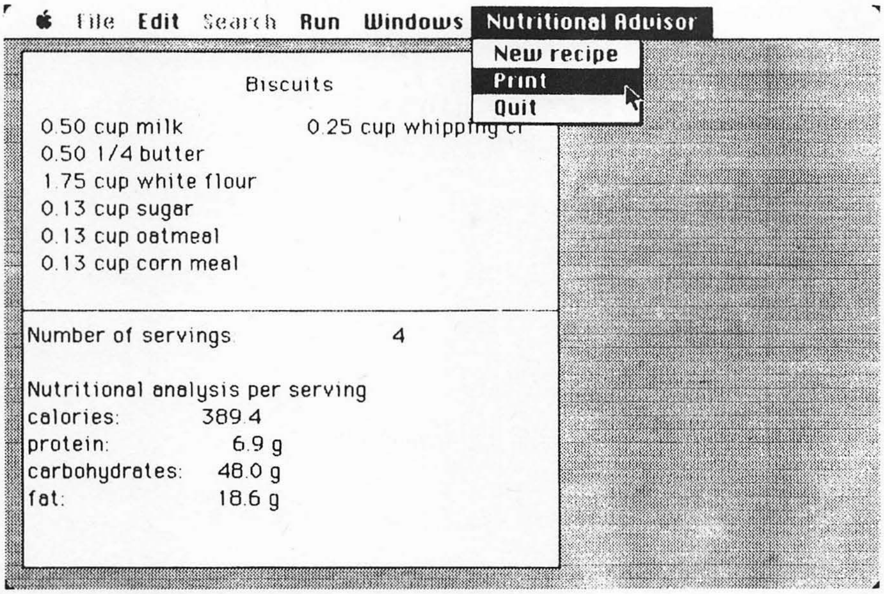
```
recipe.done:
WINDOW CLOSE 2
WINDOW 1
LINE (0,wl%(1)\2+1)-(ww%(1),wl%(1)),0,bf
IF nr.selected%=0 THEN idle.loop
CALL MOVETO(2,fy%(3)+12)
PRINT "Number of servings:"
get.servings:
EDIT FIELD 1,"",(fx%(3),fy%(3))-(fx1%(3),fy1%(3))
WHILE DIALOG(0)<>6
WEND
LET servings%=VAL(EDIT$(1))
EDIT FIELD CLOSE 1
IF servings%=0 THEN BEEP: GOTO get.servings
```

The program asks you to specify the number of servings in the recipe; from this information, it calculates the nutritional value of each serving. See Figure 19-4.



```
Biscuits

0.50 cup milk          0.25 cup whipping cr
0.50 1/4 butter
1.75 cup white flour
0.13 cup sugar
0.13 cup oatmeal
0.13 cup corn meal

Number of servings:   4
```

Figure 19-4. Screen appearance after pressing the OK button. The program prompts you for the number of servings



```
                    Biscuits
0.50 cup milk            0.25 cup whipping cr
0.50 1/4 butter
1.75 cup white flour
0.13 cup sugar
0.13 cup oatmeal
0.13 cup corn meal

Number of servings          4

Nutritional analysis per serving
calories:        389.4
protein:           6.9 g
carbohydrates:    48.0 g
fat:              18.6 g
```

Figure 19-5. The completed ingredient list and nutritional analysis

The Nutritional Analysis

The following lines put the nutritional analysis on the screen:

```
CALL MOVETO(fx%(3),fy%(3)+12)
PRINT servings%
PRINT
PRINT "Nutritional analysis per serving"
PRINT USING "calories:      ####.#"; cal/servings%
PRINT USING "protein:       ####.# g"; protein/servings%
PRINT USING "carbohydrates: ####.# g"; carbo/servings%
PRINT USING "fat:           ####.# g"; fat/servings%
GOTO idle.loop
```

After displaying the analysis, the program goes to the idle.loop routine to wait for another menu selection. This is the point at which you would normally select the Print option from the Nutritional Advisor menu, as shown in Figure 19-5.

The food list is based on data from the U.S. Department of Agriculture. The data is available in many encyclopedias and books. One handy compilation can be found in *Let's Get Well*, by Adelle Davis (New York: Harcourt Brace Jovanovich, Inc. 1965).

Chapter 20

The Time Machine

A calendar is like a time machine: it helps you wander through the past and future. In this chapter, we present the Time Machine program, which produces calendars over any 89-year period from 1900 to 2099.

In addition to facilitating mental time-travel, the program has practical benefits as a scheduling tool for the home or office. Before printing a month's calendar, you can insert information about birthdays, appointments, social engagements, deadlines, holidays, and other events. You can even save and retrieve calendar notes to and from disk so you won't have to retype events and schedules that are the same from month to month.

Figures 20-1 through 20-9 show typical screens from the program.

—Calendar Logic

Two items are needed to produce an accurate monthly calendar: the number of days in the month and the weekday on which the month begins.

Finding the number of days in a month is a trivial exercise, even for the Macintosh. February is a special case, since its length depends on whether the year is a leap year. A leap year is evenly divisible by 4, unless the year happens to be the first year of a new century, in which case it must be divisible by 400. For instance, 1900 is not a leap year, 2000 is, and 2100 is not.

Finding the weekday on which a month begins is more difficult. One method involves referring to tables consisting of hundreds of numbers and letter codes. A simpler method starts with a known base date and extrapolates forward from that date. For example, if you know that March 1, 1940, occurred on a Friday, you can calculate the first day of the week for any subsequent date.

The Time Machine uses the latter method. Because of limitations in numeric precision (16-bit integers are used), its calendar calculations are limited to a span of 89 years anywhere in the range from 1900 to 2099.

—The Program

The program starts with the window descriptors:

```
REM Windows
DATA 7, 4.38, .05, 0.32
READ inches.wide, inches.long, ulcx, ulcy
LET ww%=inches.wide*72
LET wl%=inches.long*72
LET wx%=ulcx*72
LET wy%=ulcy*72
LET wx1%=wx%+ww%
LET wy1%=wy%+wl%
```

The next block loads the calendar data:

```
DIM mon$(12),mlen%(12),wday$(7),cal%(42),note$(31)
FOR m%=1 TO 12
READ mon$(m%),mlen%(m%)
NEXT m%
FOR day%=1 TO 7
READ wday$(day%)
NEXT day%
DATA January, 31, February, 28, March, 31, April, 30
DATA May, 31, June, 30, July, 31, August, 31, September, 30
DATA October, 31, November, 30, December, 31
DATA Sun, Mon, Tue, Wed, Thu, Fri, Sat
LET base.year%=1940   :REM any leap year from 1900 to 2010
LET first.of.march%=5   :REM weekday of March 1st in the base year
      (0=Sunday)
LET last.year%=base.year%+89
```

Mon$( ) and mlen%( ) store the names and lengths of the months. Wday$( ) stores the names of the weekdays.

Cal%( ) stores the 6 × 7 grid onto which the calendar days are mapped. For i=1 to 42, cal$(i) refers to the cell at row ((i−1)/7) + 1 and column ((i−1) mod 7) + 1. Cal%(i)=0 indicates a blank cell (cells before the first and after the last day of the month); nonzero values of cal%( ) indicate the date that is given to that calendar cell.

Note$( ) stores the notes that go with each numbered cell of the calendar.

Base.year% is set to 1940, giving the calendar a range from March 1940 through November 2029. You may change this range by giving another value for base.year%. Base.year% must be a leap year from 1900 to 2010. If you change base.year%, you must also change the value of first.of.march% to correspond to the day of the week of March 1 in base.year%. For instance, if you set base.year%=1920, then first.of.march% must be set to 1, since March 1, 1920, falls on a Monday. (Sunday=0, Monday=1,...Saturday =6).

The next block defines a function and assigns additional constants and parameters:

```
DEF FNstrip$(x)=RIGHT$(STR$(x),LEN(STR$(x))-1)
LET cw%=ww%\7
LET cl%=wl%\7
LET hl%=wl%\14
LET bx%=1
LET bx1%=ww%-1
LET by%=1
LET by1%=hl%-1
LET fy%=by%+hl%\2
LET yes%=(1=1)
LET no%=(1=0)
LET nu$=""   :REM no spaces inside quotes
```

The function FNstrip$ returns the string equivalent of a positive number without the usual leading space. Cw% and cl% are the relative width and length of each cell. Hl% is the relative length of the heading box.

The other numeric variables define specific points inside the window.

Setting Up the Menu

The program next sets up the menu seen in Figure 20-1:

```
WINDOW 1,,(wx%,wy%)-(wx1%,wy1%),3
CALL TEXTMODE(1)
CALL TEXTFONT(2)  :REM New York (must pick a typefont with 9-, 12-, and
       14-point)
FOR mnu%=3 TO 5
MENU mnu%,0,1," "
NEXT mnu%
MENU 3,0,0,"Time Machine"
MENU 3,1,1,"Erase notes "
MENU 3,2,1,"Load note file "
MENU 3,3,1,"Save note file "
MENU 3,4,1,"Print  "
```

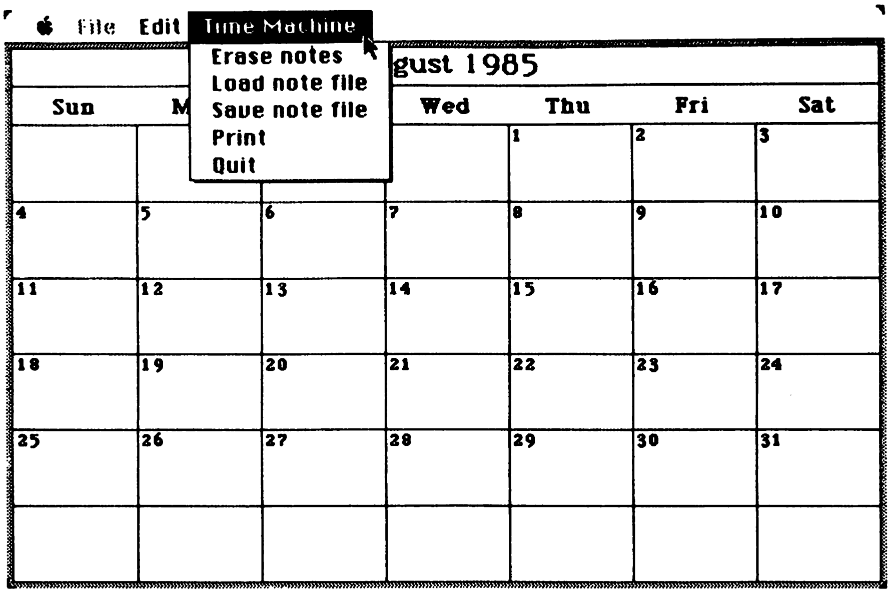


Figure 20-1. Initial screen appearance of the Time Machine and its pull-down menu

```
MENU 3,5,1,"Quit  "
LET m%=8   :REM initial month for calendar
LET y%=1985   :REM initial year for calendar
GOSUB new.date
GOSUB redraw.calendar
```

The program uses text font #4 (New York) for all text. You can use another font, but whichever font you use, it must be available in sizes 9, 12, and 14. (You can use Apple's Font Mover program to get these font sizes onto the BASIC startup disk if you don't have them.)

The initial calendar month m% is set to 8 (August) and the year y% is set to 1985. You can change these initial settings to suit your preference.

The new.date subroutine performs all the date calculations based on the setting for m% and y%. Redraw.calendar displays the resultant calendar.

Main Control Loop

While the calendar is displayed, the following routine waits for you to make a selection from the pull-down menu or to click the mouse inside one of the calendar cells:

```
calendar.displayed:
MENU 3,0,1   :REM enable menu
LET x%=MOUSE(0)   :REM ignore previous mouse events
LET x%=MENU(0)   :REM ignore previous menu events
LET menu.event%=0
LET mouse.event%=0
WHILE menu.event%<>3 AND mouse.event%<>1
LET menu.event%=MENU(0)
LET mouse.event%=MOUSE(0)
WEND
MENU 3,0,0   :REM disable menu
IF menu.event%<>3 THEN clicked.mouse
LET chosen%=MENU(1)
IF chosen%=5 THEN MENU RESET: END
ON chosen% GOSUB erase.notes,load.notes,save.notes,print.cal
GOTO calendar.displayed
```

To add a note to one of the numbered cells, you click the mouse inside the cell. To change the calendar date, you click in the upper portion of the calendar (the area that contains the calendar month and year).

The following block discriminates between the various mouse-clicks:

```
clicked.mouse:
LET mx%=MOUSE(1)
LET my%=MOUSE(2)
IF my%<hl% THEN GOSUB change.date: GOTO calendar.displayed
IF my%<cl% THEN BEEP: GOTO calendar.displayed
LET col%=(mx%-1)\cw%+1
LET row%=(my%-1)\cl%   :REM don't add 1 because of title row
LET cn%=col%+(row%-1)*7   :REM cell number
LET d%=cal%(cn%)
IF d%=0 THEN BEEP: GOTO calendar.displayed
LET dx%=(col%-1)*cw%
LET dy%=row%*cl%
CALL TEXTFACE(0)
CALL TEXTSIZE(9)
EDIT FIELD 1,note$(d%),(dx%+3,dy%+2)-(dx%+cw%-2,dy%+cl%-2),2
BUTTON 1,1,"OK",(ww%*13/16,hl%/8)-(ww%*15/16,hl%*7/8),1
LET x%=DIALOG(0)   :REM ignore previous events
WHILE DIALOG(0)<>1
WEND
```

My%<hl% indicates that you clicked the mouse in the area containing the month and year; in this case the program goes to the change. date subroutine. The program identifies the row number, row%, and column number, col%, of the cell that you clicked. Cn% is the cell's index number in the cal%( ) array, and d% is the number assigned to that cell.

Adding Notes to a Cell

If you click on an unnumbered cell (d%=0), the program beeps and returns to the main control loop. Otherwise, it creates an edit field for the note, as shown in Figure 20-2.

You must press ENTER or RETURN to end each line of the note. When you are through entering text into the edit field, click the OK button. At that point the following block takes over:

```
LET note$(d%)=EDIT$(1)
BUTTON CLOSE 1
EDIT FIELD CLOSE 1
LET dy%=dy%+9
LET dx%=dx%+3
```

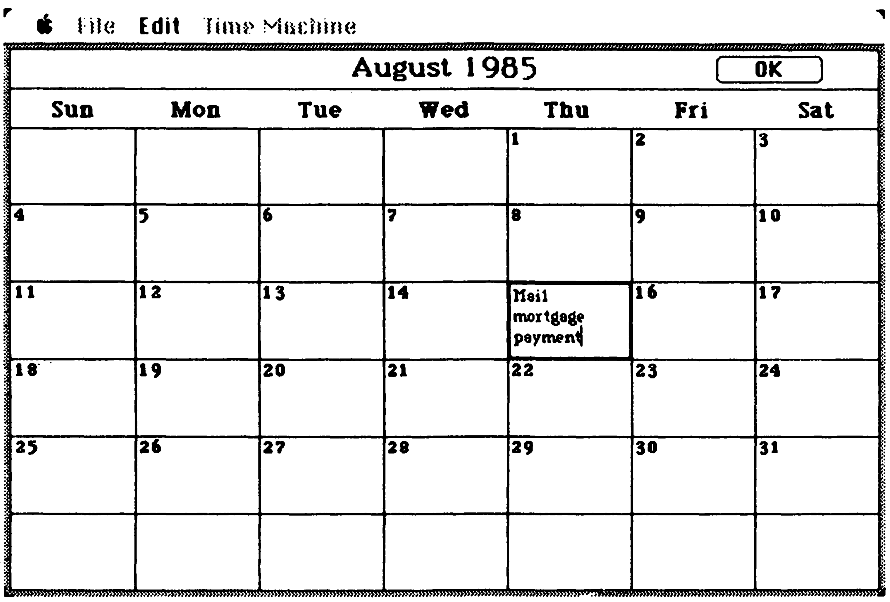


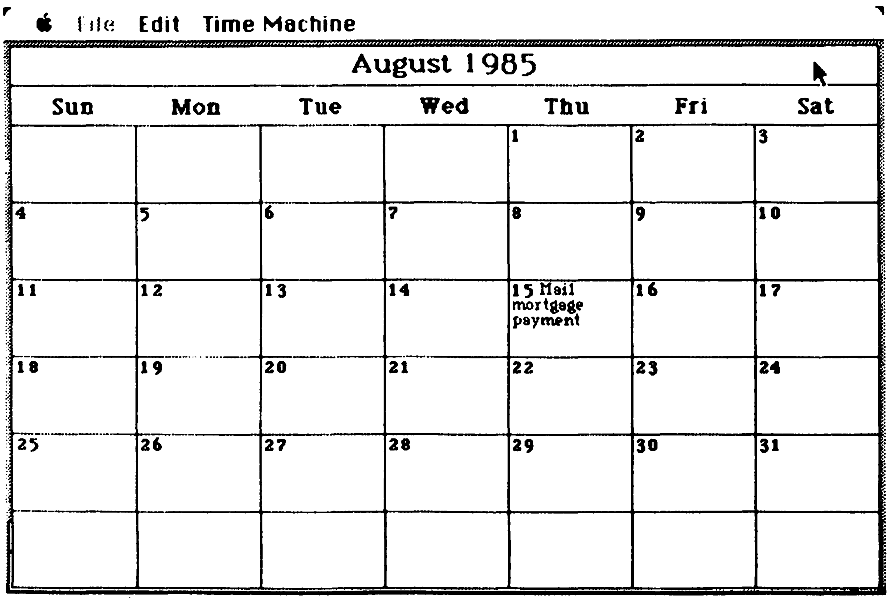
Figure 20-2. Entering notes for August 15. You must press ENTER after each line of the note and click the OK button when the note is complete

```
GOSUB update.cell
LET dx%=(col%-1)*cw%
LET dy%=row%*cl%
CALL MOVETO(dx%+2,dy%+10)
CALL TEXTFACE(1)
PRINT FNstrip$(d%);
GOTO calendar.displayed
```

Note$(d%) gets the entire contents of the edit field, which may include several lines. After the program closes the OK button and the edit field, the update.cell subroutine redisplays the cell with the new note (Figure 20-3). Finally, the program restores the date in the upper left corner of the cell.

Changing the Date

Clicking the mouse in the area containing the month and year initiates the following logic.



August 1985

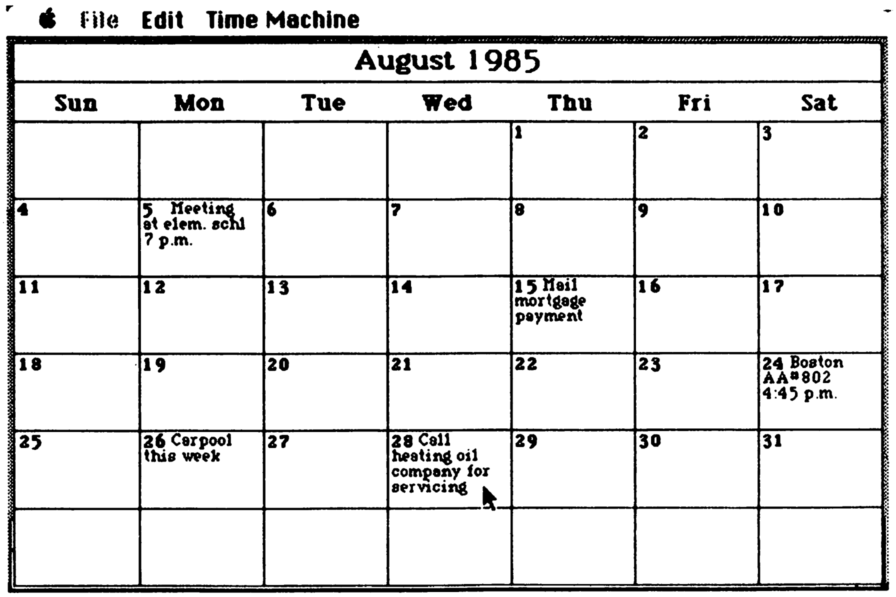
Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15 Mail mortgage payment	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Figure 20-3. Appearance of the note after you press ENTER. The first line is shifted right to make room for the date, and the lines are printed closer together

```
change.date:
LINE (bx%,by%)-(bx1%,by1%),0,bf    :REM erase dialogue area
CALL TEXTSIZE(12)
CALL TEXTFACE(1)
CALL MOVETO(bx%+6,fy%+3)
PRINT "Change month & year (";
PRINT "Mar. "; FNstrip$(base.year%);"- Nov. "; FNstrip$(last.year%);")";
CALL TEXTFACE(0)
cd.loop:
EDIT FIELD 1,m.title$,(ww%*11/16,fy%-9)-(ww%*11/16+144,fy%+6)
LET x%=DIALOG(0)   :REM ignore previous events
WHILE DIALOG(0)<>6
WEND
```

The program creates an edit field as shown in Figure 20-5 and waits for you to enter a new date. The following block evaluates your entry:

```
LET mmyy$=UCASE$(EDIT$(1))
IF mmyy$=nu$ THEN date.err
```

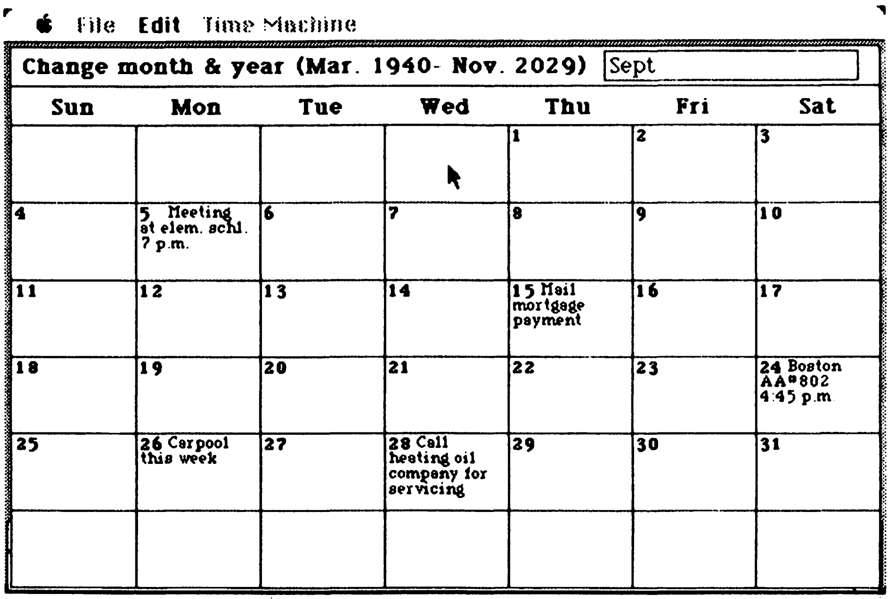


August 1985						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5 Meeting at elem. schl 7 p.m.	6	7	8	9	10
11	12	13	14	15 Mail mortgage payment	16	17
18	19	20	21	22	23	24 Boston AA#802 4:45 p.m.
25	26 Carpool this week	27	28 Call heating oil company for servicing	29	30	31

Figure 20-4. A calendar with assorted notes. Notice that the note for August 28 is four lines long; when entering such a note, the fourth line must be typed first and the insertion point repositioned to the start of the line so that the first three lines may be inserted ahead of the fourth

```
LET valid.month%=no%
LET tried%=0
WHILE NOT valid.month% AND tried%<12
LET tried%=tried%+1
LET valid.month%=INSTR(1,mmyy$,UCASE$( LEFT$(mon$(tried%),3)))<>0
WEND
IF NOT valid.month% THEN date.err
LET new.m%=tried%
LET new.y=0
LET char%=0
WHILE new.y=0 AND char%<LEN(mmyy$)
LET char%=char%+1
LET new.y=INT(VAL(RIGHT$(mmyy$,LEN(mmyy$)-char%+1)))
WEND
```

First the program looks for a valid month in mmyy$ (the string you entered). The program looks only for the first three letters of each



Change month & year (Mar. 1940- Nov. 2029) Sept

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5 Meeting at elem. schl. 7 p.m.	6	7	8	9	10
11	12	13	14	15 Mail mortgage payment	16	17
18	19	20	21	22	23	24 Boston AA#802 4:45 p.m.
25	26 Carpool this week	27	28 Call heating oil company for servicing	29	30	31

Figure 20-5. Screen appearance after you click on the area containing the month and year

month, so you can type **Mar** for March, for example. If you have typed in a valid month, the program looks for a year specification, new.year, inside mmyy$. If a year specification is not found, new.year is set to 0.

Given a month and year setting, the following block ensures that the date is within the acceptable range:

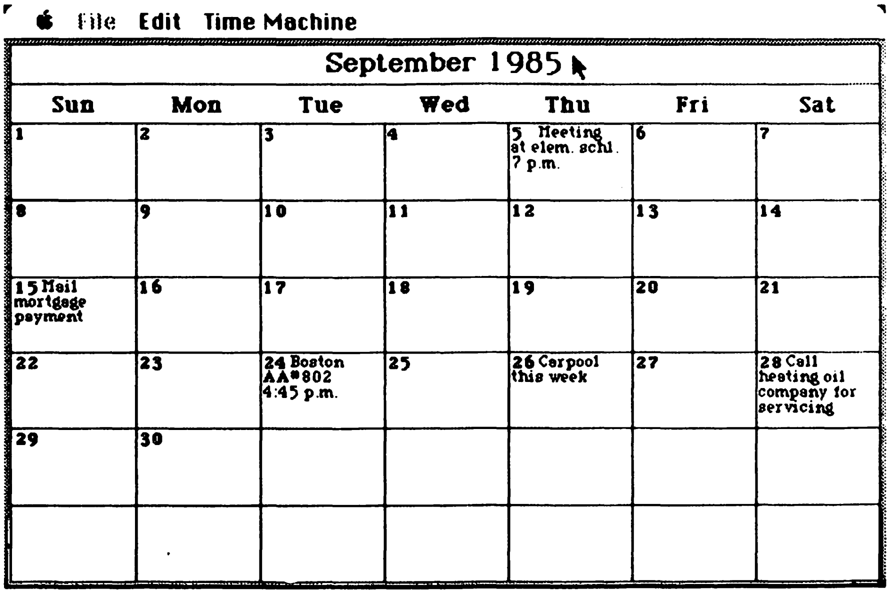
```
IF new.y=0 THEN new.y=y%
IF new.y<base.year% OR new.y>last.year% THEN date.err
IF new.y=base.year% AND new.m%<3 THEN date.err
IF new.y=last.year% AND new.m%>11 THEN date.err
LET date.changed%=(m%<>new.m%) OR (y%<>new.y)
LET m%=new.m%
LET y%=new.y
EDIT FIELD CLOSE 1
IF NOT date.changed% THEN redraw.title
GOSUB new.date
GOSUB redraw.calendar
RETURN
date.err:
```

```
BEEP
GOTO cd.loop
```

If new.year=0 (indicating that a year specification was not found), the program leaves the year at the current setting, y%. Given a new month and year specification, the program performs the calendar calculations (new.date) and redraws the calendar (redraw.calendar). The existing notes are copied into the corresponding dates in the new month (compare Figures 20-6 and 20-4).

If you did not change the month or year, the following lines redraw the title and return to the control loop:

```
redraw.title:
LINE (bx%,by%)-(bx1%,by1%),0,bf    :REM erase title area
CALL TEXTSIZE(14)
CALL TEXTFACE(1)
CALL MOVETO(hm.center%,vm.center%)
PRINT m.title$;
RETURN
```



September 1985

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5 Meeting at elem. schl. 7 p.m.	6	7
8	9	10	11	12	13	14
15 Mail mortgage payment	16	17	18	19	20	21
22	23	24 Boston AA*802 4:45 p.m.	25	26 Carpool this week	27	28 Call heating oil company for servicing
29	30					

Figure 20-6. After you enter a new date, the program draws a calendar for that date but retains the notes from the previously displayed month

—Menu Options

The following blocks handle the Time Machine menu options: Erase notes, Load note file, Save note file, and Print.

Erasing Notes

First, here is the logic to erase notes:

```
erase.notes:
FOR cd%=1 TO 31
LET note$(cd%)=nu$
NEXT cd%
GOSUB redraw.calendar
RETURN
```

All 31 elements of the note are set to null, and the calendar is redrawn.

Saving Notes

The following lines save the currently displayed notes in a disk file:

```
save.notes:
CLS
CALL TEXTSIZE(12)
CALL TEXTFACE(1)
PRINT "SAVE CALENDAR NOTES:"
LET fo$=FILES$(0,"NAME THE OUTPUT FILE")
IF fo$=nu$ THEN sn.done
ON ERROR GOTO sn.err
OPEN fo$ FOR OUTPUT AS 1
FOR cd%=1 TO 31
PRINT#1, LEN(note$(cd%))
PRINT#1, note$(cd%);
NEXT cd%
CLOSE
sn.done:
ON ERROR GOTO 0
GOSUB redraw.calendar
RETURN
```

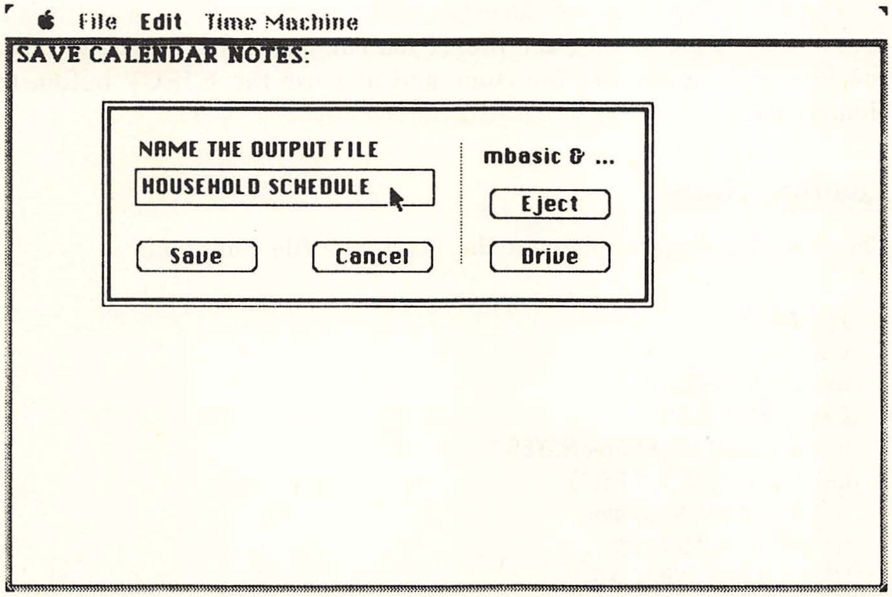


Figure 20-7. Screen appearance during execution of the Save note file command

Figure 20-7 shows the screen appearance during execution of this routine. After the file is saved, the program redraws the calendar and returns to the control loop.

If an error occurs while the notes are being saved, the following lines take over:

```
sn.err:
BEEP
PRINT
PRINT "Can't save notes to:"
PRINT fo$
BUTTON 1,1,"OK",(ww%*13/16,hl%/8)-(ww%*15/16,hl%*7/8),1
LET x%=DIALOG(0)   :REM ignore previous events
WHILE DIALOG(0)<>1
WEND
CLOSE
BUTTON CLOSE 1
RESUME sn.done
```

The most likely source of an error while you are saving a note file is a lack of disk space to store the file. If you run out of disk space, you can re-execute the Save note file command and use the EJECT button to change disks.

Loading Notes

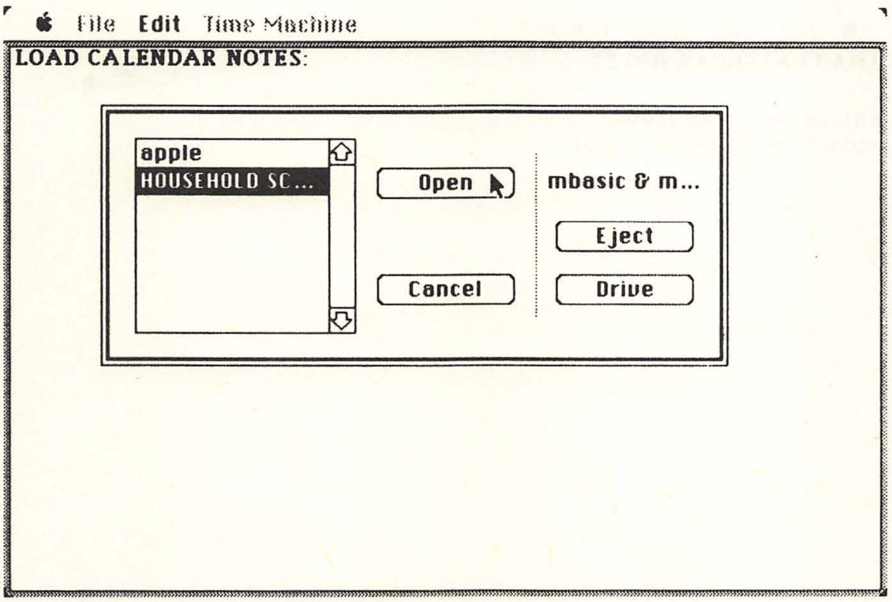
The following lines implement the Load note file command:

```
load.notes:
CLS
CALL TEXTSIZE(12)
CALL TEXTFACE(1)
PRINT "LOAD CALENDAR NOTES:"
LET fi$=FILES$(1,"TEXT")
IF fi$=nu$ THEN ln.done
ON ERROR GOTO ln.err
OPEN fi$ FOR INPUT AS 1
FOR cd%=1 TO 31
LET note$(cd%)=nu$
INPUT#1, note.length%
IF note.length%=0 THEN next.note:
LET note$(cd%)=INPUT$(note.length%,#1)
next.note:
NEXT cd%
CLOSE
ln.done:
ON ERROR GOTO 0
GOSUB redraw.calendar
RETURN
```

Figure 20-8 shows the screen appearance when the program asks you to name the input file; you must specify a file that was created with the Save note file command.

In the case of an error during the loading of the file, the following lines take over:

```
ln.err:
BEEP
PRINT
PRINT "Can't load notes from:"
PRINT fi$
```



LOAD CALENDAR NOTES:

Figure 20-8. Screen appearance during execution of the Load note file command

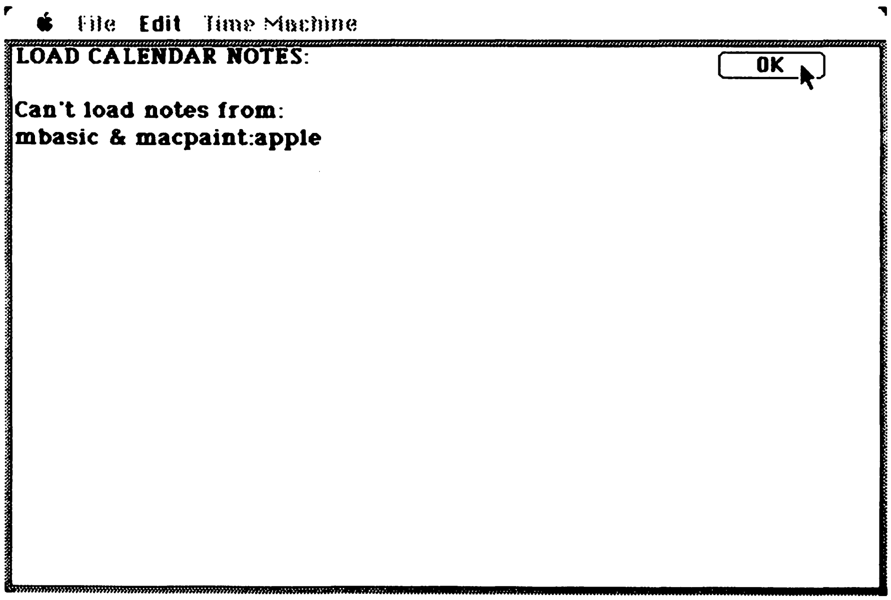
```
BUTTON 1,1,"OK",(ww%*13/16,hl%/8)-(ww%*15/16,hl%*7/8),1
LET x%=DIALOG(0)    :REM ignore previous events
WHILE DIALOG(0)<>1
WEND
CLOSE
BUTTON CLOSE 1
RESUME ln.done
```

Figure 20-9 shows the kind of error message displayed by the load-error routine. Errors may be caused by an invalid file format (the file was not created with this program) or by some other disk input error.

Printing the Calendar

The following lines copy the calendar from the screen to the printer:

```
print.cal:
CALL HIDECURSOR
```



LOAD CALENDAR NOTES:

Can't load notes from:
mbasic & macpaint:apple

Figure 20-9. Error notice when a file cannot be loaded

```
LCOPY
CALL SHOWCURSOR
RETURN
```

Calendar Calculations

Given a month m% and a year y%, the next few program blocks map the month into the 6 × 7 calendar grid.

The first block converts the month and year into relative offsets from the base date:

```
new.date:   :REM 20-18
FOR cn%=1 TO 42
LET cal%(cn%)=0
NEXT cn%
LET y1%=y%-base.year%   :REM years since base year
IF m%<=2 THEN before.march   :REM March is base month
LET m1%=m%-3
GOTO relative.date.ready
```

```
before.march:
LET m1%=m%+9
LET y1%=y1%-1
```

First the calendar map, cal%( ), is erased. Then the program calculates y1% and m1% (the number of complete years and months between the selected year and the base year). The following equations illustrate the calculation process for August 1985, using a base date of March, 1940:

```
 1985 years, 8 months
-1940 years, 3 months
---------------------
   45 years, 5 months
```

Expressing this example using the program's variables, year y%=1985 and month m%=8, while the relative year y1%=45 and the relative month m1%=5.

Given a relative date, the following block calculates the days elapsed from March 1 of the base year to March 1 of the selected month and year:

```
relative.date.ready:
LET jd%=INT(1461*y1%/4)+(153*m1%+2)\5   :REM jd%=days from base
    date
LET wd%=(jd%+first.of.march%) MOD 7
LET leap.yr%=((y% MOD 4=0) AND (y% MOD 100<>0)) OR (y% MOD 400=0)
LET ld%=mlen%(m%)
IF leap.yr% AND m%=2 THEN LET ld%=29   :REM adjust Feb. in a leap year
FOR d%=1 TO ld%
LET cal%(d%+wd%)=d%
NEXT d%
RETURN
```

Using the total elapsed days jd%, the program calculates the day of the week for March 1 in the selected year (LET wd%=(jd%+first.of.march%) MOD 7). The program also gets the correct number of days for the current month and adjusts for a leap year when necessary.

Given the day of the week for March 1 and the number of days in the month, the program maps the calendar days into the 6 × 7 grid (LET cal%(d%+wd%)=d%). For instance, cal%(3)=1 indicates that the first of the month is the third element of cal%( ).

—Redrawing the Calendar

The next three blocks draw the calendar on the screen. Here is the block that draws the lines and the month and year headings:

```
redraw.calendar:
CLS
FOR rule%=1 TO 6
LINE (0,rule%*cl%)-STEP(ww%,0)   :REM horizontal
LINE (rule%*cw%,cl%)-STEP(0,wl%-cl%)   :REM vertical
NEXT rule%
LINE (0,hl%)-STEP(ww%,0)   :REM rule under heading
CALL TEXTSIZE(14)   :REM large for heading
CALL TEXTFACE(1)   :REM bold
LET m.title$=mon$(m%)+STR$(y%)
LET hm.center%=(ww%-WIDTH(m.title$))\2
LET vm.center%=(hl%-14)\2+11
CALL MOVETO(hm.center%,vm.center%)
PRINT m.title$;
LET vw.center%=hl%+(hl%-12)\2+12
```

This block displays day headings:

```
CALL TEXTSIZE(12)
FOR dn%=1 TO 7
LET w.title$=wday$(dn%)
LET hw.center%=(dn%-1)*cw%+(cw%-WIDTH(w.title$))\2
CALL MOVETO(hw.center%,vw.center%)
PRINT w.title$;
NEXT dn%
```

And finally, this block displays the dates and the corresponding notes:

```
CALL TEXTSIZE(9)
CALL TEXTFACE(1)
FOR d%=1 TO ld%
LET cn%=d%+wd%
LET dx%=2+((cn%-1) MOD 7)*cw%
LET dy%=10+((cn%-1)\7+1)*cl%
CALL MOVETO(dx%,dy%)
PRINT FNstrip$(d%);
```

```
NEXT d%
CALL TEXTFACE(0)
FOR d%=1 TO ld%
LET cn%=d%+wd%
LET dx%=2+((cn%-1) MOD 7)*cw%
LET dy%=9+((cn%-1)\7+1)*cl%
GOSUB update.cell
NEXT d%
CALL TEXTFACE(1)
RETURN
```

The update.cell subroutine writes any notes in the appropriate cell for day d%. Here is the subroutine:

```
update.cell:
IF note$(d%)=nu$ THEN nothing.here
CALL MOVETO(dx%+16,dy%)   :REM leave room for date
LET last.cr%=0
LET next.cr%=INSTR(last.cr%+1,note$(d%),CHR$(13))
WHILE next.cr%<>0
IF next.cr%-last.cr%=1 THEN empty
PRINT MID$(note$(d%),last.cr%+1,next.cr%-last.cr%-1);
empty:
LET dy%=dy%+9
CALL MOVETO(dx%,dy%)
LET last.cr%=next.cr%
LET next.cr%=INSTR(last.cr%+1,note$(d%),CHR$(13))
WEND
PRINT RIGHT$(note$(d%),LEN(note$(d%))-last.cr%);
nothing.here:
RETURN
```

Notice that in printing the first line of the notes, the program shifts the text 16 graphics points to the right (CALL MOVETO (dx%+16,dy%)). This prevents the first line from being printed on top of the date.

—Testing and Using the Program

After entering the entire program and carefully checking a printout of your work, run the program. You should be able to duplicate the screens shown in this chapter.

Remember that the first line of the note is always shifted to the right to make room for the date, so you should keep this line shorter than the rest. To start a new line in a note, press ENTER or RETURN.

Look at the note for August 28 in Figure 20-4. It contains four lines. When you try to type in such a note, you'll find that the fourth line is hidden from view (the edit box isn't large enough to hold four lines).

In order to enter a fourth line, you must type in the fourth line first and then move the insertion point to the beginning of the line and press ENTER. Now type in the first three lines. The fourth line will move down in the edit box until it is no longer visible. However, as soon as you click the OK button and the cell is redisplayed, you'll be able to see all four lines.

Practice this until you can get similar results. Just remember that the note lines are loosely spaced while you're typing them in; they don't take on their final appearance until after you click the OK button.

Note: If you interrupt the program by typing COMMAND-. or if an error stops the program, you must type in the following command to restore the normal top-bar menu:

```
MENU RESET
```

Index

10[redacted]20394

[redacted]

[illegible]

MACINTOSH™ PROGRAM FACTORY™

Here's an exciting collection of puzzles and games that will keep you and your Macintosh™ entertained for hours.

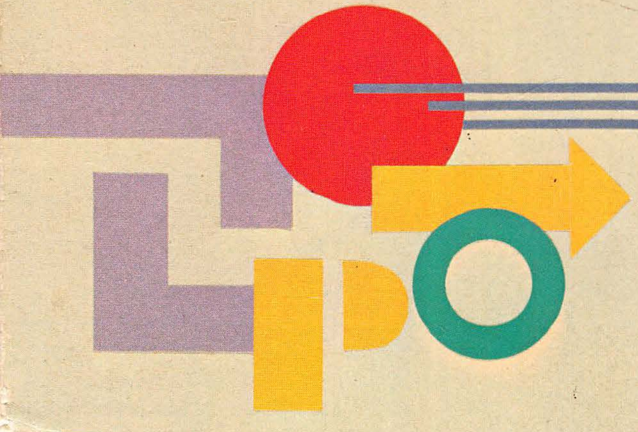
George Stewart has assembled more than 20 programs, some adapted from his articles in *Popular Computing*, "The Program Factory.™" These take full advantage of all of the Mac's special features.

Key these programs in and enjoy
—Crossword puzzles
—Codebreaker
—Billiard Practice
—Secret Messages
—Time Machine
...and more.

Beginners can quickly access these programs, while experienced users who want to understand how the programs work can learn from the explanations that accompany each project.

Ideal for all ages, **Macintosh™ Program Factory™** provides hours of fun and learning.

- **Macintosh is a trademark of Apple Computer, Inc.**
- **Program Factory is a trademark of the author, George Stewart.**



ISBN 0-07-881175-9