Osborne McGraw-Hill

THE C-64® PROGRAM FACTORY™

George Stewart

The C-64®
Program Factory™

The C-64® 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 C-64® Program Factory™

1234567890 DODO 8987654

ISBN 0-88134-150-9

Judy Ziajka, Acquisitions Editor
Karen Hanson, Project Sponsor
Harry Wong, Technical Reviewer
Ted Gartner, Copy Editor
Judy Wohlfrom, Text Design
Donna Behrens, Composition
Yashi Okita, Cover Design

To my wife Marguerite, who encouraged me to write this book (and designed the office in which I wrote it).

Table Of Contents

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 Susan Sitkin for her help in keeping this work on schedule.

—Introduction

Your Commodore 64 has a tremendous amount of power hidden away inside. And it doesn't take a spreadsheet or word processing program to unleash it, either. The programs presented in this book will put your C-64 computer to work right now as a puzzle generator, entertainer, teacher, creative assistant, and general helper.

Most of the programs let *you* contribute something as well—so that the program and 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 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 step-by-step method of presentation and many of the programs are adapted from the author's Program Factory series appearing each month in *Popular Computing* magazine. However, all of the programs in this book (the new ones as well as the adaptations) have been designed or redesigned specifically for computers that run Commodore BASIC, taking advantage of graphics, sound, and disk file capabilities wherever possible.

Contents of the Book The 20 programs fall into five categories:

- *Puzzles* generated by the computer and printed on paper. The finished puzzles may be used without further reference to the computer.
- *Games and simulations* for one or more persons; the computer plays an active role in the games.
- *Education and self-improvement* projects to teach and exercise your mind on just about any subject you can imagine.
- *Creativity and art* projects—the computer becomes a way of extending your imagination.
- *Handy tools*—programs for use around the computer-age home or office.

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 blocks of approximately 10-25 lines, each block accompanied by some explanatory comments. Another section gives hints and tips for using the program, with suggestions for program changes in some cases.

Computer Requirements To run these programs you'll need a Commodore 64 computer with Commodore BASIC built-in. Many of the programs assume you have a printer attached as device number 4 and a disk drive attached as device number 8. If you don't have one of these attachments, simply skip the program options that require the disk or printer; you can still use the program with the minimal C-64 system (keyboard and television set).

Suggestions for Entering the Programs Type slowly and carefully when entering the program lines. Check your work as you go along. Before trying to run the program, save it on disk (if available) 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 little bit out of place and a useless mutation may result.

Be especially careful to distinguish the letter O from the number 0 and the letter I 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, and it's important that you type in a null string when that is called for.

After making a visual, line-for-line check of your program, try to run it. To determine whether your version is working or not, compare your results (shown on your computer screen) with the photographs or sample printouts given in the chapter.

Program Disks All of the programs in this book are available on 5-1/4 inch disk. For prices and details write to the author, care of: Commodore 64 Factory, POB 137, Hancock, New Hampshire 03449.

Chapter 1

Making Mazes

If you like solving mazes, you'll find making them even more challenging. But telling your computer how to make mazes is the most interesting challenge of all.

A maze is like the floor plan of a house with only one entrance and one exit. In making a maze, the first step is to picture the house with walls dividing it into rooms, but without any doors between the rooms or in or out of the house. Next you add doors until you have one path between any two rooms in the house. Finally, you add an entrance and an exit anywhere you like.

Figure 1-1 shows a 4×4 maze. Verify for yourself that there is only one path between any two rooms. Try closing the entrance and exit and opening new ones: you still will have a perfectly good maze because its internal structure always remains the same.

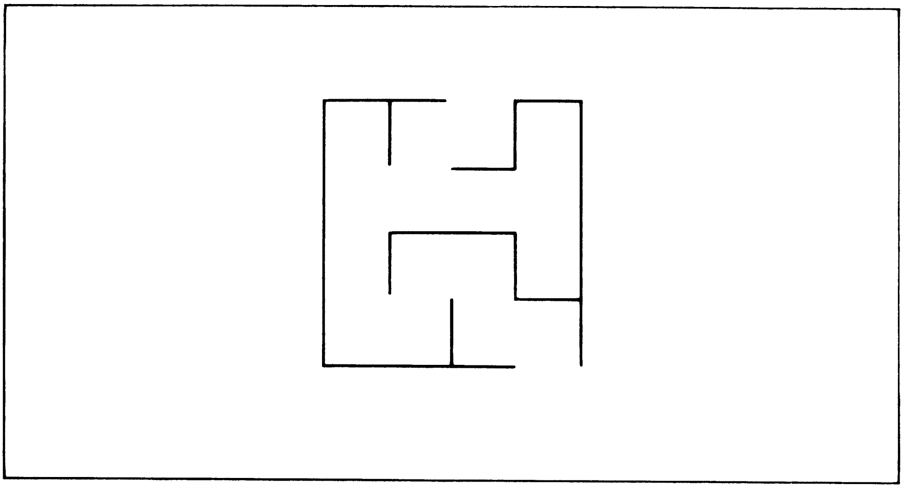


Figure 1-1. 4 × 4 maze

—Construction Procedure

A maze is divided into three types of rooms while it is being constructed:

- Living quarters (LQ): rooms that are connected by doorways.
- 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 abbreviations LQ, PE, and US will indicate variable storage locations in the program shown later in this chapter.

Here are the steps for building a maze (refer to Figure 1-2):

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).

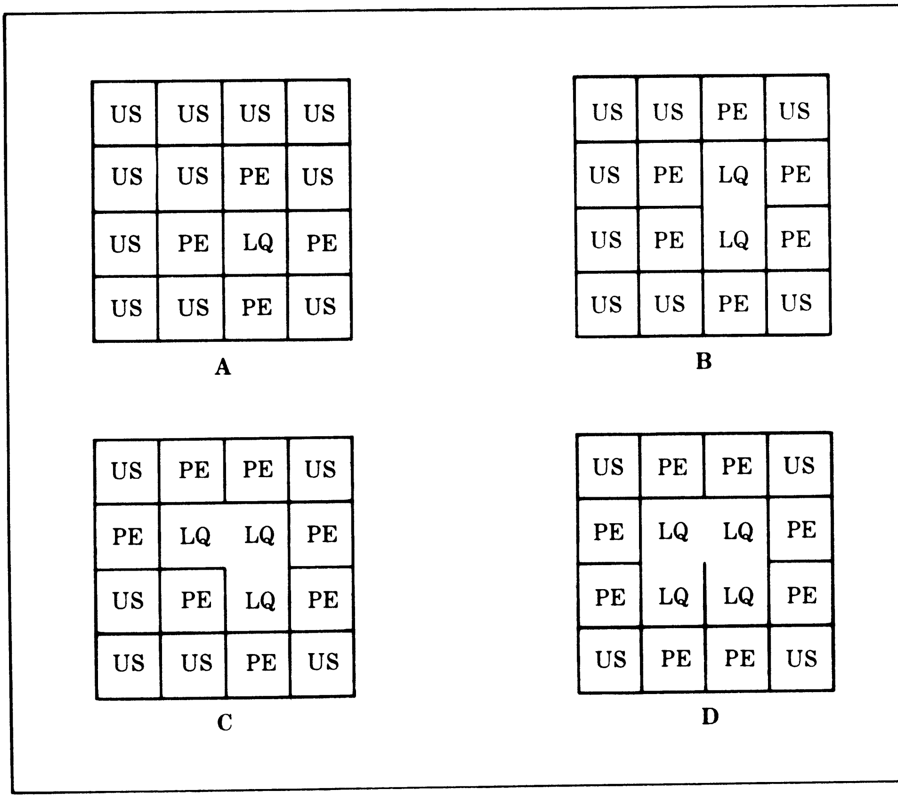


Figure 1-2. First four steps in creating a maze

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.

You can verify that this procedure works by using it to create a 4 × 4 maze on paper. Figure 1-2 shows a few steps in the process.

—How the Program Stores the Maze

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

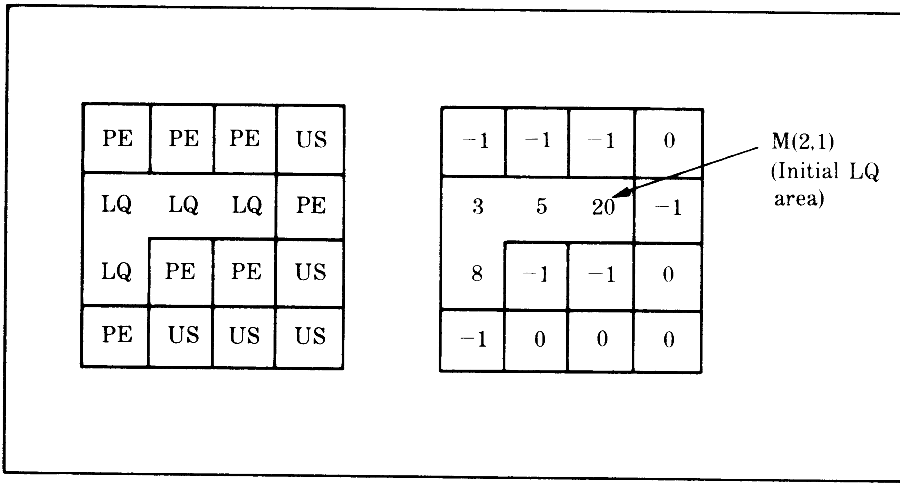


Figure 1-3. Maze under construction with LQ/PE/US and with numerical coding

Figure 1-3 shows a maze under construction using the LQ/PE/US coding system and again with the computer's numerical coding system. All US rooms are represented with 0. All PE rooms are represented as −1. All LQ rooms are represented by a positive number from 1 through 15 with one exception. The very first LQ room is a special case because it has no doors. Thus, it has a door code of 0, the same code as a US room. To distinguish this first LQ room from unused space, 16 is added to its initial door code.

The number of an LQ room is calculated by assigning the numbers 1, 2, 4, and 8 to its four walls, as shown in Figure 1-4, and taking the sum of all walls with doors.

—The Program

Throughout this book, the programs will be presented in short logical blocks to keep the explanations short and clear. However, you can just enter the listings and return to the explanations at a later time.

Setting Up the Program Constants

The first block initializes the random number generator (so you'll get different mazes each time you run the program) and sets up certain constant values:

```
10 INPUT "ENTER A RANDOM NUMBER ";X
20 X=RND(-ABS(X))
30 CLR
40 CS$=CHR$(147): REM CLEAR SCREEN
70 S6$=CHR$(15): REM 6 LINES/INCH PRINTER CODE
80 S9$=CHR$(8): REM 9 LINES/INCH PRINTER CODE
130 W$=CHR$(166): REM SOLID BLOCK CHARACTER
140 OP$=" ": REM ONE SPACE IN QUOTES
```

Line 30 erases the previous contents of all variables and arrays; after creating a maze, the program can start over at this line (if you ask for a new maze).

Lines 70 and 80 store printer control codes for use with the Commodore MPS-801 printer (or other compatible models). Outputting S6$ to the printer selects a line spacing of 6 lines per inch; outputting S9$ selects a line spacing of 9 lines per inch. The latter spacing allows the printing of mazes in which there are no gaps in the vertical walls.

Lines 130 and 140 store the characters used to represent walls and open areas within the maze. W$ is a solid block and OP$ is a single space.

If you don't want to use the solid block for walls (or if your printer can't produce that character), set S6$ and S9$ equal to the null string (a pair of double quotes with no spaces in between them), and set W$ equal to a single "X".

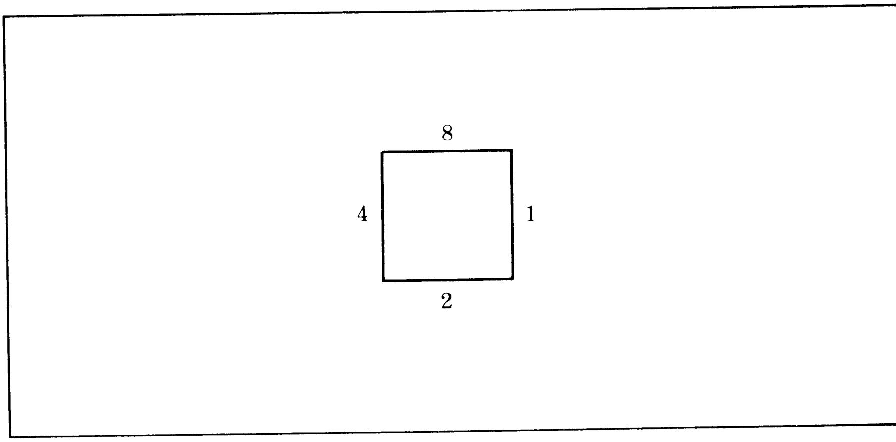


Figure 1-4. The numerical codes used to represent which walls of a cell are open

Defining the Size of the Maze

The following lines request the maze dimensions and then set up the necessary arrays:

```
150 PRINT "ENTER MAZE LENGTH AND WIDTH"
160 PRINT "(LL,WW) ";
170 INPUT RX,CX
173 RX=INT(RX)
175 CX=INT(CX)
180 FX=2/3*RX*CX
190 DIM M(RX,CX),FR(FX),FC(FX),VU(4)
200 N=0
```

In line 180, FX is the maximum number of planned expansion cells (PE) allowable based on the maze dimensions RX and CX. M(,)stores the maze, and FR( ) and FC( ) store the row and column locations of the PE cells. VU( ) stores the contents of rooms adjacent to the most recently added LQ room. N stores the number of PE cells (0 when we first begin).

Identifying the First Room

The program selects the first room of the living quarters (LQ) by randomly picking a row R and a column C.

```
210 R=INT(RND(1)*RX)+1
220 C=INT(RND(1)*CX)+1
230 C0=C
240 R0=R
250 M(R,C)=16
```

In line 250, M(R,C) gets the special value of 16, which indicates it is the first LQ room. As doors are added to this room, it will take on values from 17 to 31.

Identifying the PE Areas

It's time to identify all the rooms. To do this, the program must look at the four adjacent rooms (left, right, up, and down).

```
260 GOSUB 1520
270 IF VU(1)<>0 THEN 320
280 N=N+1
```

```
290 FR(N)=R
300 FC(N)=C+1
310 M(R,C+1)=-1
320 IF VU(2)<>0 THEN 370
330 N=N+1
340 FR(N)=R+1
350 FC(N)=C
360 M(R+1,C)=-1
370 IF VU(3)<>0 THEN 420
380 N=N+1
390 FR(N)=R
400 FC(N)=C-1
410 M(R,C-1)=-1
420 IF VU(4)<>0 THEN 470
430 N=N+1
440 FR(N)=R-1
450 FC(N)=C
460 M(R-1,C)=-1
```

The subroutine call at line 260 gets the view from the current room and stores it in array VU( ). This facilitates the updating of the PE list. VU(1) through VU(4) list the contents of the rooms to the right, below, to the left, and above, respectively. If VU( ) refers to a room that is beyond the boundaries of the maze, its value is set to −1.

The program checks the contents of all four views VU(1) through VU(4). We'll look at lines 270-310 as an example. These lines check VU(1).

Whenever VU(1) is 0 (line 270), indicating a room with no doors, the program adds 1 to the PE counter N (line 280). The PE counter stores the row and column address of the room that is referenced by VU(1) (lines 290-300). Finally, the room is marked as a PE room in line 310.

Checking for New Expansion

After checking all four views, the program continues. First it tests the value of the PE counter, N.

```
470 IF N=0 THEN 840
```

If N is 0, there are no more PE rooms, so the program advances to line 840. However, if there are PE rooms remaining, the program randomly selects one to become the newest addition to the LQ area.

```
490 F=INT(RND(1)*N)+1
500 R=FR(F)
510 C=FC(F)
520 GOSUB 1520
530 P=INT(RND(1)*4)+1
540 IF VU(P)<=0 THEN 530
550 M(R,C)=2↑(P-1)
560 FR(F)=FR(N)
570 FC(F)=FC(N)
580 N=N-1
590 ON P GOTO 600,660,720,790
600 M(R,C+1)=M(R,C+1) OR 4
650 GOTO 260
660 M(R+1,C)=M(R+1,C) OR 8
710 GOTO 260
720 M(R,C-1)=M(R,C-1) OR 1
770 GOTO 260
790 M(R-1,C)=M(R-1,C) OR 2
830 GOTO 260
```

Lines 490-510 select a room from the PE list. This room, M(R,C), shares at least one common wall with the LQ area. The program must select a wall to remove so M(R,C) can become part of the LQ area.

The subroutine call at line 520 gets the view from M(R,C). Line 530 randomly selects a direction P (right, down, left, up). If the room in that direction is in the LQ area (line 540), the wall between the two rooms is removed.

This is a two-step process: line 550 stores the open-door code in M(R,C); but the open-door code of the other room (the "destination room") also must be updated. Lines 590-830 update this code by using the OR function.

Look at lines 600 and 650 as an example. These lines operate when P=1, indicating that the target room is to the right of M(R,C) and giving it an array address of M(R,C+1). The program computes the value M(R,C+1) OR 4, which opens the appropriate door in the target room without affecting any of its other four doors.

Lines 660-830 handle P=2, P=3, and P=4 in an analogous manner. In every case, the program jumps back to line 260, using M(R,C) as the new LQ room.

Locating the Entrance and Exit

This repetitive process ends when no more PE rooms remain (when N=0 in line 470). The following lines select entrance and exit cells.

```
840 M(R0,C0)=M(R0,C0) AND 15
890 SC=INT(RND(1)*CX)+1
900 EC=INT(RND(1)*CX)+1
910 M(1,SC)=M(1,SC) OR 8
960 M(RX,EC)=M(RX,EC) OR 2
```

Recall that the first LQ room, M(R0,C0), receives a special code of 16. During the course of the maze construction, additional open-door codes are added to this value, depending on which doors of M(R0,C0) are opened. Line 840 converts the special code into a standard code ranging from 1 to 15.

Lines 890 and 900 select entrance and exit cells for the maze. Lines 910 and 960 remove the outer wall of the entrance cell.

The maze is complete as far as the computer's digital logic is concerned. Now the program makes it visible, by printing it on the display or outputting it to a printer.

Printing the Maze

The following lines let you select the output device:

```
970 DV=1
980 PRINT "SELECT OUTPUT DEVICE"
990 PRINT "1=DISPLAY  2=PRINTER"
1000 INPUT DV
1010 IF DV<>1 AND DV<>2 THEN 980
1020 IF DV=1 THEN 1050
1030 OPEN 1,4
1040 CMD 1
```

The program assumes that "device 4" is your printer. If you have another device number assigned to your printer, change "4" to the correct value in line 1030.

```
1050 PRINT CS$;S6$;: REM CLEAR SCREEN,
     SELECT 6 L/IN PRINTING
1090 FOR R=1 TO RX
1100 FOR C=1 TO CX
1110 PRINT W$;
1120 IF (M(R,C) AND 8) <> 0 THEN 1190
1170 PRINT W$;
1180 GOTO 1200
1190 PRINT OP$;
1200 NEXT C
1210 PRINT W$;S9$: PRINT S6$;: REM SELECT 9 L/IN
     PRINTING THEN RETURN TO 6 L/IN
```

```
1220 FOR C=1 TO CX
1230 IF (M(R,C) AND 4) <> 0 THEN 1300
1280 PRINT W$;
1290 GOTO 1310
1300 PRINT OP$;
1310 PRINT OP$;
1320 NEXT C
1330 PRINT W$;S9$: PRINT S6$;: REM SELECT 9 L/IN
     PRINTING THEN RETURN TO 6 L/IN
1340 NEXT R
1350 FOR C=1 TO CX
1360 PRINT W$;
1370 IF (M(RX,C) AND 2) <> 2 THEN 1440
1420 PRINT OP$;
1430 GOTO 1450
1440 PRINT W$;
1450 NEXT C
1460 PRINT W$;S9$: PRINT S6$;: REM SELECT 9 L/IN
     PRINTING THEN RETURN TO 6 L/IN
1470 IF DV=2 THEN PRINT#1,: CLOSE 1
1490 INPUT "SELECT: 1-REPEAT  2-NEW MAZE  3-END";CT
1495 IF CT<1 OR CT>3 THEN 1490
1500 ON CT GOTO 970,30,1510
1510 END
```

Line 1050 erases the screen (if the CRT was selected) and activates the standard printer character set (if the printer was selected).

In printing the maze, the program starts at row 1 and counts up to row RX (the bottom row). For each row, it counts from column 1 to column CX, the right-hand column.

Printing a row of cells requires two lines on the display or printer: one for the top of the cell, consisting of horizontal walls and openings, and one for the middle of the cell, consisting of vertical walls, doors, and spaces. Lines 1100-1210 print the horizontal walls; lines 1220-1340 print the vertical walls. Lines 1350-1460 print the last row of horizontal walls, completing the maze.

Here's the subroutine that gets the four views (right, below, left, above) from a cell and stores them in array elements VU(1) through VU(4).

```
1520 IF C<>CX THEN 1550
1530 VU(1)=-1
1540 GOTO 1560
1550 VU(1)=M(R,C+1)
1560 IF R<>RX THEN 1590
1570 VU(2)=-1
```

```
1580 GOTO 1600
1590 VU(2)=M(R+1,C)
1600 IF C<>1 THEN 1630
1610 VU(3)=-1
1620 GOTO 1640
1630 VU(3)=M(R,C-1)
1640 IF R<>1 THEN 1670
1650 VU(4)=-1
1660 GOTO 1680
1670 VU(4)=M(R-1,C)
1680 RETURN
```

—Suggestions for Using the Program

Try the maze with small dimensions (4 × 4, 5 × 6, and so on). Direct all output to the display (select slot 0) to speed the debugging process. You can use the following subroutine to obtain a printout of the maze at any time during construction:

```
1740 FOR I=1 TO RX
1750 FOR J=1 TO CX
1760 PRINT M(I,J);TAB(J*3);
1770 NEXT J
1780 PRINT
1790 NEXT I
1800 PRINT
1810 RETURN
```

Add GOSUB 1740 at strategic points in your program. For example,

```
475 GOSUB 1740
```

is a good idea, because it will give you a printout each time the program prepares to add a room to the living quarters.

After debugging the program, delete line 475 and lines 1740-1810.

The printed maze is composed of X's for walls and blanks for doors and spaces. Use this formula to calculate the space required for a maze of dimensions RX by CX:

$$\text{Width} = 2 \times \text{RX} + 1$$
$$\text{Length} = 2 \times \text{CX} + 1$$

Figure 1-5 shows a maze created on the Commodore MPS-801 printer with block graphics at 9 lines per inch. The Commodore 64 took several minutes to create the maze (not including printing time).

Although a Commodore printer can produce mazes as large as 39 columns in width, mazes wider than 19 columns or taller than 11 rows will either appear scrambled or will scroll off the C-64's 40-column

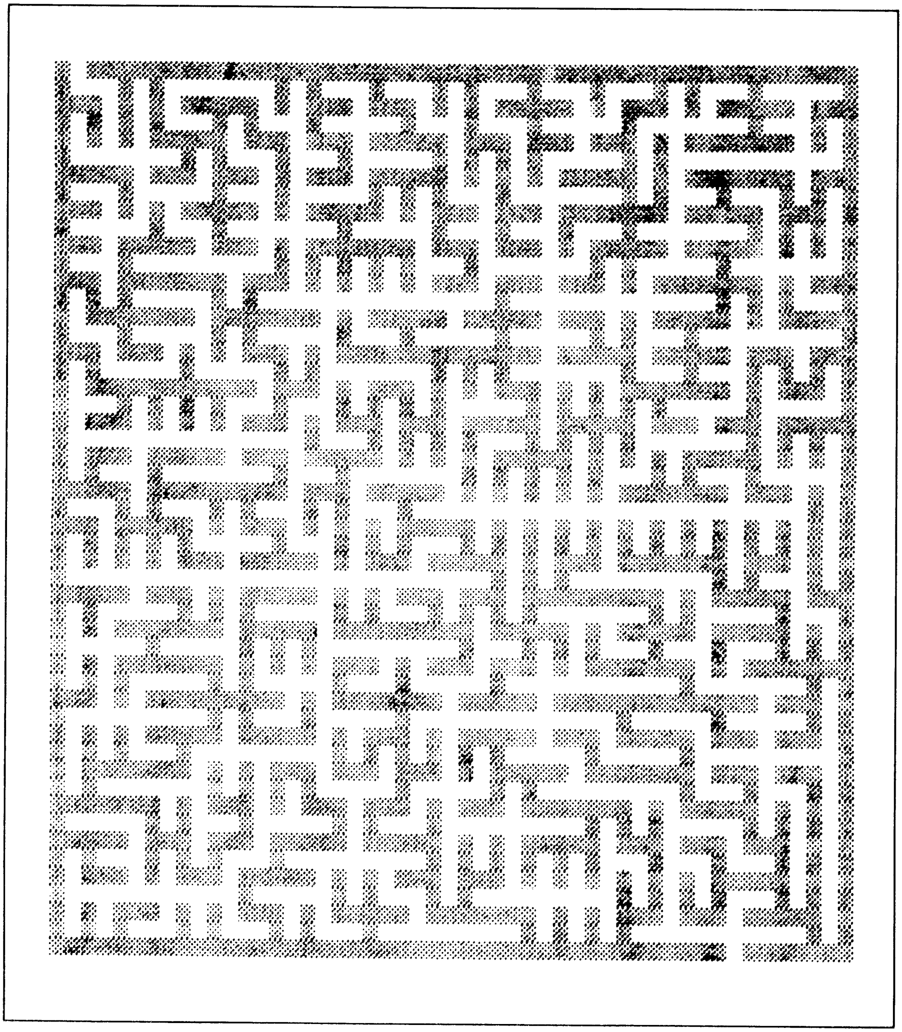


Figure 1-5. A completed maze printed with block graphics at 9 lines per inch

screen. One solution is to limit the size of your maze by rejecting large row or column sizes with the following line:

```
176 IF (RX>11) OR (CX>19) THEN 150
```

When an invalid row or column size is entered, line 176 will make the user reenter the row and column dimensions for the maze. If you also want to protect against invalid input of zero or negative numbers for row or column sizes, you could try the following version of line 176:

```
176 IF (RX<1) OR (CX<1) OR (CX>19) OR (RX>11) THEN 150
```

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

Chapter 2

Hidden Words

This program generates hidden-word puzzles that are more challenging and entertaining than those you're likely to find in newspapers or game magazines.

The puzzles are more fun because you choose the words that are hidden and more challenging because words can be spelled in any of eight directions (most versions of this puzzle use only four directions).

Figure 2-1 shows a sample puzzle created by the program. The solution to the puzzle is given in Figure 2-2.

Depending on the size of the puzzle grid and the vocabulary you choose, the program could need from five minutes to more than an hour to generate each puzzle. So if you're thinking of using puzzles as gifts or party favors, don't wait until the last minute to start your computer.

—How the Program Creates the Puzzle

To generate a hidden-word puzzle, the program must first create an array representing the puzzle. It then tries to fit all the words into the array, filling the remaining spaces with randomly chosen letters. The words may be written in any one of the eight directions.

Animals

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

Figure 2-1. The names of 25 animals are hidden among the letters

The process involves nine steps:

1. The program creates a two-dimensional array and stores a hyphen (-) in each array location or cell.
2. The program creates a randomly ordered list of all the cells, which it uses to examine each cell in turn.
3. If the cell under examination contains a letter, the program moves to the next cell in the list. If the cell contains a hyphen, the program attempts to fit a word into a path that intersects the cell in one of the eight directions.
4. The program starts with the *major path*—the path that touches a border of the puzzle at each end.
5. The program tries to find a word that fits the path: the word length must be the same and the letters must coincide with any letters already filled in along that path. If the program finds a matching word, it fills in all the cells along the path and returns to Step 3 using the next cell in the list.

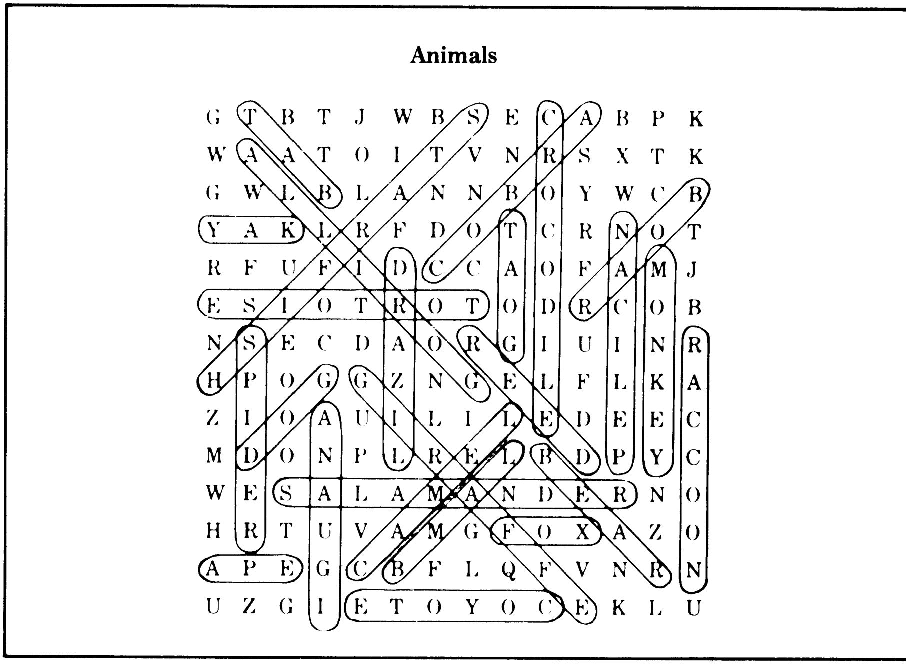


Figure 2-2. Solution to Figure 2-1

6. If none of the words fits, the program establishes a *subpath* in the same direction and returns to Step 5.
7. After trying all subpaths without finding a match, the program shifts directions (moving clockwise) and returns to Step 4.
8. After trying all eight directions without matching, the program leaves that cell blank, selects the next cell in the list, and continues at Step 3.
9. When all cells have been examined, the computer fills all the empty cells with randomly chosen letters, completing the puzzle.

To see this series of steps more graphically, refer to the partially completed puzzle in Figure 2-3.

Refer to Figure 2-4. Suppose the computer starts with direction 6 (northwest): ---* RU. To fill the pathway, the computer must find a six-letter word ending in RU. If it can't, the computer tries a shorter path in the same direction. The next path contains five cells: ---* R. The computer looks for any five-letter word ending in R.

The program continues in this fashion until all cells have been examined and filled in.

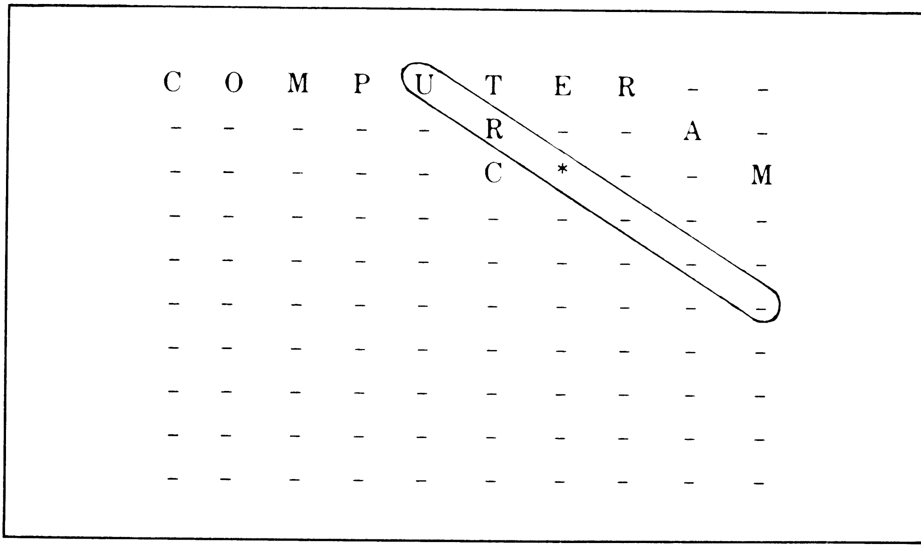


Figure 2-3. A puzzle under construction: hypens indicate empty cells; an asterisk marks the current cell, and the current pathway is circled

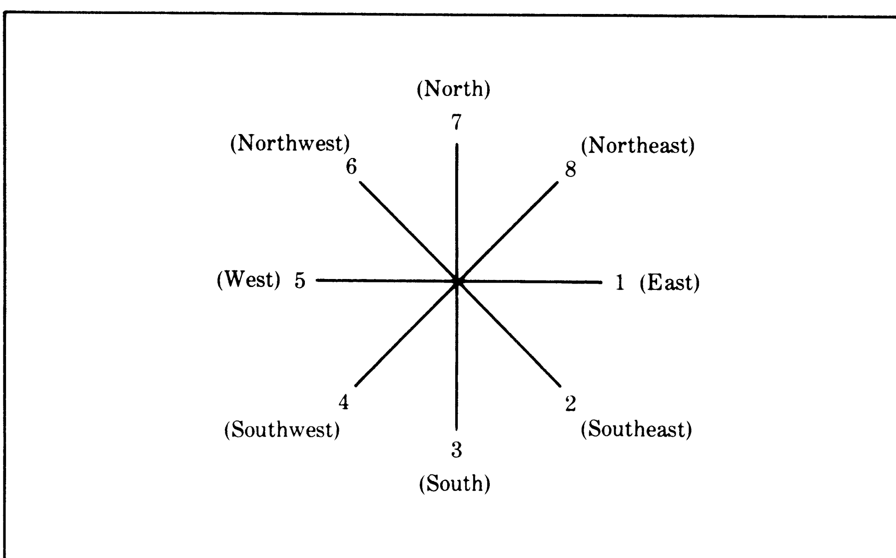


Figure 2-4. Words can run in any of these eight directions

—The Program

The program is presented in logical blocks. Type them in as you read along.

Storing Rows, Columns, and Other Constants

The first block sets up the program constants.

```
5 INPUT "ENTER A RANDOM NUMBER ";X
7 X=RND(-ABS(X))
10 MR=6
20 MC=6
30 NC=MR*MC
40 SP$="-"
50 MK$="*"
60 DC$="+"
70 NW=0
80 READ WD$
90 IF WD$="/" THEN 120
100 NW=NW+1
110 GOTO 80
120 DIM M$(MR,MC),WD$(NW),D(8,2),SQ(NC),WU(NW),
    WQ(NW)
130 RESTORE
140 FOR I=1 TO NW
150 READ WD$(I)
160 NEXT I
170 READ WD$
180 DATA BASIC,NEW,NEXT,PRINT,CURSOR
190 DATA DISK,RUN,STOP,HOME,BIT,BYTE
200 DATA BUS,BUG,REM,RAM,ROM,/
210 FOR D=1 TO 8
220 READ D(D,1),D(D,2)
230 NEXT D
240 DATA 0,1,1,1,1,0,1,-1,0,-1,-1,-1,-1,0,-1,1
```

Lines 5 and 7 set the random number generator. Type a different number each time you run the program for a different word arrangement.

MR is the number of rows in the grid. MC is the number of columns. Change these values to suit your preference. The computed value NC is the number of cells in the grid. How large should you make the grid? Experiment to find out what size gives you the best results. Here are a few guidelines:

- The larger the grid with respect to the word list, the more difficult the puzzle will be. However, a smaller grid with plenty of words packed in makes the puzzle more interesting.

- At least one of the grid's dimensions, including the diagonal, must be large enough to accommodate the longest word in the word list.
- To improve the chances of fitting in all your words, choose MR and MC so that the number of cells in the grid (MR × MC) is 25 to 50 percent greater than the total number of letters in the word list. For example, if your longest word is 10 characters and the word list consists of a total of 100 characters, you might use MR=12 and MC=12 for the grid.

Lines 70-110 count the words in the word list. Line 120 sets up the arrays used in the program. Lines 130-170 re-read the word list, storing it in the array WD$( ).

The word list is stored in DATA lines 180-200. For the time being, use the words provided; after you have the program running, replace them with your own. Insert as many extra DATA lines as you need between lines 180 and 200, and use as many words as you wish. *Be sure to include the "/" character after the last word, as shown in line 200.*

Lines 210-240 define the eight directions for the computer. (Refer to Figure 2-4.) A pair of numbers is associated with each direction-number 1 through 8. The two numbers indicate vertical and horizontal increments of the path. For example, direction 1 (east) is defined by the pair (0,1), which indicates a zero vertical movement and a positive horizontal (left-to-right) movement. Direction 7 (north) is defined by the pair (−1,0), which indicates an upward vertical movement and a zero horizontal movement.

Although the program normally uses all eight directions, you can make the puzzles easier by eliminating directions 4, 5, 6, and 7. The easiest way to do this is to change line 240 to read as follows:

```
240 DATA 0,1,1,1,1,0,-1,1,0,1,1,1,1,0,-1,1
```

Printing Introductory Messages

The following lines print an introductory message and set up the grid:

```
250 PRINT CHR$(147);: REM CLEAR SCREEN
260 PRINT "HIDDEN-WORD PUZZLE GENERATOR"
270 PRINT
280 PRINT "GRID SIZE IS " MR " BY " MC
290 PRINT
300 PRINT "VOCABULARY CONTAINS " NW " WORDS."
310 PRINT
320 PRINT "SETTING UP THE GRID. PLEASE WAIT."
330 FOR R=1 TO MR
```

```
340 FOR C=1 TO MC
350 M$(R,C)=SP$
360 NEXT C,R
370 REM
380 FOR C=1 TO NC
390 SQ(C)=0
400 NEXT C
410 FOR C=1 TO NC
420 Q=INT(RND(1)*NC)+1
430 IF SQ(Q)<>0 THEN 420
440 SQ(Q)=C
450 NEXT C
460 FOR W=1 TO NW
470 WQ(W)=0
480 WU(W)=0
490 NEXT W
500 FOR W=1 TO NW
510 Q=INT(RND(1)*NW)+1
520 IF WQ(Q)<>0 THEN 510
530 WQ(Q)=W
540 NEXT W
```

Line 350 stores a hyphen (SP$) in each cell to indicate that the cell is empty. Lines 380-450 define the sequence in which the computer will examine the grid for empty cells. The sequence is randomized so that you can produce different puzzles using the same word list and grid dimensions. Lines 460-540 define the sequence in which the computer picks words to fit into the grid.

Checking the Puzzle Status

Now the program can start checking the grid cells.

```
550 MF=0
560 WA=NW
570 REM
580 DI=1
590 PRINT "STARTING TO FILL IN THE GRID . . ."
600 FOR QP=1 TO NC
610 CP=SQ(QP)
620 CR=INT((CP-1)/MC)+1
630 CC=CP-(CR-1)*MC
640 IF M$(CR,CC)<>SP$ THEN 1779
650 IF WA<>0 THEN 680
660 PRINT "USED ALL THE WORDS"
665 QP=NC
670 GOTO 1780
680 M$(CR,CC)=MK$
```

MF is a status flag (this will be explained later). WA is the number of words available; initially, it is the same as NW, the number of words in the vocabulary. When no words are left, the program quickly fills in all blank cells with randomly chosen letters.

DI is the starting direction. You may wish to set DI=2 so that the program starts with a diagonal (southeast) direction. In any case, DI changes as needed, so that all eight directions are tried.

Grid cells are numbered from left to right. The variable CP is the current cell number, which ranges from 1 to NC. Lines 620 and 630 calculate the row and column "address" (CR,CC) of the cell using the value of CP.

Line 640 checks whether the cell is filled. If it is, the program jumps to line 1779 and calls for the next cell. Line 650 checks whether all the words have been used. If they have, the program jumps to line 1730 and fills the cell with a randomly chosen letter. Line 680 marks the current cell with an asterisk (MK$) so that it is readily visible inside each path and subpath that is generated later.

Finding the Current Direction

Given an empty cell, the program now finds the major path containing that cell.

```
690 DK=1
700 IR=D(DI,1)
710 IC=D(DI,2)
720 RT=1
730 IF IR<0 THEN RT=MR
740 IF IR=0 THEN RT=CR
750 CT=1
760 IF IC<0 THEN CT=MC
770 IF IC=0 THEN CT=CC
780 BR=CR
790 BC=CC
800 IF ((BR=RT) AND (IR<>0)) OR ((BC=CT) AND
    (IC<>0)) THEN 840
810 BR=BR-IR
820 BC=BC-IC
830 GOTO 800
840 RT=1
850 IF IR>0 THEN RT=MR
860 IF IR=0 THEN RT=CR
870 CT=1
880 IF IC>0 THEN CT=MC
890 IF IC=0 THEN CT=CC
```

```
900 ER=CR
910 EC=CC
920 IF ((ER=RT) AND (IR<>0)) OR ((EC=CT) AND
    (IC<>0)) THEN 960
930 ER=ER+IR
940 EC=EC+IC
950 GOTO 920
```

DK, initialized in line 690, counts the number of directions tested for a given cell. When all eight directions have been tried (DK=8), the program changes the asterisk back to a blank and skips to the next cell.

Lines 700-830 find the beginning cell in the path. IR and IC are the row and column increments that correspond to direction DI. The row and column limits are stored in RT and CT respectively.

The program starts at the current cell and steps through the grid in the specified direction until it reaches one of the limits RT or CT. That's how it finds the beginning position (coordinates BR,BC) of the path.

Lines 840-950 find the ending cell in the path (coordinates ER,EC) in the same manner.

Finding the Current Subpath

The following block of lines builds a string containing the contents of the major pathway and tries each subpath in the major pathway:

```
960 UR=ER
970 IF BR>ER THEN UR=BR
980 LR=BR
990 IF ER<BR THEN LR=ER
1000 UC=EC
1010 IF BC>EC THEN UC=BC
1020 LC=BC
1030 IF EC<BC THEN LC=EC
1040 PR=BR
1050 PC=BC
1060 P$="": REM NO SPACES IN QUOTES
1070 P$=P$+M$(PR,PC)
1080 PR=PR+IR
1090 PC=PC+IC
1100 IF (PR>=LR) AND (PR<=UR) AND (PC>=LC) AND
     (PC<=UC) THEN 1070
1110 PL=LEN(P$)
1120 Q0=1
1130 Q1$=P$
1140 Q2$=MK$
1150 GOSUB 2140
```

```
1160 SP=QF
1170 FOR LS=1 TO SP
1180 FOR RS=PL TO SP STEP -1
1190 CP$=MID$(P$,LS,RS-LS+1)
1200 CL=LEN(CP$)
```

The variable P$ stores the contents of the path. To generate subpaths, the progam refers to P$ rather than stepping through the grid array each time. In line 1190, MID$(P$,LS,RS−LS+1) is the current subpath.

Fitting a Word to the Subpath

Next the program attempts to fit a word into the subpath.

```
1210 Q=1
1220 W=WQ(Q)
1230 IF LEN(WD$(W))=CL THEN 1260
1240 MF=0
1250 GOTO 1650
1260 MF=1
1270 FOR C=1 TO CL
1280 W1$=MID$(CP$,C,1)
1290 IF W1$=SP$ OR W1$=MK$ OR W1$=MID$(WD$(W),C,1)
     THEN 1320
1300 C=CL
1310 MF=0
1320 NEXT C
1330 IF MF=0 THEN 1650
```

Lines 1210-1220 select the number of the first word in the random sequence. Line 1230 checks whether the word's length matches that of the current subpath.

In line 1240, MF is a status indicator: MF=0 signifies that a word cannot be used in the path for some reason (it has been used already, is the wrong length, or contains a conflicting letter).

Lines 1270-1320 compare the trial word WD$(W) with the contents of the subpath. If the word doesn't match, the program selects another word (see line 1650).

Adding a Word to the Puzzle

If the word matches, it must be implanted in the grid. The following lines perform this task:

```
1340 FW$=WD$(W)
1350 IF LS<=1 THEN 1390
1360 FW$=DC$+FW$
1370 LS=LS-1
1380 GOTO 1350
1390 IF RS>=PL THEN 1430
1400 FW$=FW$+DC$
1410 RS=RS+1
1420 GOTO 1390
1430 PR=1
1440 R=BR
1450 C=BC
1460 RC$=MID$(FW$,PR,1)
1470 IF RC$=DC$ THEN 1490
1480 M$(R,C)=RC$
1490 IF (R=ER AND IR<>0) OR (C=EC AND IC<>0)
     THEN 1540
1500 C=C+IC
1510 R=R+IR
1520 PR=PR+1
1530 GOTO 1460
```

Line 1340 stores the current word in FW$. Lines 1350-1420 pad the word with dummy characters DC$ so that its length matches that of the major path P$. Lines 1430-1530 insert the word into the grid one letter at a time.

Checking the Word List

After adding a word, the program must do quite a bit of housekeeping.

```
1540 IF (Q=WA) THEN 1580
1550 FOR QI=Q TO WA-1
1560 WQ(QI)=WQ(QI+1)
1570 NEXT QI
1580 WA=WA-1
1590 WU(W)=1
1600 RS=SP
1610 LS=SP
1620 DK=8
1630 PRINT "W ";: REM 1 SPACE AFTER W
1640 GOTO 1670
1650 Q=Q+1
1660 IF (Q<=WA) THEN 1220
1670 NEXT RS,LS
1690 DI=DI+1
1700 DK=DK+1
1710 IF DI>8 THEN DI=1
1720 IF DK<=8 THEN 700
```

```
1730 IF MF<>0 THEN 1779
1740 M$(CR,CC)=SP$
1779 PRINT NC-QP;" ";: REM 1 SPACE IN QUOTES
1780 NEXT QP
1781 PRINT: PRINT "FILLING IN UNUSED BLANKS"
1782 FOR QP=1 TO NC
1783 CP=SQ(QP)
1784 CR=INT((CP-1)/MC)+1
1785 CC=CP-(CR-1)*MC
1786 IF M$(CR,CC)<>SP$ THEN 1789
1787 M$(CR,CC)=CHR$(INT(RND(1)*26)+65)
1789 NEXT QP
```

Lines 1540-1580 update the list of available words and words used. When a word WD$(W) is used, WU(W) is set to 1 (line 1590). Lines 1650-1670 cause the program to select the next word and subpath. Line 1690 selects a new direction. Line 1720 checks whether all eight directions have been tried; if they have, the program has exhausted all possibilities for the current cell, so it restores a blank in that cell (line 1740). Line 1780 moves the program on to the next cell.

After all cells have been tried, lines 1781-1789 fill the empty cells with randomly chosen letters.

Printing the Puzzle

When all cells have been examined, the puzzle is complete. The following block of lines lets you select which output device to use for the puzzle:

```
1790 PRINT
1800 PRINT "PUZZLE COMPLETED."
1810 PRINT
1820 DV=1
1830 INPUT "SELECT OUTPUT TO: 1-CRT 2-PRINTER ";DV
1840 IF DV<>1 AND DV<>2 THEN 1830
1850 IF DV=1 THEN 1870
1860 OPEN 1,4: CMD 1
1870 PRINT CHR$(147);
```

Lines 1820-1850 let you specify the output device. The program assumes that device number 4 is your printer. If necessary, change 4 to the appropriate number in lines 1860 and 1950.

The following lines print the puzzle:

```
1900 GOSUB 2070
1920 PRINT
```

```
1930 IF DV=2 THEN PRINT#1,: CLOSE 1
1940 INPUT "PRESS RETURN FOR HIDDEN WORD LIST";D$
1950 IF DV=2 THEN OPEN 1,4: CMD 1
1960 PRINT
1970 PRINT "THE HIDDEN WORDS ARE:"
1980 PRINT
2000 FOR QI=1 TO NW
2010 IF WU(QI)<>0 THEN PRINT WD$(QI)
2020 NEXT QI
2040 PRINT
2050 IF DV=2 THEN PRINT#1,:CLOSE 1
2060 INPUT "SELECT: 1-REPEAT 2-NEW PUZZLE 3-END";X
2062 IF X<1 OR X>3 THEN 2060
2064 ON X GOTO 1810,250,2066
2066 END
2070 FOR TR=1 TO MR
2080 FOR TC=1 TO MC
2090 PRINT M$(TR,TC)" ";: REM 1 SPACE IN QUOTES
2100 NEXT TC
2110 PRINT
2120 NEXT TR
2130 RETURN
```

Line 1900 calls a subroutine to print the puzzle. A subroutine is used here to facilitate testing of the program.

Lines 1940-2020 print the list of words that were used. The computer isn't always able to fit the entire vocabulary into the grid. Line 2010 ensures that only those words actually used in the grid are printed.

After printing the puzzle, lines 2060-2064 print a continuation menu with three options: reprint the current puzzle, create a new one, or quit.

The subroutine in lines 2070-2130 prints the puzzle one line at a time with a space added after each letter. You can change the proportions of the puzzle by storing more spaces inside the quotes in line 2090.

Finding a Substring

Finally, here's the subroutine that searches for one string inside another:

```
2140 QF=0
2150 IF Q2$="" THEN RETURN:REM NO SPACES IN QUOTES
2160 IF Q0+LEN(Q2$)-1>LEN(Q1$) THEN RETURN
2170 IF MID$(Q1$,Q0,LEN(Q2$))=Q2$ THEN 2200
2180 Q0=Q0+1
2190 GOTO 2160
2200 QF=Q0
2210 RETURN
```

The U.S.A.

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

Alabama
Alaska
Arizona
Arkansas
California
Colorado
Connecticut
Delaware
Florida
Georgia
Hawaii
Idaho
Illinois

Indiana
Iowa
Kansas
Kentucky
Louisiana
Maine
Maryland
Massachusetts
Michigan
Minnesota
Mississippi
Missouri
Montana

Nebraska
Nevada
New Hampshire
New Jersey
New Mexico
New York
North Carolina
North Dakota
Ohio
Oklahoma
Oregon
Pennsylvania

Rhode Island
South Carolina
South Dakota
Tennessee
Texas
Utah
Vermont
Virginia
Washington
West Virginia
Wisconsin
Wyoming

Figure 2-5. The names of all 50 states are hidden in the grid

This handy subroutine searches for Q2$ inside Q1$ starting at position Q0. Upon return from the subroutine, QF gives the position at which Q2$ starts in Q1$; QF=0 indicates that Q2$ was not found.

—Testing and Using the Program

For large vocabularies, puzzle construction may take as long as 30 minutes. The giant U.S.A. puzzle in Figure 2-5 required four hours to generate.

The program prints various messages during the process to let you know it's working. For example, you will see SETTING UP THE GRID. PLEASE WAIT. while the program initializes the random word and cell sequences, W when it implants a word, F when it cannot place a word and implants a randomly chosen letter. Before moving onto a new cell, the program prints the number of cells remaining to be examined.

—Viewing the Program's Operation

While testing the program, reduce the vocabulary to three or four short words and the maze dimensions to 4 × 8. To obtain a printout of the puzzle in progress, intersperse GOSUB 2070 statements at strategic points in the program. For example, the statement

```
685 PRINT: PRINT: PRINT "TRYING A NEW CELL...":
    GOSUB 2070: PRINT
```

prints the puzzle in its current form followed by the word it is trying to fit in. An asterisk marks the currently selected cell.

Delete line 685 after you have the program running.

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 3

The Matchmaker

The Matchmaker program enables you to create a never-ending series of personalized logic puzzles. What are logic puzzles? They are puzzles in which, given a set of logical clues, you are to reach the one and only solution that satisfies each of the conditions presented by the clues.

For example, if Ann likes alligators, then Cathy detests cats. If Cathy likes alligators, then Bill likes birds. If Ann detests alligators, then Cathy detests cats. If Cathy detests cats, then Bill likes alligators. Match each person with his or her favorite animal. The answer is given at the end of this chapter.

Perhaps your tastes run toward mysteries: If the murderer does not have a blue pickup, then the postman has red hair. If the postman does not have a tattoo, then the milkman does not have a blue pickup. If the milkman does not have red hair, then the postman does not have white overalls.

If the murderer has a tattoo, then the garbageman has white overalls. If the garbageman does not have a tattoo, then the milkman does not have white overalls. If the garbageman does not have a blue pickup, then the postman has red hair. If the garbageman does not have a blue pickup, then the postman does not have red hair.

Describe the murderer. (The answer is also given at the end of this chapter.)

—Supplying Lists of Clues

The Matchmaker starts with two lists that you provide: the first is a list of characters, and the second, of attributes. The favorite animal puzzle is based on the lists shown in Table 3-1.

With the character and attribute lists, the Matchmaker formulates a system of logical propositions or clues concerning the pairings of items from the two lists. Taken together, the clues imply a unique solution in which every item from the first list is paired with one and only one item from the second list.

Propositions can take four forms:

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

In formal logic, *p* is known as the *antecedent* and *q* as the *consequent.* In our puzzles, p and q stand for pairs of items from the two lists. The logical operator *not* indicates that a pairing is not true. Here are a couple of examples: *If Bill likes cats then Ann likes alligators* corresponds to *p implies q,* while *if Ann detests (does not like) cats, then Bill likes birds* corresponds to *not p implies q.*

The favorite animal puzzle includes all four types of propositions. Read through the puzzle again, identifying the four types.

In order to realize the extent of the Matchmaker's talents, try to construct your own logic puzzle using the data in Table 3-1. The trick is to give only enough clues so there will be a unique solution. You must

Table 3-1. Two Lists for the Favorite Animal Puzzle

Characters	Attributes
Ann	Alligators
Bill	Birds
Cathy	Cats

also take care not to create invalid logical systems, that is, puzzles for which there is no solution.

Go ahead, try it. Then read on to see the Matchmaker's way.

How the Program Generates a Puzzle

The first challenge for the computer is to generate all potential solutions before any clues have been given.

For two lists of n items, there are

$$n \times (n-1) \times (n-2) \times \ldots \times 1$$

potential solutions. (Technically, the formula gives the number of permutations of a set of n objects.) For groups of three, six matchups are possible; for groups of four, 24 matchups are possible.

As an exercise, list all potential solutions to the favorite animal puzzle, assuming that no clues have been given yet. Hint: the first might be Ann likes alligators, Bill likes birds, and Cathy likes cats.

The program generates each potential solution and stores the solution as a column in a truth table. A truth table represents the true or false value for every combination of items from two groups. Table 3-2 shows a truth table for the favorite animal puzzle. Each row in the table corresponds to a pair of items; each column corresponds to a puzzle solution.

A T or F in the table indicates whether a given pair is true or false for the corresponding solution. For instance, at the intersection of row A1 and column 1 we find a T, indicating that in solution 1, Ann likes alligators. At row B1, column 1, we find an F, indicating that in solution 1, Bill does not like alligators.

Using the potential-solution truth table, the Matchmaker generates a succession of clues. There are several steps in this process.

First the program randomly selects a potential solution. From the corresponding solution column in Table 3-2, it randomly selects a pair. This pair becomes the antecedent in the proposition *p implies q*.

Next the program randomly selects another pair from the same solution column. This pair becomes the consequent in the proposition *p implies q*.

Refer to Table 3-2. Suppose the program randomly selects solution 4 (column 4). Then it randomly selects the pairing A1 (corresponding to Ann/alligators). The truth value of that pair in column 4 is F, indicating

Table 3-2. Truth Table Showing Potential Solutions to Logic Puzzles With Three Items in Each List (A,B,C) and (1,2,3)

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

that Ann does not like (detests) alligators. Suppose the program then randomly selects C3 (Cathy/cats). The truth value is also F, indicating that Cathy detests cats.

Putting the two pairs together, we have the proposition: *if Ann detests alligators, then Cathy detests cats.* We know that the clue is consistent with at least one of the solutions because the pairings are taken directly from one of the solutions.

Before accepting the clue, the program checks it against all previously generated clues to ensure that

- The clue is not redundant; that is, the clue must eliminate at least one potential solution.
- The cumulative effect of the preceding clues and the latest one is to leave at least one solution; otherwise, the puzzle would be insoluble.

If both conditions are satisfied, the clue is accepted and the program continues. If either condition is not satisfied, the clue is discarded and the program generates a new candidate.

Table 3-3. Effects of Four Clues on the Potential-Solution List

Clues:	Consistency With Potential Solutions 1	2	3	4	5	6
A1 implies not C3	F	T	T	T	T	T
C1 implies B2	T	T	T	F	T	T
not A1 implies not C3	T	T	F	T	T	T
not C3 implies B1	T	F	T	F	T	F

Table 3-3 shows the effects of four clues on the list of potential solutions.

The Matchmaker continues generating clues until only one solution remains. At that point, the puzzle is complete.

—The Program

We present the program in logical blocks. The first block reads in the data lists and creates several arrays.

```
10 PRINT CHR$(147);: REM CLEAR SCREEN
20 PRINT "THE MATCHMAKER"
30 PRINT
40 INPUT "ENTER A RANDOM NUMBER";X
50 X=RND(-ABS(X))
60 DATA 3
70 DATA IS NOT PAIRED WITH, IS PAIRED WITH
80 DATA A,B,C
90 DATA 1,2,3
100 READ N
110 NS=1
120 FOR J=1 TO N
130 NS=NS*J
140 NEXT J
150 NC=N*N
160 DIM A(N),PT(N),K(N),C(N),T(NC,NS),F(NS,NS),
    FT(NS),P(NS,3),A$(2,N),TF$(2)
```

```
170 READ TF$(1),TF$(2)
180 FOR J=1 TO 2
190 FOR K=1 TO N
200 READ A$(J,K)
210 NEXT K,J
```

Lines 40 and 50 set the random number seed. Enter a different number each time you run the program; otherwise, the program will generate the same series of puzzles.

Lines 60-90 determine what kinds of puzzles are generated. Line 60 gives the number of items in each list. The number must be 3 or 4. Line 70 gives the verbs that relate the items from the two lists. Line 80 contains the list of characters. Line 90 contains the list of attributes.

(For the time being, use the rather abstract data provided; it will simplify the discussion of the program. After testing the program, you can personalize the puzzles by making your own lists. Instructions for personalizing the program are given later in this chapter.)

Lines 110-140 compute the number of potential solutions NS. This value depends on N, the number of items in each list. For N=3, NS=6; for N=4, NS=24; for N=5, NS=120. Unfortunately, the Commodore doesn't have enough memory to store this many potential solutions. Line 150 computes the number of combinations of items from lists 1 and 2.

Line 160 sets up the arrays. A( ), PT( ), C( ), and K( ) are used during the generation of the NS distinct solutions. T(,) is a truth table representing these solutions. It corresponds to Table 3-2. For example, T(2,3) indicates the true/false value for pair 2 in solution 3.

P(,) stores the propositions. F(,) is a truth table indicating the results of each proposition. It corresponds roughly to the right side of Table 3-3. As an example, F(1,2) indicates whether proposition 1 is consistent with solution 2.

FT( ) is a truth table showing the cumulative effect of all preceding propositions. As an example, FT(1) indicates whether solution 1 is consistent with all preceding clues.

A$(,) stores the two lists. A$(1,3) is the third item in list 1, for example. TF$( ) stores the verbs used to relate items from lists 1 and 2.

Lines 170-210 read the data into the appropriate arrays.

Generating Potential Solutions

The next logical block generates the truth table T(,). Recall that a truth table represents the true or false value for every combination of items from two groups.

```
230 FOR J=1 TO N
240 A(J)=1
250 NEXT J
260 P=0
270 LC=1
280 A(PT(LC))=1
290 PT(LC)=PT(LC)+1
300 IF PT(LC)<=N THEN 330
310 PT(LC)=PT(LC)-N
320 GOTO 300
330 IF A(PT(LC))=0 THEN 290
340 C(LC)=PT(LC)
350 K(LC)=K(LC)+1
360 A(PT(LC))=0
370 IF LC=N THEN 400
380 LC=LC+1
390 GOTO 290
400 P=P+1
410 FOR J=1 TO N
420 T((J-1)*N+C(J),P)=1
430 NEXT J
440 A(PT(LC))=1
450 K(LC)=0
460 LC=LC-1
470 IF LC=0 THEN 500
480 IF K(LC)=N-LC+1 THEN 440
490 GOTO 280
```

In lines 230-390, the program generates all arrangements of the items in list 2: 123, 132, 213, 231, 312, and 321.

For each triplet, A is inserted ahead of the first number, B ahead of the second number, and C ahead of the third number. This gives us the following sequence of triplets: A1B2C3, A1B3C2, A2B1C3, A2B3C1, A3B1C2, and A3B2C1.

Each triplet thus produced constitutes a potential solution to the logic puzzle. For instance, A1B2C3 represents the solution: A is paired with 1, B with 2, and C with 3.

Lines 400-430 record the details of each solution in the truth table T(,).

Lines 440-490 set the program to generate the next arrangement of items (for example, 123).

Making and Selecting Propositions

After the truth table is completely filled in, the program can begin making logical propositions.

```
500 PRINT CHR$(147);
510 PRINT "THE MATCHMAKER IS"
520 PRINT
530 PRINT "CONSTRUCTING A LOGIC PUZZLE"
540 PRINT
550 PRINT "FROM "N" DATA PAIRS."
560 PRINT
580 PRINT "PLEASE WAIT."
600 PO=0
610 QO=0
620 FC=0
630 PN=0
640 FOR J=1 TO NS
650 FT(J)=1
660 NEXT J
```

In line 650, the array FT( ) is filled with 1's, indicating that none of the solutions has been ruled out yet.

The next lines randomly select a proposition.

```
670 G=INT(RND(1)*NS)+1
680 P=INT(RND(1)*NC)+1
690 Q=INT(RND(1)*NC)+1
700 P1=INT((P-1)/N)+1
710 P2=P-(P1-1)*N
720 Q1=INT((Q-1)/N)+1
730 Q2=Q-(Q1-1)*N
740 IF (P1=Q1) OR (P2=Q2) THEN 670
750 PV=T(P,G)
760 QV=T(Q,G)
770 IF (PV=PO) AND (QV=QO) THEN 670
780 PN=PN+1
790 P(PN,1)=G
800 P(PN,2)=P
810 P(PN,3)=Q
```

Lines 670-690 randomly select solution G, antecedent pair P, and consequent pair Q. G ranges from 1 to NS (the number of solutions); P and Q range from 1 to NC (the number of combinations).

Lines 700 and 710 break P into two numbers, P1 and P2, corresponding to the items P represents from lists 1 and 2. For example, P=6 breaks down into P1=2, P2=3. This stands for the pair item 2, list 1/item 3, list 2.

Lines 720 and 730 accomplish a similar function for Q, Q1, and Q2.

At this point, we have an antecedent pair P1-P2 and a consequent pair Q1-Q2. Typical values might be A2 and B3, producing clues like this: *if A2 then B3.*

However, it is possible to have duplicate pairs like A2 and A2—

which would produce useless propositions such as: *if A2, then A2.* In fact, even if the two pairs have just one item in common, as in A2–C2, the resultant proposition will also be useless, as in *if A2, then C2.*

Line 740 eliminates all such "weak" propositions from consideration.

Lines 750 and 760 get the truth values of the antecedent P1-P2 and the consequent Q1-Q2 for solution G. The truth table entries T(P,G) and T(Q,G) contain this information.

Line 770 ensures that there is some variety from one proposition to the next by comparing the current proposition with the previous one. If both have the same form, the current proposition is rejected and a new one is selected. For example, if the current proposition and the previous proposition both have the form *not p implies q*, the current one is rejected.

At this point, the proposition has passed first inspection. Lines 780-810 increment the proposition counter and store the details of the latest proposition in the array P(,).

Testing the Clues

Now the Matchmaker tests the effects of the latest clue on each of the potential solutions.

```
820 FOR J=1 TO NS
830 PT=T(P,J)
840 QT=T(Q,J)
850 IF PV<>PT THEN 890
860 IF QV=QT THEN 890
870 F(PN,J)=0
880 GOTO 900
890 F(PN,J)=1
900 NEXT J
910 FA=0
920 FOR J=1 TO NS
930 IF FT(J)=1 AND F(PN,J)=0 THEN FA=FA+1
940 NEXT J
950 IF FA>0 THEN 980
960 PN=PN-1
970 GOTO 670
980 IF FA+FC>=NS THEN 620
```

For each potential solution column, the program examines the truth values PT, QT of the pairs P1-P2 and Q1-Q2 (lines 820-840). It compares these with the truth values PV, QV of the latest clues (lines 850-860). If PT=PV and QT=QV, then the solution is consistent with the latest clue.

The result of the comparison is stored in array F(,) (lines 870 and 890).

Lines 910-960 determine whether the latest clue has actually provided any new information, that is, whether it has ruled out any solutions that were previously viable. FA counts the number of solutions ruled out by the latest clue. If FA=0, the latest clue is redundant, so the program rejects it by decrementing the proposition counter in line 960 and going back to line 670 for a new clue.

Line 980 ensures that at least one solution remains viable after the latest clue. FC counts the total number of solutions eliminated by previous clues. FA+FC gives the total eliminated when the latest clue is taken into account. If the sum equals or exceeds the number of solutions NS, the puzzle is insoluble. In that case, the program erases the entire sequence of propositions and starts over by going back to line 620.

Completing the Puzzle

After passing all these hurdles, the clue is finally accepted.

```
990 FC=FA+FC
1000 FOR J=1 TO NS
1010 IF F(PN,J)=0 THEN FT(J)=0
1020 NEXT J
1030 IF FC=NS-1 THEN 1070
1040 PO=PV
1050 QO=QV
1060 GOTO 670
```

Line 990 updates the counter FC for solutions eliminated. Lines 1000-1020 update the cumulative truth table F( ), which shows whether a given solution has been eliminated by any proposition thus far.

Line 1030 serves the very important function of determining whether the puzzle is complete. When FC=NS−1, only one solution remains, and the puzzle is solved and ready for presentation.

On the other hand, if more than one solution remains, lines 1040 and 1050 record the latest truth values so they will not be repeated in the next clue.

Printing the Puzzle and Clues

The following lines let you select the output device for the puzzle:

```
1070 PRINT CHR$(147);: REM CLEAR SCREEN
1080 PRINT "THE PUZZLE IS READY."
1090 PRINT
```

```
1100 DV=1
1110 PRINT "OUTPUT TO 1-DISPLAY 2-PRINTER"
1120 INPUT "SELECT 1 OR 2";DV
1130 IF DV<>1 AND DV<>2 THEN 1090
1140 INPUT "PRESS RETURN WHEN READY";RT$
1150 PRINTCHR$(147);: REM CLEAR SCREEN
1155 IF DV=2 THEN OPEN 1,4: CMD 1
```

If your printer has a device number different from 4, change 4 to the correct value in line 1155. After printing the puzzle, the program will give you the option of reprinting it to another device, so it's a good idea to start with output to the CRT.

Before printing any clues, the program prints the two lists, so you'll know what items are to be matched.

```
1160 PRINT "MATCH LEFT COLUMN WITH RIGHT COLUMN"
1170 PRINT
1180 PRINT TAB(8) "(" TF$(2) ")"
1190 PRINT
1200 FOR J=1 TO N
1210 PRINT A$(1,J) TAB(30) A$(2,J)
1220 NEXT J
1230 PRINT
```

After this, the program prints the clues.

```
1240 IF DV=2 THEN PRINT#1,: CLOSE 1
1250 INPUT "PRESS RETURN TO SEE YOUR CLUES";RT$
1260 IF DV=2 THEN OPEN 1,4: CMD 1
1270 PRINT
1280 PRINT "HERE ARE THE CLUES..."
1290 PRINT
1300 FOR J=1 TO PN
1310 G=P(J,1)
1320 P=P(J,2)
1330 Q=P(J,3)
1340 PV=T(P,G)
1350 QV=T(Q,G)
1360 P1=INT((P-1)/N)+1
1370 P2=P-(P1-1)*N
1380 Q1=INT((Q-1)/N)+1
1390 Q2=Q-(Q1-1)*N
1400 PRINT "IF " A$(1,P1) " " TF$(PV+1) "
     " A$(2,P2)
1410 PRINT "THEN " A$(1,Q1) " " TF$(QV+1) "
     " A$(2,Q2) "."
1420 PRINT
1430 NEXT J
```

If your printer has a device number different from 4, change 4 to the correct value in line 1260. In lines 1300-1420, the program reconstructs each clue by cross-referencing P(,) and T(,). Let's take clue 1 for an example. G=P(1,1) stores the solution column number from which the clue was drawn. P=P(1,2) and Q=P(1,3) store the rows corresponding to the antecedent and consequent pairs. Therefore, PV=T(P,G) gives the truth value of the antecedent, and QV=T(Q,G) gives the truth value of the consequent. By way of illustration, assume G=4, P=1, and Q=9. Reading PV and QV from Table 3-2, we can recover the first clue: *not A1 implies not C3.*

Lines 1400-1430 print the clue in more conversational form, using the verbs provided in line 70.

Printing the Answers

After printing all the clues, the program offers to give the answers.

```
1440 IF DV=2 THEN PRINT#1,:CLOSE 1
1450 INPUT "PRESS RETURN TO SEE THE ANSWERS";EN$
1460 IF DV=2 THEN OPEN 1,4: CMD 1
1470 PRINT
1480 PRINT "HERE ARE THE ANSWERS
1490 PRINT
1500 FOR J=1 TO NS
1510 IF FT(J)=1 THEN X=J
1520 NEXT J
1530 FOR L=1 TO NC
1540 IF T(L,X)=0 THEN 1580
1550 P1=INT((L-1)/N)+1
1560 P2=L-(P1-1)*N
1570 PRINT A$(1,P1) " " TF$(2) " " A$(2,P2)
1580 NEXT L
```

If your printer's device number is not 4, change 4 to the correct value in line 1460.

In lines 1500-1520, the program recovers the solution by examining each element of FT( ) to find the one element that equals 1. The subscript of that element is the solution number.

Lines 1530-1580 go through each row of the truth table, looking only at column X (the solution column). If the truth value at that row-column intersection is 0 (for false), the program skips to the next row. Whenever it finds a 1 (for true), it prints the pairing in conversational form. For any given solution column, there will be only N true pairings, resulting in N positive statements of the form *P1-P2 is paired with Q1-*

Q2. Note there is a single space inside each pair of otherwise empty quotes in lines 1400-1410.

Starting Over

The following lines give you three choices: reprint the same puzzle, create a new puzzle, or quit:

```
1590 PRINT
1592 IF DV=2 THEN PRINT#1,:CLOSE 1
1595 INPUT "PRESS RETURN FOR WORKSHEET";RT$
1596 IF DV=2 THEN OPEN 1,4: CMD 1
1600 GOSUB 1720
1610 PRINT
1620 GOSUB 1880
1630 IF DV=2 THEN PRINT#1,:CLOSE 1
1640 PRINT
1650 REM
1660 PRINT "SELECT: 1-REVIEW PUZZLE"
1670 PRINT "        2-NEW PUZZLE"
1680 PRINT "        3-END"
1690 INPUT S
1695 IF S<1 OR S>3 THEN 1660
1700 ON S GOTO 1070,500,1710
1710 END
```

If your printer's device number is not 4, change 4 to the correct value in line 1596.

Lines 1600 and 1620 call the following subroutines that you may find helpful in the testing phase:

```
1720 PRINT "HERE ARE ALL THE VALID COMBINATIONS"
1730 PRINT
1740 PRINT
1750 FOR LZ=1 TO NC
1760 Z1=INT((LZ-1)/N)+1
1770 Z2=LZ-(Z1-1)*N
1780 PRINT CHR$(64+Z1); CHR$(48+Z2) TAB(5)
1790 FOR PZ=1 TO NS
1800 IF T(LZ,PZ)=0 THEN 1830
1810 PRINT "T";
1820 GOTO 1840
1830 PRINT "F";
1840 NEXT PZ
1850 PRINT
1860 NEXT LZ
1870 RETURN
1880 PRINT "HERE IS THE SOLUTION WORKSHEET"
```

```
1890 PRINT
1900 FOR IZ=1 TO PN
1910 GZ=P(IZ,1)
1920 PZ=P(IZ,2)
1930 QZ=P(IZ,3)
1940 PX=T(PZ,GZ)
1950 QX=T(QZ,GZ)
1960 IF PX=0 THEN C1$="#"
1970 IF PX=1 THEN C1$=" ": REM 1 SPACE IN QUOTES
1980 IF QX=0 THEN C2$="#"
1990 IF QX=1 THEN C2$=" ": REM 1 SPACE IN QUOTES
2000 Z1=INT((PZ-1)/N)+1
2010 Z2=PZ-(Z1-1)*N
2020 Z3=INT((QZ-1)/N)+1
2030 Z4=QZ-(Z3-1)*N
2040 P$=CHR$(64+Z1)+CHR$(48+Z2)
2050 Q$=CHR$(64+Z3)+CHR$(48+Z4)
2060 PRINT C1$ P$ "=>" C2$ Q$ TAB(10);
2070 FOR JZ=1 TO NS
2080 IF F(IZ,JZ)=0 THEN 2110
2090 PRINT "T";
2100 GOTO 2120
2110 PRINT "F";
2120 NEXT JZ
2130 PRINT
2140 NEXT IZ
2150 RETURN
```

Lines 2040 and 2050 build the antecedent and consequent pairs using a letter for the first element and a number for the second.

—Testing the Program

After typing in the entire program, run it. Eliminate any obvious typing errors you may have made. Figure 3-1 shows a sample run of the program for comparison.

The program solution worksheet is similiar to that shown in Table 3-3, but with the following notation: the letters A, B, and C are used in place of the items in list 1; the numbers 1, 2, and 3 are used in place of the items in list 2. The symbol # stands for *not* and the symbol => stands for *implies.* Accordingly, B3=>#A1 is shorthand for *if* B *is paired with 3, then A is not paired with 1.*

After you are satisfied that the program is running properly, you may wish to delete lines 1595, 1600, and 1620, which call the trouble-shooting subroutines. On the other hand, you may find it instructive to leave them in; the program can serve as a logic tutor in that way.

```
RUN
THE MATCHMAKER

ENTER A RANDOM NUMBER

THE MATCHMAKER

THE MATCHMAKER IS

CONSTRUCTING A LOGIC PUZZLE

FROM 3 DATA PAIRS.

PLEASE WAIT.

THE PUZZLE IS READY.

OUTPUT TO 1-CRT 2-PRINTER
SELECT 1 OR 2 1
PRESS RETURN WHEN READY
MATCH LEFT COLUMN WITH RIGHT COLUMN
                 (LIKES)

ANN                           ALLIGATORS
BILL                             BIRDS
CATHY                            CATS

PRESS RETURN TO SEE YOUR CLUES

HERE ARE THE CLUES...

IF ANN LIKES ALLIGATORS
THEN BILL DETESTS CARS.

IF CATHY DETESTS BIRDS
THEN BILL LIKES CATS.

IF BILL DETESTS BIRDS
THEN CATHY DETESTS ALLIGATORS.

PRESS RETURN TO SEE THE ANSWER

HERE ARE THE ANSWERS

ANN LIKES CATS
BILL LIKES ALLIGATORS
CATHY LIKES BIRDS

PRESS RETURN FOR WORKSHEET
```

Figure 3-1. Sample run of The Matchmaker (keyboard entries are underlined)

```
HERE ARE ALL THE VALID COMBINATIONS

A1       TTFFFF
A2       FFTTFF
A3       FFFFTT
B1       FFTFTF
B2       TFFFFT
B3       FTFTFF
C1       FFFTFT
C2       FTFFTF
C3       TFTFFF

HERE IS THE SOLUTION WORKSHEET

A1=>#B3      TFTTTT
#C2=> B3     FTFTTF
#B2=>#C1     TTTFTT

SELECT:      1 - REVIEW PUZZLE
             2 - NEW PUZZLE
             3 - END
```

Figure 3-1. Sample run of The Matchmaker (keyboard entries are underlined) (*continued*)

—Personalizing the Program

When you're ready to begin generating your own personalized puzzles, change lines 60-90 to suit your preferences.

First decide how many items to include in each list; you must choose either three or four. Lists of four items will produce the more difficult puzzles.

Make up the items for each list. Use characters in list 1 (line 80) and attributes in list 2 (line 90). Start with attributes that are mutually exclusive, such as *red hair, black hair, blond hair.* That makes things a little easier to keep track of.

Finally, select the verbs that will be used to indicate whether a given pairing is true or not. Use verbs with opposing meanings, such as *detests/likes, is not/is,* or *does not have/has.* Store the two verbs in line 70. Be sure to put the negative verb first.

By choosing the list items carefully, you can come up with some very interesting puzzles. Remember to use a different random number each time you run the program to generate a different series of clues.

To get you started, here are the data lines used to generate the puzzles at the beginning of this chapter. For the favorite animal puzzle:

```
60 DATA 3
70 DATA DETESTS,LIKES
80 DATA ANN,BILL,CATHY
90 DATA ALLIGATORS,BIRDS,CATS
```

To describe the murderer:

```
60 DATA 4
70 DATA DOES NOT HAVE,HAS
80 DATA THE MURDERER, THE MILKMAN,THE POSTMAN,
   THE GARBAGEMAN
90 DATA WHITE OVERALLS,A TATTOO,RED HAIR,
   A BLUE PICKUP
```

—Answers to Puzzles

Favorite animals: Ann likes cats. Bill likes alligators. Cathy likes birds.

Describe the murderer: The murderer has white overalls. The milkman has a tattoo. The postman has red hair. The garbageman has a blue pickup.

Chapter 4

Crossword Puzzle Designer—Part 1

This program generates eye-catching patterns for crossword puzzles. You can use these patterns to create personalized crossword puzzles. All you need is a good vocabulary, some free time, and a large eraser.

Even readers who don't enjoy crossword puzzles will find the Crossword Puzzle Designer an interesting exercise in array manipulation and print formatting.

A separate program that helps you fit a word list into a puzzle pattern is given in the next chapter.

How Crossword Puzzle Designer Works

A lot of care goes into the creation of the puzzle pattern before there's any thought about fitting in the words. Good puzzle patterns have the following properties:

- Solid blocks are arranged in symmetric, geometric, or representational patterns.

- Every possible path is numbered.
- Only one set of numbers is used for both horizontal and vertical paths.

Figure 4-1 illustrates each of these properties.

The Crossword Puzzle Designer starts by creating an empty grid that it divides into four numbered quadrants, as shown in Figure 4-2. The grid can range in size from 3 × 3 to 19 × 19.

To start, a certain number of randomly selected cells in quadrant 1 are marked as *block cells* (the black cells in a printed puzzle). The resultant pattern of blocks is rotated 90, 180, and 270 degrees and copied into quadrants 2, 3, and 4 respectively. This produces a radially symmetrical pattern: each of the four quadrants looks the same when viewed from the centerpoint. Of course, other types of symmetry are possible, but this one seems to give pleasing results.

Marking the block cells is only the program's first pass. The program begins a second pass for quality control by examining every path to find if any are too short (you may specify 2 or 3 as the minimum path length).

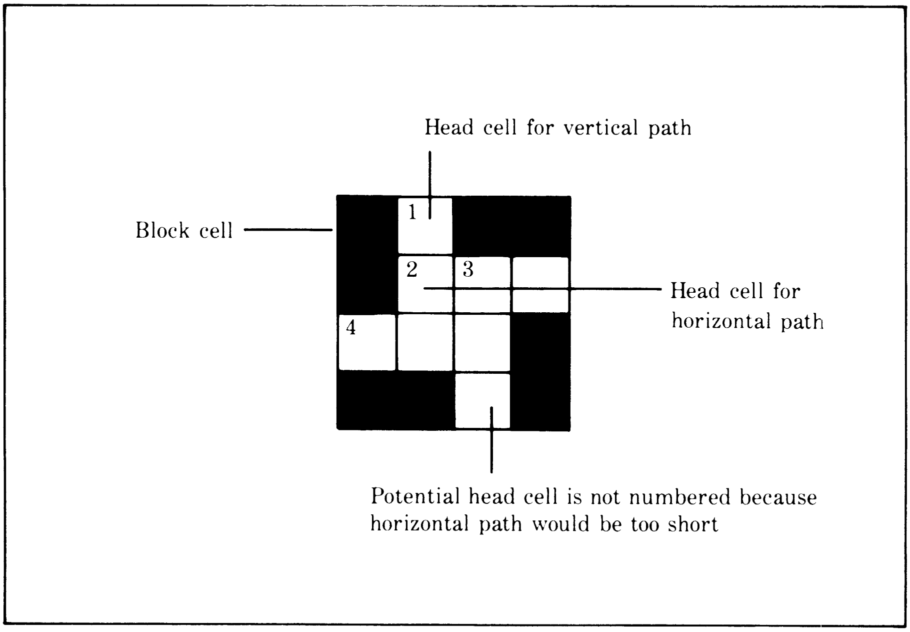


Figure 4-1: An illustrated puzzle pattern

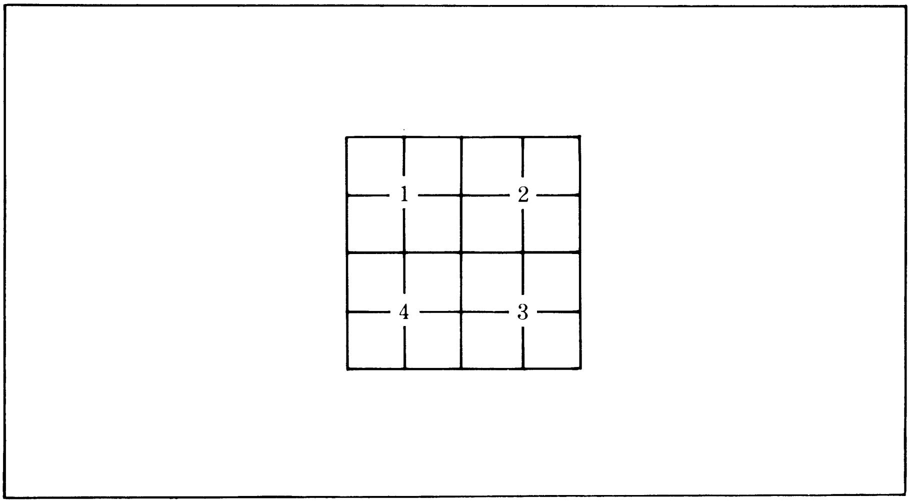


Figure 4-2: Four symmetrical quadrants of a puzzle pattern

The program looks for each potential *head cell* (the numbered cell that starts a path). Potential head cells are immediately below or to the right of block cells. An imaginary boundary of block cells surrounds the grid, so that every cell in the top row and left column is a potential head cell.

After finding a head cell, the program checks the length of the two paths (horizontal and vertical) originating there. If one of the paths is long enough, but the other is too short, the short path is left unused. If both paths are too short, the head cell is blocked out. To preserve the grid's overall symmetry, the corresponding cells in the other three quadrants are blocked out as well.

Finally, the program numbers the paths by locating the head cells and numbering them.

—The Program

The first block prints a title, asks you to input the puzzle size, and initializes the arrays.

```
10 PRINT CHR$(147);: REM CLEAR SCREEN
20 PRINT"CROSSWORD PUZZLE PATTERN GENERATOR"
30 PRINT
40 INPUT "PUZZLE SIZE (3-19) ";M
```

```
50 IF M>=3 AND M<=19 THEN 80
60 PRINT "INVALID SIZE."
70 GOTO 40
80 N=INT(M/2)
90 EO=0
100 IF N<>M/2 THEN EO=1
110 NC=M*M
120 NB=INT(NC/4)+1
130 DIM M$(M+1,M+1),PL(M,M),AD(NC,2),RC(NC,2)
133 S6$=CHR$(15):S9$=CHR$(8)
135 PQ=4: REM PRINTER'S DEVICE NUMBER
140 SC$="—"
150 BK$=CHR$(166): REM BLOCK CHARACTER
160 SP$="   ": REM 3 SPACES IN QUOTES
170 BL$=CHR$(166)+CHR$(166)+CHR$(166)
180 VL$=CHR$(125): REM VERTICAL LINE
190 HL$=CHR$(96)+CHR$(96)+CHR$(96)
```

The variable M is the overall grid size, and N is the size of each quadrant. Lines 90 and 100 determine whether M is even or odd; EO stores 0 when M is even and 1 when M is odd. This information is needed later when the puzzle is printed, because odd-sized grids cannot be cut down the middle. They have a center column and row which must be taken care of separately.

NC is the total number of cells in the grid. NB determines the number of blocks that will be stored in the grid—one for every four cells. However, the final number is usually higher because of blocks added during the check for short paths.

The array M$(*row,column*) stores the grid. For instance, M(1,2) is the cell at row 1, column 2. PL(*row,column*) indicates whether a grid cell is numbered or not; that is, whether it is a head cell. For instance, if a path originates at row 1, column 2, PL(1,2) stores the path number; if no path originates there, PL(1,2)=0.

AD(*path number,direction*) stores the length of each path in each direction. For instance, AD(3,1) stores the length of path 3 down; AD(3,2) stores the length of path 3 across. AD(,)=0 indicates that a path is unused.

Finally, RC(*path number,location*) stores the row and column address of each path's head cell. RC(5,1) stores the row of path 5; RC(5,2) stores the column address of path 5.

S6$ and S9$ are printer control codes used to select spacing of 6 or 9 lines per inch on the Commodore MPS-801 printer and compatible models. Use S6$=" " and S9$=" " if your printer doesn't have variable line spacing. PQ is the device number normally assigned to the printer. Change the number if necessary for use with your printer.

SC$ and BK$ indicate whether a grid cell is a letter cell or a block. The other variables assigned in lines 160-190 are used when the puzzle is printed.

Setup of the Puzzle's Design

The next lines set up the parameters for a specific pattern. They are executed each time you ask for a new pattern.

```
200 INPUT "ENTER A RANDOM NUMBER ";RN
210 X=RND(-ABS(RN))
220 INPUT "MINIMUM WORD LENGTH (2 OR 3)"; ML
230 IF ML<2 OR ML>3 THEN 220
240 PRINT "INITIALIZING"
250 GOSUB 1540
```

RN sets the random number generator seed so you can repeat a puzzle design. ML sets the minimum word length—something that can have a great effect on the puzzle's appearance. For puzzles smaller than 7 × 7, you will probably want to specify a minimum word length of 2 when you run the program; otherwise too many blocks will be filled in. Line 250 calls a subroutine that clears out all the arrays.

Marking the Blocked-Out Cells

The following block prepares the initial puzzle pattern:

```
260 PRINT "FIRST PASS"
270 FOR J=1 TO NB/4+1
280 R=INT(RND(1)*(N+EO))+1
290 C=INT(RND(1)*(N+EO))+1
300 IF M$(R,C)=BK$ THEN 280
310 NC$=BK$
320 GOSUB 1490
330 NEXT J
```

Lines 270-300 randomly block out NB/4+1 cells from quadrant 1. For every cell blocked out in quadrant 1, lines 310 and 320 block out a corresponding cell in quadrants 2, 3, and 4, producing a total of NB + 4 blocked-out cells. Line 330 repeats the process until the variable J has counted to NB/4+1.

Checking for Path Length

Now for the quality control check.

```
340 PRINT "SECOND PASS"
350 AF=0
360 FOR R=1 TO N+EO
370 FOR C=1 TO N+EO
380 IF M$(R,C)=BK$ THEN 590
390 HS=C-1
400 IF M$(R,HS)=BK$ THEN 430
410 HS=HS-1
420 GOTO 400
430 HE=C+1
440 IF M$(R,HE)=BK$ THEN 470
450 HE=HE+1
460 GOTO 440
470 VS=R-1
480 IF M$(VS,C)=BK$ THEN 510
490 VS=VS-1
500 GOTO 480
510 VE=R+1
520 IF M$(VE,C)=BK$ THEN 550
530 VE=VE+1
540 GOTO 520
550 IF HE-HS>ML OR VE-VS>ML THEN 590
560 AF=1
570 NC$=BK$
580 GOSUB 1490
590 NEXT C
600 NEXT R
610 IF AF=1 THEN 350
```

In line 350, AF is a status indicator whose function is explained below. Lines 360-600 examine every cell in quadrant 1 one row at a time. If a cell is not a block, it is assumed to be part of a path. Lines 390-460 measure the length of the horizontal path containing that cell. Lines 470-540 measure the length of the vertical path. If either of the paths satisfies the minimum path length requirement, the cell is allowed to remain as is (line 550). If both of the paths are too short, the cell is blocked out. Line 560 sets AF=1, indicating that a cell is going to be changed. Lines 570 and 580 change the cell in quadrant 1 and all corresponding cells in the other quadrants.

When the program reaches line 610, every cell in quadrant 1 has been checked. If AF=1, a cell has been changed to a block. That additional block might have created additional short paths, so the quality control check (lines 340-600) is repeated.

Numbering the Paths

When the program goes through the check without finding a short path, the check is complete and the cells can be numbered.

```
620 PRINT "NUMBERING THE PATHS NOW..."
630 PN=0
640 FOR R=1 TO M
650 FOR C=1 TO M
660 IF M$(R,C)=BK$ THEN 900
670 IF M$(R-1,C)<>BK$ THEN 780
680 VE=R+1
690 IF M$(VE,C)=BK$ THEN 720
700 VE=VE+1
710 GOTO 690
720 IF VE-R<ML THEN 780
730 PN=PN+1
740 PL(R,C)=PN
750 RC(PN,1)=R
760 RC(PN,2)=C
770 AD(PN,1)=VE-R
780 IF M$(R,C-1)<>BK$ THEN 900
790 HE=C+1
800 IF M$(R,HE)=BK$ THEN 830
810 HE=HE+1
820 GOTO 800
830 IF HE-C<ML THEN 900
840 IF PL(R,C)>0 THEN 890
850 PN=PN+1
860 PL(R,C)=PN
870 RC(PN,1)=R
880 RC(PN,2)=C
890 AD(PN,2)=HE-C
900 NEXT C
910 NEXT R
```

These lines examine every cell to find the head cells (cells that will be numbered in the final puzzle pattern).

PN counts the number of head cells found. This is not the same as the total number of paths, since a single head cell often references horizontal and vertical paths.

Lines 660-720 determine whether a cell heads a vertical path. If it does, lines 730-770 record the relevant information: the head cell count is incremented; the head cell number is stored in PL(,); the row and column of the head cell are stored in RC(,); and the path length is stored in AD(,1).

Lines 780-890 perform a similar function for horizontal paths. Lines 900-910 repeat the process until every cell in every row is checked.

Puzzle Options Menu

Now the pattern is complete. The program gives you several options: to view or print a miniature version (in case a large version won't fit on the screen), to view (or print) the puzzle, to view (or print) the path directory, to file the puzzle on disk, to erase and start a new puzzle, or to quit.

```
920 PRINT "PUZZLE PATTERN IS READY."
930 PRINT
935 PRINT "1-VIEW MINI-PUZZLE"
940 PRINT "2-VIEW THE EXPANDED PUZZLE"
950 PRINT "3-VIEW PATH DIRECTORY"
960 PRINT "4-FILE THE PUZZLE"
970 PRINT "5-ERASE AND START A NEW PUZZLE"
980 PRINT "6-QUIT."
990 PRINT
1000 INPUT "SELECT 1-6";S
1010 IF S<1 OR S>6 THEN 930
1020 ON S GOTO 1042,1050,1090,1250,200,1040
1040 END
```

Line 1020 selects the program block corresponding to the option you select.

Viewing the Puzzle

Here are the lines that enable you to view the miniature puzzle (use these lines if the puzzle size is greater than 13):

```
1042 GOSUB 2090
1043 GOSUB 2180
1044 GOSUB 2160
1045 GOTO 930
```

Line 1042 calls a subroutine to select the output device. Line 1043 calls a subroutine to print the miniature puzzle. Line 1044 calls a subroutine to restore the video display as the output device. All these subroutines are explained in detail later. Line 1045 returns to the option menu.

Here are the lines that print the full-size, presentation-quality puzzle pattern:

```
1050 GOSUB 2090
1060 GOSUB 1680
1070 GOSUB 2160
1080 GOTO 930
```

The lines are the same as for the miniature puzzle, except that line 1060 calls a subroutine to print the full-size puzzle.

Displaying the Path Directory

Here's the subroutine that displays the path directory.

```
1090 GOSUB 2090
1100 PRINT
1110 PRINT "ACROSS"
1120 PRINT "PATH #" TAB(10) "LENGTH"
1130 FOR QI=1 TO PN
1140 IF AD(QI,2)>0 THEN PRINT QI; TAB(10);AD(QI,2)
1150 NEXT QI
1160 PRINT
1170 PRINT "DOWN"
1180 PRINT "PATH #" TAB(10) "LENGTH"
1190 FOR QI=1 TO PN
1200 IF AD(QI,1)>0 THEN PRINT QI; TAB(10);AD(QI,1)
1210 NEXT QI
1230 GOSUB 2160
1240 GOTO 930
```

Line 1090 calls a subroutine to select the output device. Lines 1100-1150 print all the horizontal paths. For each path number QI, AD(QI,2)>0 indicates that a path exists. In that case, line 1200 prints the path number followed by the path length.

Lines 1160-1210 print all vertical paths in a similar fashion.

Line 1230 restores the video display as the output device, and line 1240 returns to the menu.

Saving the Puzzle on Disk

Here are the lines that file the puzzle pattern on disk.

```
1250 PRINT
1260 FO$="XWORD"+"."+RIGHT$(STR$(RN),
     LEN(STR$(RN))-1)
1262 FO$=FO$+"."+RIGHT$(STR$(M),LEN(STR$(M))-1)
1270 PRINT "FILING PUZZLE IN "FO$
1280 OPEN 1,8,4,"@0:"+FO$+",SEQ,W"
1320 PRINT#1,M
1330 FOR I=1 TO M
1340 FOR J=1 TO M
1350 PRINT#1, M$(I,J)
1360 PRINT#1, PL(I,J)
```

```
1370 NEXT J,I
1390 PRINT#1,PN
1400 FOR I=1 TO PN
1410 PRINT#1, AD(I,1)
1420 PRINT#1, AD(I,2)
1430 PRINT#1, RC(I,1)
1440 PRINT#1, RC(I,2)
1450 NEXT I
1460 CLOSE 1
1470 GOTO 930
```

Lines 1260-1262 construct an output file name XWORD and store it in FO$, along with the random number you supplied and the puzzle size. For example, if you have supplied the random number 1234 and puzzle size 9, the file will be named XWORD 1234.9.

Line 1280 opens a new file under the name in FO$.

Lines 1320-1380 print M, the puzzle size; M$(I,J), the contents of each cell (either a block or a blank); and PL(I,J), the path number to be printed in that cell (non-head cells are numbered 0).

Lines 1390-1450 print details about each path: PN, the number of paths; AD(I,1) and AD(I,2), the path length in the vertical and horizontal direction; and RC(I,1), RC(I,2), the row and column addresses of the path's head cell.

Line 1460 closes the file, and line 1470 returns to the menu.

The following subroutine stores the contents of NC$ in symmetrical positions in quadrants 1 through 4:

```
1480 REM
1490 M$(R,C)=NC$
1500 M$(C,N+N+EO+1-R)=NC$
1510 M$(N+N+EO+1-R,N+N+EO+1-C)=NC$
1520 M$(N+N+EO+1-C,R)=NC$
1530 RETURN
```

The variable EO effects the calculations only for odd-sized patterns since EO=0 when M is even and EO=1 when M is odd.

Preparing a New Puzzle

Here is the subroutine that erases an existing puzzle pattern and prepares for a new one.

```
1540 FOR I=0 TO M+1
1550 FOR J=0 TO M+1
1560 IF I=0 OR J=0 OR I=M+1 OR J=M+1 THEN 1600
```

```
1570 M$(I,J)=SC$
1580 PL(I,J)=0
1590 GOTO 1610
1600 M$(I,J)=BK$
1610 NEXT J
1620 NEXT I
1630 FOR I=1 TO NC
1640 AD(I,1)=0
1650 AD(I,2)=0
1660 NEXT I
1670 RETURN
```

For every row I, M$(I,0) represents the left boundary of the puzzle and M$(I,M+1) represents the right boundary. Similarly, for every column J, M$(0,J) represents the top boundary and M$(M+1,J) represents the bottom boundary.

Lines 1540 and 1550 and 1610 and 1620 cause I and J to count from 0 to M+1. The subscripts 0 and M+1 are used to generate an imaginary boundary around the grid. A block character is stored in each of these boundary cells (line 1600). For all nonboundary cells, line 1570 stores a blank character SC$ to indicate that the cell is available for a letter or a block.

Line 1580 stores a 0 in every element of PL(,), indicating that no cells have yet been numbered.

Lines 1630-1660 store 0 in every element of AD(,1) and AD(,2), indicating that no paths exist yet. Line 1670 returns to the main program.

Printing the Expanded Puzzle

The following subroutine prints the puzzle in expanded form. It is presented here in blocks for easier reading. The first block prints the top line of each row of cells.

```
1680 PRINT S6$;"PUZZLE #" RN "." ML: PRINT
1690 FOR QR=1 TO M
1700 FOR QC=1 TO M
1710 PRINT VL$;HL$;
1720 NEXT QC
1730 PRINT VL$;S9$: PRINT S6$;
```

The next block prints the second line of each row. For head cells, this line contains the path number. For other cells, it contains spaces or block characters.

```
1740 FOR QC=1 TO M
1750 PRINT VL$;
1760 IF M$(QR,QC)=BK$ THEN 1800
1770 IF PL(QR,QC)>0 THEN 1820
1780 PRINT SP$;
1790 GOTO 1850
1800 PRINT BL$;
1810 GOTO 1850
1820 QI$=STR$(PL(QR,QC))
1830 QI$=LEFT$(QI$+SP$,3)
1840 PRINT QI$;
1850 NEXT QC
1860 PRINT VL$;S9$: PRINT S6$;
```

Lines 1820-1840 handle the case of head cells. QI$ contains the appropriate head cell number.

The following block prints the last two lines of each row of cells. These lines consist of spaces for path cells and block characters for block cells.

```
1870 FOR QC=1 TO M
1880 PRINT VL$;
1890 IF M$(QR,QC)<>BK$ THEN 1920
1900 PRINT BL$;
1910 GOTO 1930
1920 PRINT SP$;
1930 NEXT QC
1940 PRINT VL$;S9$: PRINT S6$;
1950 FOR QC=1 TO M
1960 PRINT VL$;
1970 IF M$(QR,QC)<>BK$ THEN 2000
1980 PRINT BL$;
1990 GOTO 2010
2000 PRINT SP$;
2010 NEXT QC
2020 PRINT VL$;S9$: PRINT S6$;
2030 NEXT QR
```

All the cells have been printed at this point. Now the program adds a bottom line to close the puzzle.

```
2040 FOR QC=1 TO M
2050 PRINT VL$;HL$;
2060 NEXT QC
2070 PRINT VL$;S9$: PRINT S6$;
2080 RETURN
```

Line 2080 returns to the main program.

Finally, here are the subroutines to select an output device and restore the display as the output device.

```
2090 PRINT
2100 DV=1
2110 INPUT "SELECT 1-DISPLAY 2-PRINTER";DV
2120 IF DV<>1 AND DV<>2 THEN 2090
2130 IF DV=2 THEN OPEN 1,PQ: CMD 1
2140 RETURN
2160 IF DV=2 THEN PRINT#1,:CLOSE 1
2170 RETURN
2180 FOR I1=1 TO M
2190 FOR J1=1 TO M
2200 PRINT S6$; M$(I1,J1);
2210 NEXT J1
2220 PRINT S9$
2230 NEXT I1
2235 PRINT S6$;
2240 RETURN
```

—Testing and Using the Program

Many of the key sections of the program are set up as subroutines to facilitate testing and debugging the program.

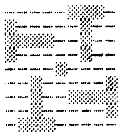
Perhaps the most useful is the puzzle printout subroutine (lines 1680-2080). To get a printout of the pattern, insert the command GOSUB 1680 at strategic points in the program. For instance, add line 335 GOSUB 1680 to see the initial pattern before the quality control check. Add line 615 GOSUB 1680 to see the pattern after the quality control check.

Typically, you will view the puzzle first and perhaps examine the word path directory. If you like the puzzle, you'll file it on disk for later use. Then you can erase and start over with a new random number and a new minimum word length.

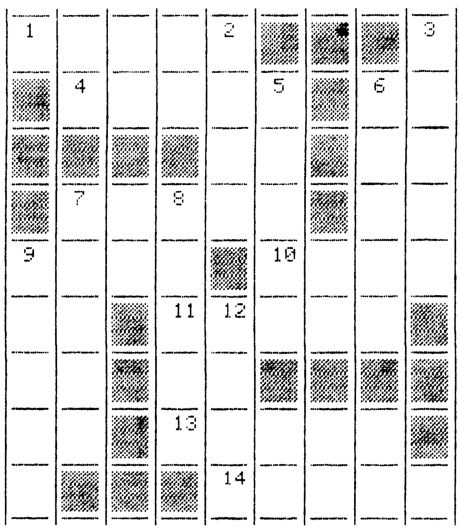
When generating small puzzles, the program may occasionally produce one that consists entirely of blocks. Simply erase the puzzle and generate a new one using a different random number seed. Use a minimum word length of 2 to reduce the chances of this happening.

—Printing Considerations

In printed form, each cell requires four columns and four lines. Adding one column for the right boundary and one line for the bottom boun-



PUZZLE # 32050 . 3



ACROSS

PATH #	LENGTH
1	5
4	5
7	5
9	4
10	4
11	5
13	5
14	5

DOWN

PATH #	LENGTH
2	4
3	5
5	5
6	5
7	5
8	5
9	5
12	4

Figure 4-3: Sample results of the program showing the miniature puzzle, the full-size puzzle, and the word directory.

dary, we have the following formulas for puzzle dimensions:

$$columns = 4 \times size + 1$$
$$lines = 4 \times size + 1$$

Accordingly, a 19 × 19 puzzle requires 77 columns and 77 lines in printed form, which almost fills an 8-1/2 × 11 sheet of paper.

Figure 4-3 shows sample results of the program using the Commodore MPS-801 printer. If you have another printer, you may need to make these changes:

```
133 S6$="": S9$=""
150 BK$="#": REM BLOCK CHARACTER
170 BL$="###"
180 VL$="!"
190 HL$="---"
```

Chapter 5 presents a program that reads and places words in the puzzle patterns created by this program.

Chapter 5

Crossword Puzzle Designer—Part 2

In the last chapter, you generated empty crossword puzzle patterns. In this chapter, you complete the puzzles by supplying answers and clues.

Selecting answers that fit together is often a tedious and time-consuming project. It can also be frustrating. Using Crossword Puzzle Designer, your computer eliminates much of the tedium, yet still allows you to make your own creative finishing touches. More than any other puzzle project in this book, Crossword Puzzle Designer involves a good deal of collaboration between you and your computer.

—How Words Are Stored in the Puzzle

The puzzle pattern in Figure 5-1 was created by the program developed in Chapter 4. The fill-in program presented in this chapter can recover the puzzle pattern directly from a disk file. But what if you don't have such a disk file? The fill-in program also gives you the option of storing the pattern recovery information in DATA lines in the program. DATA lines are explained later in this chapter.

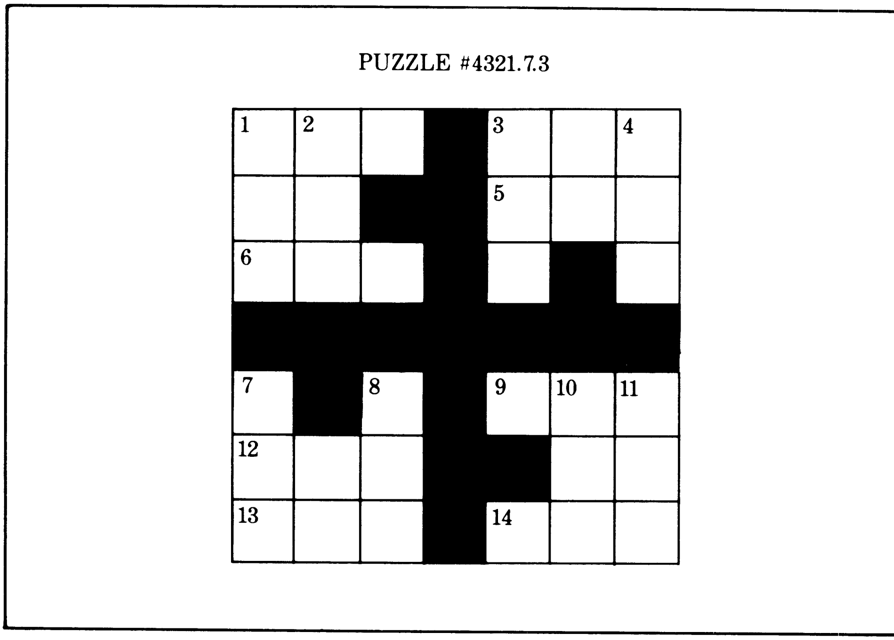


Figure 5-1. A puzzle pattern

Once you have recovered the pattern through a disk file or DATA lines, you can begin attempting to fill in words with the program. It reads in a word list that you provide and tries to fill each path from the word list. After it has tried all paths or used up all the words, the program lets you take over, filling in the gaps and changing any paths you don't like.

The result of this collaboration is a solved crossword puzzle; you must then make up the clues to go with the words. For example, if your puzzle has BIT in the path 1-across, you might use the clue "A binary digit" or "What the programmer did to the incompetent computer repairman."

Figure 5-2 shows the puzzle after the computer has automatically filled in words and before you have had a chance to perfect it. In this case, the computer used a word list of 67 common BASIC keywords.

Obviously, you must complete the paths that contain hyphens—the computer cannot fill these paths from its word list. Referring to Figure 5-2, you might use ARC in path 3-across; ARC isn't a BASIC keyword, but at least it can easily be related to programming. Path 12-across is

Solution

	1	2	3	4	5	6	7
1	A	S	C	#	A	-	C
2	B	G	#	#	T	-	O
3	S	N	-	#	N	#	S
4	#	#	#	#	#	#	#
5	D	#	D	#	D	I	M
6	E	-	E	#	#	N	O
7	F	-	L	#	-	T	D

Word Directory

ACROSS:	DOWN:
1 ASC	1 ABS
3 A-C	2 SGN
5 T-O	3 ATN
6 SN-	4 COS
9 DIM	7 DEF
12 E-E	8 DET
13 F-T	10 INT
14 -TD	11 MOD

Figure 5-2. A partial solution for the puzzle pattern in Figure 5-1

more challenging: E-E. If you can't come up with a computer word that fits this pattern, go ahead and broaden the scope a little. The word EWE fits, so use it. Later you can come up with a clever hint, however tenuous, that relates to programming.

Paths that are filled indirectly through the completion of intersecting paths create another problem. When the program fills a path, it does not check all the intersecting paths that may be affected. As a result, the puzzle may contain "words" that are unusable. You will have to make a number of path replacements to eliminate these nonwords.

Figure 5-3 shows the finished puzzle with a set of clues. Of course, this is just one way of completing it; you might find better words to use.

Notice that some of the words that were taken from the original word list have been replaced. For example, INT was changed to ISO and DET to DEN. Such replacements are required to eliminate non-words.

—The Program

The program uses numerous subroutine calls (for instance, line 70 ON S GOSUB 1840,2170). The subroutines are explained in a separate section after the full program has been presented.

Before trying this program, you should have read Chapter 4 and run the pattern generation program. That way you'll have some puzzle patterns stored in disk files and ready to use. If you want to create your

Across

1 BASIC function to get the ASCII code of a character
3 Inverse of the tangent function is the _____-tangent
5 Also, besides
6 Abbreviation for IBM's System Network Architecture
9 BASIC statement to create an array
12 Opposite of RAM
13 BASIC command to erase a resident program
14 Abbreviation for End of Data

Down

1 BASIC function that always returns nonnegative numbers
2 BASIC function that tells whether a number is positive, negative, or zero
3 Inverse of the BASIC TAN function
4 Opposite of the BASIC SIN function
7 Favorite resting place for programmers and lions
8 Found on the grass when programmers are going to bed and milkmen are making their deliveries
10 International Standards Organization
11 BASIC function that returns the remainder of x after integer division by m

Solution

	1	2	3	4	5	6	7
1	A	S	C	#	A	R	C
2	B	G	#	#	T	O	O
3	S	N	A	#	N	#	S
4	#	#	#	#	#	#	#
5	D	#	D	#	D	I	M
6	E	W	E	#	#	S	O
7	N	E	W	#	E	O	D

Word Directory

ACROSS:	DOWN:
1 ASC	1 ABS
3 ARC	2 SGN
5 TOO	3 ATN
6 SNA	4 COS
9 DIM	7 DEN
12 EWE	8 DEW
13 NEW	10 ISO
14 EOD	11 MOD

Figure 5-3. A perfected solution to the puzzle of Figure 5-1

own patterns manually, you should still read the text of Chapter 4. This chapter's fill-in program uses many of the same arrays as the pattern program.

Menu Options

The first block of the fill-in program prints a title and menu:

```
10 PRINT CHR$(147) "CROSSWORD PUZZLE FILL-IN"
20 PRINT
30 PRINT "1-LOAD PATTERN FROM DISK"
40 PRINT "2-READ PATTERN FROM DATA LINES"
50 INPUT S
60 IF S<> 1 AND S<>2 THEN 20
70 ON S GOSUB 1840,2170
```

The two menu options are to load pattern from disk and to read pattern from data lines. Use the first option in conjunction with puzzle patterns created by the program of Chapter 4; use the second option if you have stored the pattern recovery information in DATA lines.

Line 70 calls the pattern recovery subroutine that you select.

Storing the Word List

The following lines read in the word list and set up a few other variables and arrays:

```
80 PRINT
90 PRINT "READING THE WORD LIST"
100 READ NW
110 DIM WD$(NW),WL(NW),DI(2,2)
120 DI(1,1)=1
130 DI(1,2)=0
140 DI(2,1)=0
150 DI(2,2)=1
160 SC$="-"
165 PZ=4
170 FOR W=1 TO NW
180 READ WD$(W)
190 WL(W)=W
200 NEXT W
```

The words are contained in DATA lines at the end of the program. NW is the number of words. WD$( ) stores the words. WL( ) lists the index numbers of each available word. Initially, WL(W)=W for every W, since none of the words have been used yet. When a word WD$(W) is used, that index is removed from array WL( ).

DI(,) stores the direction increments for horizontal and vertical paths. The first subscript indicates the direction (1=vertical, 2=horizontal) and the second subscript indicates the increment (1=row increment, 2=column increment). For instance, DI(1,1) gives the row increment for vertical paths, and DI(1,2) gives the column increment for vertical paths. The values are 1 and 0 respectively (see Figure 5-4).

In line 165, PZ is the device number normally assigned to the printer. If your printer has another device number, change 4 in line 165 accordingly.

Filling In the Puzzle

The program now asks whether it should fill in all the paths without pausing or request your approval before inserting a word into a path.

```
210 PRINT
220 PRINT "STARTING TO FILL IN THE PUZZLE"
230 AF=0
240 PRINT "<C>ONTINUOUS FILL-IN OR"
250 PRINT "<R>EQUEST APPROVAL FOR EACH WORD?"
260 INPUT "ENTER <C> OR <R>";CR$
270 IF CR$="R" THEN 300
280 IF CR$<>"C" THEN 240
290 AF=1
```

The variable AF stores the choice you make (0=request approval for each path, 1=continuous fill-in).

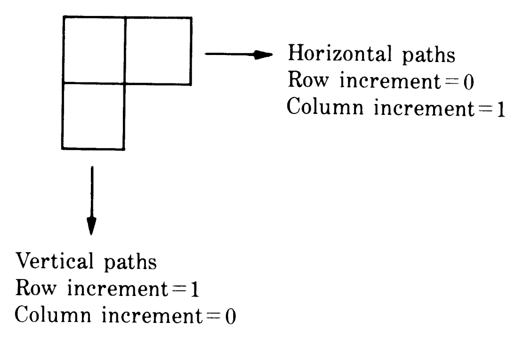


Figure 5-4. Illustration of the direction increments for paths

Filling the Paths

Now the program is ready to fill in the words. It tries every path number; for each path number, it tries both the vertical and horizontal directions if appropriate. The following lines select the path number and direction:

```
300 WU=0
310 REM
320 FOR P=1 TO PN
330 FOR D=1 TO 2
340 IF AD(P,D)=0 THEN 790
350 GOSUB 1700
360 IF SF=0 THEN 790
```

The variable WU stores the number of words used. P is the path number; it counts from 1 to PN, the highest path number used. For each path number, D counts from 1 to 2, which represents the two possible path directions (1=down, 2=across).

Line 340 determines whether a path exists in a given direction. If no path exists, the program jumps to 790, which checks the next path number. Refer to Chapter 4 for an explanation of the array AD(,).

If the path exists, the subroutine called in line 350 gets its present contents.

The variable SF indicates whether the path contains any spaces. If there are no spaces present (SF=0), the path is already complete, so the program skips to the next path (line 360).

Given a path containing spaces, the program attempts to fill it in from its word list.

```
370 IF WU=NW THEN 680
380 PRINT
390 PRINT "NEW PATH SELECTED."
400 PRINT "CHECKING THE WORD LIST..."
410 W=1
420 WT$=WD$(WL(W))
430 IF LEN(WT$)=PL THEN 470
440 W=W+1
450 IF W>NW-WU THEN 680
460 GOTO 420
470 GOSUB 1540
480 IF XF=1 THEN 440
490 GOSUB 1620
500 IF WT$=T$ THEN 440
510 PRINT "PLACED A WORD:"
520 IF AF=1 THEN 620
```

When WU=NW, all the words in the list have been used; in that case, the program skips to the next logical block (line 370). Otherwise, it tries to find a matching word starting with the first word available (W=1 in line 410).

If the length of the chosen word WD$(W) is not the same as the path length PL, the program skips to the next word (lines 430-460). If the lengths are the same, the subroutine called in line 470 checks whether WD$(W) can be plugged into the path without changing any of the letters that have been already filled in. The variable XF indicates whether a conflicting character exists. If it does, the program tries the next word (line 480). If the word fits, the subroutine called in line 490 plugs it into the path.

Upon return from the subroutine, line 500 checks to see whether the replacement string WT$ is the same as the path's original contents T$. The two are the same only under special circumstances explained later. If WT$ is not the same as T$, the program has indeed filled a path, and line 510 prints a message to that effect.

The Request-Approval Option

Line 520 checks whether you have selected the request-approval option. If you have, the following lines let you view and approve the latest word-path assignment:

```
530 PRINT
540 GOSUB 2300
550 PRINT
560 PRINT "<A>CCEPT OR <C>ANCEL THE WORD?"
570 INPUT "<A> OR <C>";AC$
580 IF AC$="A" THEN 620
590 IF AC$<>"C" THEN 490
600 WT$=T$
610 GOTO 490
```

The subroutine called in line 540 prints the puzzle in its current form. Lines 560-590 ask you whether you accept or want to cancel the change. If you cancel it, lines 600 and 610 restore the old contents of the path by setting WT$=T$ and returning to line 490, which puts WT$ into the current path. Upon returning from that subroutine, line 500 detects that a change was canceled and jumps to 440, which selects another word to try.

Updating the Word List

After a word has been placed in a path and accepted, the following lines remove the word from the list of words available:

```
620 IF W=WN-WU THEN 660
630 FOR I=W TO NW-WU-1
640 WL(I)=WL(I+1)
650 NEXT I
660 WU=WU+1
670 GOTO 790
```

WL(I) contains the index or subscript of the word just used. To eliminate that subscript from the list, every succeeding element in the list WL( ) is moved down one: WL(I+1) replaces WL(I), WL(I+2) replaces WL(I+1), and so forth.

Line 670 causes the program to select the next path in the puzzle.

Adding a Word Not on the List

What happens when the program cannot find a word that fits a given path? The following lines allow you to add a word not on the original list:

```
680 PRINT
690 PRINT "CAN'T FIND A MATCHING WORD."
700 IF AF=1 THEN 790
710 PRINT "CAN YOU HELP? THE PATH IS HIGHLIGHTED:"
720 PRINT
730 GOSUB 2300
740 PRINT
750 XC=0
760 GOSUB 2520
770 IF WT$="" THEN 790: REM NO SPACES IN QUOTES
780 GOSUB 1620
```

If you have specified the automatic fill-in option, the program skips to the next path without asking you for help with the current path (line 700). Otherwise, the program will ask you to fill in the path.

The subroutine called in line 730 prints the puzzle with the current path highlighted. Line 760 calls a subroutine to get your suggested word for the current path. Line 770 checks whether you indeed supplied a word; WT$=" " indicates that you did not supply a word in the subroutine at line 2520; in that case, the program skips to the next path.

If you did enter a replacement word, the subroutine called in line 780 plugs it into the puzzle.

The following two lines select the next direction and the next path number:

```
790 NEXT D
800 NEXT P
```

A Chance to Correct

After trying all combinations of directions D and paths P, the program gives you a chance to perfect the puzzle by modifying any path you choose. Here is the puzzle-perfection menu:

```
810 PRINT
820 PRINT "PUZZLE COMPLETE"
830 PRINT
840 PRINT "1-PRINT OR REVISE PUZZLE"
850 PRINT "2-PRINT THE WORD DIRECTORY"
860 PRINT "3-SAVE THE PUZZLE ON DISK"
870 PRINT "4-END
880 INPUT "SELECT 1-4"; SE
890 IF SE<1 AND SE>4 THEN 810
900 ON SE GOTO 910,1170,1310,1530
```

The menu options are to print or revise the puzzle, print the word directory, save the puzzle on disk, or end. Line 900 jumps to the logic corresponding to the option you select.

Printing the Puzzle

Here's the puzzle-printing logic:

```
910 PRINT
920 GOSUB 2650
930 P=0
940 PRINT
950 GOSUB 2300
960 PRINT
970 GOSUB 2710
980 PRINT "ENTER STARTING ROW AND COLUMN"
990 PRINT "OF PATH TO BE CHANGED"
1000 INPUT "ROW,COL (ENTER 0,0 TO QUIT)";R,C
1010 IF R<1 OR R>M OR C<1 OR C>M THEN 830
1020 INPUT "ENTER DIRECTION (1=DOWN,2=ACROSS)";D
1030 IF D<>1 AND D<>2 THEN 1020
```

```
1040 P=PL(R,C)
1050 IF AD(P,D)>0 THEN 1080
1060 PRINT "NO SUCH PATH. TRY AGAIN"
1070 GOTO 980
1080 P=PL(R,C)
1090 GOSUB 1700
1100 PRINT "THE PATH IS HIGHLIGHTED"
1110 PRINT
1120 GOSUB 2300
1130 PRINT
1140 XC=1
1150 GOSUB 2520
1160 GOTO 830
```

The subroutine called in line 920 lets you select either the display or the printer for output. If you have a printer handy, you should get a hard copy of the puzzle before you begin changing it. The subroutine called in line 950 prints the current maze, and the subroutine called in line 970 restores the display as the output device.

Lines 980-1070 let you specify a path in terms of the path's starting row and column and its direction.

The subroutine called in line 1090 builds up a copy of the path's contents, and the subroutine called in line 1120 prints the puzzle on the display with the specified path number P, direction D highlighted.

The subroutine called in line 1150 lets you correct a path's contents or leave the path as is. Line 1160 returns to the menu.

Here's the logic to print the contents of all paths (print the word directory):

```
1170 GOSUB 2650
1180 PRINT
1190 FOR D=2 TO 1 STEP -1
1200 IF D=2 THEN PRINT "ACROSS:"
1210 IF D=1 THEN PRINT "DOWN:"
1220 FOR P=1 TO PN
1230 GOSUB 1700
1240 IF PL=0 THEN 1260
1250 PRINT P;" ";T$: REM 1 SPACE IN QUOTES
1260 NEXT P
1270 PRINT
1280 NEXT D
1290 GOSUB 2710
1300 GOTO 810
```

The subroutine called in line 1170 selects the output device. Lines 1190-1280 print the contents of each path; the paths across are printed first.

The subroutine called in line 1290 restores the display as the output device, and line 1300 returns to the menu.

The following lines store the puzzle data in a disk file. You can reload the same disk file later on to do more work on it.

```
1310 INPUT "NAME THE OUTPUT FILE";FO$
1320 PRINT "FILING PUZZLE IN " FO$
1330 OPEN 1,8,4,"@0:"+FO$+",SEQ,WRITE"
```

Line 1310 prompts you to name the output file. Line 1330 create the file, erasing any preexisting file with that name.

These lines store the data:

```
1370 PRINT#1,M
1380 FOR I=1 TO M
1390 FOR J=1 TO M
1400 PRINT#1,M$(I,J)
1410 PRINT#1,PL(I,J)
1420 NEXT J,I
1440 PRINT#1,PN
1450 FOR I=1 TO PN
1460 PRINT#1,AD(I,1)
1470 PRINT#1,AD(I,2)
1480 PRINT#1,RC(I,1)
1490 PRINT#1,RC(I,2)
1500 NEXT I
1510 CLOSE 1
1520 GOTO 810
```

Lines 1370-1500 print the data in the same sequence that was used by the pattern program so you can reload the puzzle from disk later on. Line 1520 returns to the menu.

Here's the logic to end the program:

```
1530 END
```

—Subroutines

Much of the program logic is placed in subroutines to shorten the program and facilitate debugging. There are nine subroutines in all.

The following subroutine compares a word WT$ with a path's contents T$. If WT$ fits into the path without any conflicts, the subroutine returns 0 in variable XF. If there are conflicts, it returns 1 in XF.

```
1540 XF=0
1550 FOR QI=1 TO PL
1560 WC$=MID$(WT$,QI,1)
1570 TC$=MID$(T$,QI,1)
1580 IF TC$=SC$ OR WC$=TC$ THEN 1600
1590 XF=1
1600 NEXT QI
1610 RETURN
```

The next subroutine replaces the contents of path number P, direction D (determined by row increment IR and column increment IC) with WT$.

```
1620 R=RC(P,1)
1630 C=RC(P,2)
1640 FOR QI=1 TO PL
1650 M$(R,C)=MID$(WT$,QI,1)
1660 R=R+IR
1670 C=C+IC
1680 NEXT QI
1690 RETURN
```

The following subroutine builds up a string T$ containing the contents of a path. SF=0 indicates that the path contains no spaces (that is, it is already filled in).

```
1700 PL=AD(P,D)
1710 IR=DI(D,1)
1720 IC=DI(D,2)
1730 R=RC(P,1)
1740 C=RC(P,2)
1750 T$="": REM NO SPACES IN QUOTES
1760 SF=0
1770 FOR I=1 TO PL
1780 T$=T$+M$(R,C)
1790 IF M$(R,C)=SC$ THEN SF=1
1800 R=R+IR
1810 C=C+IC
1820 NEXT I
1830 RETURN
```

Here's the subroutine that loads a puzzle pattern from a disk file:

```
1840 PRINT "SPECIFY THE CHARACTER USED TO INDICATE"
1850 PRINT "A BLOCK (PRESS RETURN IF NOT KNOWN)";
1855 BK$="": REM NO SPACES IN QUOTES
1860 INPUT BK$
1870 NK$="": REM NO SPACES IN QUOTES
1880 IF BK$="" THEN 1940: REM NO SPACES IN QUOTES
```

```
1890 PRINT "SPECIFY THE NEW CHARACTER TO USE"
1900 PRINT "FOR BLOCKS (RETURN TO LEAVE AS-IS) ";NK$
1910 INPUT NK$
1920 IF NK$="" THEN NK$=BK$: REM NO SPACES IN QUOTES
1940 INPUT "VIEW DISK DIRECTORY (Y/N/Q)";YN$
1942 IF YN$="N" THEN 1945
1943 IF YN$="Q" THEN STOP
1944 GOSUB 2900: REM GET DISK DIRECTORY
1945 PRINT "ENTER THE PATTERN FILE NAME"
1950 INPUT FI$
1960 OPEN 1,8,4,FI$
1970 INPUT#1,M
1971 IF ST=0 THEN 1980
1972 PRINT "FILE NOT FOUND": CLOSE 1: PRINT:
     GOTO 1940
1980 DIM M$(M,M),PL(M,M)
1990 FOR I=1 TO M
2000 FOR J=1 TO M
2010 INPUT#1, C$
2020 IF C$=BK$ THEN C$=NK$
2030 M$(I,J)=C$
2040 INPUT#1, PL(I,J)
2050 NEXT J,I
2070 INPUT#1, PN
2080 DIM AD(PN,2),RC(PN,2),PQ(PN)
2090 FOR I=1 TO PN
2100 INPUT#1, AD(I,1)
2110 INPUT#1, AD(I,2)
2120 INPUT#1, RC(I,1)
2130 INPUT#1, RC(I,2)
2140 NEXT I
2150 CLOSE 1
2160 RETURN
```

Lines 1840-1920 let you replace the block character used in the disk file with another of your own choice. If you are going to print the puzzle, you may need to use this option. For example, the pattern program (Chapter 4) uses CHR$(166) as a block character. However, many printers won't print a block for this character. In that case, you can tell the program to replace the backspace character with an alternate, such as a number sign (#) or an asterisk (*).

Line 1940 gives you the option of viewing the disk file directory before naming the file you want to load. You may also type Q to stop the program at this time. Lines 1945 and 1950 prompt you to specify the name of the pattern file. If you are using the results of the Puzzle Pattern Designer program, the file will be named XWORD.*xx.xx*. If you are recovering a file previously generated with this fill-in program, specify the corresponding name.

Line 1971 checks for disk error; in case of an error the program will give you another chance to specify the file name or stop the program.

Here's the subroutine to read a pattern from DATA lines:

```
2170 READ M
2180 DIM M$(M,M),PL(M,M)
2190 FOR I=1 TO M
2200 FOR J=1 TO M
2210 READ M$(I,J),PL(I,J)
2220 NEXT J,I
2240 READ PN
2250 DIM AD(PN,2),RC(PN,2)
2260 FOR I=1 TO PN
2270 READ AD(I,1),AD(I,2),RC(I,1),RC(I,2)
2280 NEXT I
2290 RETURN
```

If you select this option, you must have previously stored the appropriate data in the line range 2721 to 2899.

The data must be arranged in the following way:

1. The puzzle size M.
2. Two numbers for each puzzle cell: M$(I,J) (the contents of the cell—a block, a letter, or a blank); and PL(I,J) (the number of the path originating in that cell—0 if no path originates there).

   There are M × M puzzle cells, and the relevant data must be arranged as:

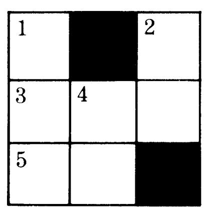
   M$(1,1), PL(1,1)
   M$(1,2), PL(1,2)
   M$(1,3), PL(1,3)
   .
   .
   .
   M$(1,M), PL(1,M)
   M$(2,1), PL(2,1)
   .
   .
   .
   M$(M,M), PL(M,M).
3. The number of paths PN.

4. For each path number P=1 through PN, there must be four numbers:

   AD(P,1) (the length of the path down—0=no path)
   AD(P,2) (the length of the path across—0=no path)
   RC(P,1) (the number of the row containing the head cell, the first cell in the path)
   RC(P,2) (the number of the column containing the head cell).

This order may seem somewhat tedious to maintain; nevertheless, there may be times when you'll want to use the manual procedure. Figure 5-5 shows a sample set of DATA lines you can use to try this option.

The presentation-quality puzzle



The corresponding puzzle recovery data

```
2800 DATA 3
2810 DATA -,1,#,0,-,2
2820 DATA -,3,-,4,-,0
2830 DATA -,5,-,0,#,0
2840 DATA 5
2850 DATA 3,0,1,1
2860 DATA 2,0,1,3
2870 DATA 0,3,2,1
2880 DATA 2,0,2,2
2890 DATA 0,2,3,1
```

Solution

	1	2	3
1	A	#	F
2	B	I	N
3	S	F	#

Word Directory

ACROSS:	DOWN:
3 BIN	1 ABS
5 SF	2 FN
	4 IF

Figure 5-5. A completed 3 × 3 puzzle generated with DATA lines 2800-2890 as listed

The next subroutine prints the puzzle in an M × M grid, adding column and row numbers along the margins for reference.

```
2300 QR=RC(P,1)
2302 QC=RC(P,2)
2304 QL=0
2310 PRINT "    " : REM 4 SPACES IN QUOTES
2320 FOR QI=1 TO M
2330 PRINT RIGHT$(STR$(QI),1);
2340 NEXT QI
2350 PRINT
2360 PRINT
2370 FOR QI=1 TO M
2380 PRINT LEFT$(STR$(QI)+"  ",4);: REM 2 SPACES
     IN QUOTES
2390 FOR QJ=1 TO M
2400 IF QL=PL OR QI<>QR OR QJ<>QC THEN 2470
2410 PRINT CHR$(18);: REM REVERSE ON
2420 PRINT M$(QI,QJ);
2430 PRINT CHR$(146);: REM REVERSE OFF
2440 QR=QR+IR
2450 QC=QC+IC
2455 QL=QL+1
2460 GOTO 2480
2470 PRINT M$(QI,QJ);
2480 NEXT QJ
2490 PRINT
2500 NEXT QI
2510 RETURN
```

The following subroutine asks you to enter a word to be placed in the current path. If the word you enter is the wrong length, it will be rejected. If the word has conflicting characters, it will also be rejected, except when XC=1. This is true during the perfection phase of the program, when you can change a path's contents even if it has been completely filled in.

```
2520 PRINT "ENTER A WORD ( LENGTH="PL")"
2530 PRINT "OR AN EMPTY LINE TO SKIP THE PATH"
2535 WT$="": REM NO SPACES IN QUOTES
2540 INPUT WT$
2550 IF WT$="" THEN RETURN
2560 IF LEN(WT$)=PL THEN 2590
2570 PRINT "WRONG LENGTH. TRY AGAIN."
2580 GOTO 2520
2590 GOSUB 1540
2600 IF XF=0 OR XC=1 THEN 2630
2610 PRINT "CONFLICTING CHARACTER. TRY AGAIN."
```

```
2620 GOTO 2520
2630 GOSUB 1620
2640 RETURN
```

Here are the subroutines that select the output device (lines 2650-2700) and restore the CRT as the output device (lines 2710-2720).

```
2650 DV=1
2660 INPUT "SELECT 1-DISPLAY 2-PRINTER ";DV
2670 IF DV<>1 AND DV<>2 THEN 2660
2680 IF DV=2 THEN OPEN 1,PZ: CMD 1
2700 RETURN
2710 IF DV=2 THEN PRINT#1,: CLOSE 1
2720 RETURN
```

The following subroutine loads the disk file directory and prints it on the screen:

```
2900 PRINT "LOADING DIRECTORY..."
2901 OPEN 1,8,4,"$,SEQ,READ"
2902 IW=0
2910 IF ST=64 THEN 2980
2920 GET#1,A$
2922 IF LEN(A$)=0 THEN 2902
2930 IF A$>CHR$(31) AND A$<CHR$(128) THEN 2938
2932 IF IW=0 THEN 2910
2934 IW=0
2935 PRINT
2937 GOTO 2910
2938 IW=1
2939 PRINT A$;
2940 GOTO 2910
2980 CLOSE 1
2985 PRINT
2990 RETURN
```

Line 2930 sifts out all the nontext characters from the file directory information. Line 2935 prints a carriage return after each sequence of textual information. When viewing the directory, you will have to ignore certain spurious information that appears on the first few lines of the directory list. However, you'll be able to recognize when the listing of valid file names begins. There will be several delays during the listing process while the program sifts through all the extraneous data. Note that this subroutine does not erase the resident program from memory as the ordinary command LOAD "$",8 does.

DATA Lines

All the data items are placed at the end of the program listing. If you are going to read a puzzle pattern from DATA lines, you must put the appropriate data in lines ranging from 2721 to 2899. (See Figure 5-5 for sample lines to use.) If you are going to load data from a disk file, you cannot put any pattern recovery data in the program.

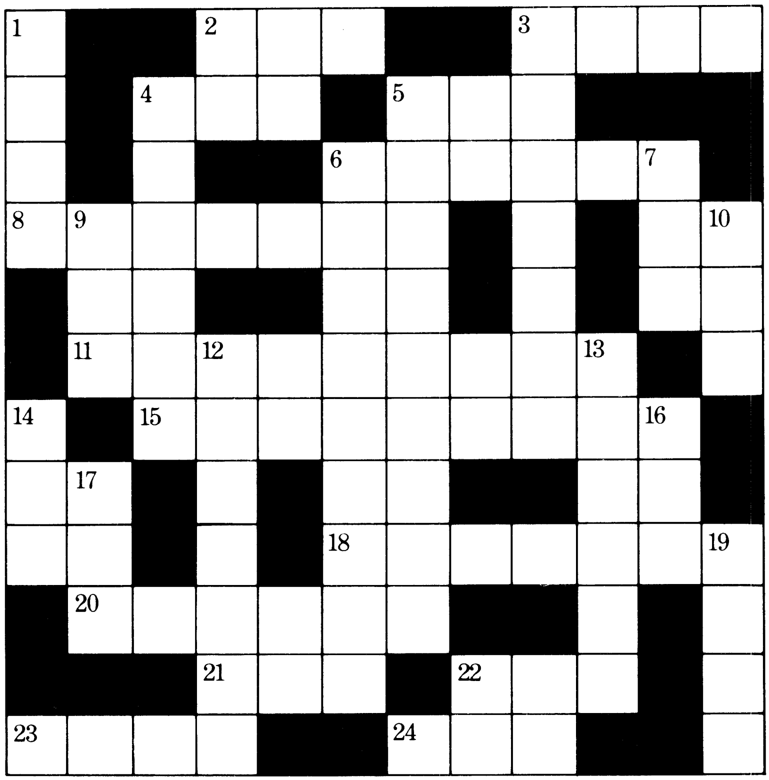
The word list starts at line 3000. It consists of the word count NW followed by the words. For instance, if you have 100 words, line 3000 should be 3000 DATA 100. Here is the word list used to fill in the puzzles:

```
3000 DATA 65
3010 DATA ABS,ASC,ATN,CALL,CLR,CONT
3020 DATA COS,DATA,DEF,DET,DIM,END,EXP
3030 DATA FN,FOR,FRE,GET,GOSUB,GOTO,IF
3040 DATA INPUT,INT,LEN,LET,LIST,LOAD
3050 DATA LOG,MOD,NEW,NEXT,NOT,ON,OR
3060 DATA PEEK,POKE,POS,PRINT,READ,REM
3070 DATA RESTORE,RESUME,RETURN,RND,RUN
3080 DATA SAVE,SGN,SIN,SQR,STEP,STOP
3090 DATA TAB,TAN,THEN,TO,TRACE,USR,VAL
3100 DATA BASIC,KEYWORD,ADDITION
3110 DATA MULTIPLICATION,BINARY
3120 DATA INTEGER,EXPONENTIATION
3140 DATA PRECISION
```

—Using the Program

Here's the typical sequence for using the fill-in program in conjunction with the pattern program of Chapter 4:

1. Run the pattern program and save the pattern in a disk file XWORD *xxxx.xx*. Get a printout of the puzzle.
2. Type in this chapter's fill-in program using the word list provided or replacing it with your own.
3. Run the program. Select the disk file option, and load the data from XWORD *xxxx.xx*.
4. Select the continuous fill-in option for speed or the request-approval option for your own curiosity to help you check that the program is working properly.
5. After the program has tried to fill all the paths, print the puzzle.



Hints

ACROSS

2 BASIC function that always returns a nonnegative number
3 BASIC statement that gets data from DATA statements
4 BASIC statement that tells the computer to ignore what follows
5 Equally at home in a math textbook and on a dessert plate
6 McA
8 Good fuel for programmers, spelled the Italian way
11 Short for "No more kites in our inventory until Rover returns"
15 What Lady Augusta Ada Byron, the Countess of Lovelace, exclaimed upon seeing her first stallion

Figure 5-6. A ready-to-use puzzle

18 Ancient vessel for storing soda pop
20 The sound of many cows
21 BASIC function to generate random numbers
22 BASIC function for measuring strings
23 BASIC program that retrieves programs into memory
24 Logically, the last word in a BASIC program

DOWN

1 Programmer's word for "invoke"
3 BASIC statement to allow re-reading of DATA
4 BASIC statement to return from an error-handling routine
5 Based on the maximum number of digits a computer can store for a single number
6 The programmer applied his brakes too hard and went ———— — ————
7 BASIC function gets the code for a character
9 BASIC function is the inverse of TAN
10 BASIC function is the opposite of SIN
12 Reserved word of a computer language
13 BASIC statement at the end of a subroutine
14 BASIC statement used to set up an intrinsic function; also commonly found in roadside signs: WATCH OUT FOR ___ DOG
16 Common command to list disk files; when spoken indicates confusion or dumbfounded condition
17 BASIC statement to create an array
19 BASIC statement to close a loop

Solution

	1	2	3	4	5	6	7	8	9	0	1	2
1	C	#	#	A	B	S	#	#	R	E	A	D
2	A	#	R	E	M	#	P	I	E	#	#	#
3	L	#	E	#	#	I	R	I	S	H	A	#
4	L	A	S	A	G	N	E	#	T	#	S	C
5	#	T	U	#	#	T	C	#	O	#	C	O
6	#	N	M	K	I	O	I	U	R	R	#	S
7	D	#	E	E	K	A	S	T	E	E	D	#
8	E	D	#	Y	#	S	I	#	#	T	I	#
9	F	I	#	W	#	K	O	L	A	U	R	N
10	#	M	O	O	D	I	N	#	#	R	#	E
11	#	#	#	R	N	D	#	L	E	N	#	X
12	L	O	A	D	#	#	E	N	D	#	#	T

Figure 5-6. A ready-to-use puzzle (*continued*)

6. Make any changes necessary to complete and perfect the puzzle.
7. Print out a final copy of the puzzle solution and the word directory.
8. Make up clues for the word list.
9. Assemble your clues with the earlier puzzle, and you have a complete puzzle. The condensed puzzle from Step 7 represents the solution to the puzzle.

Figure 5-6 is a complete puzzle package consisting of the presentation puzzle, a set of clues, and a solution.

The steps for creating a puzzle without a prepared pattern disk file are the same, except that you must prepare the data lines as in Figure 5-5. You will also have to create a presentation puzzle.

Chapter **6**

The Codebreaker

With the Codebreaker program, you compete against your computer in a game known as "Bulls and Cows" or "Mastermind."

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 scores the codebreaker, who uses this information to come up with another guess. The object of the game is to guess the code in as few tries as possible.

Codes are made up of a sequence of four letters 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 6-1 gives several examples of scoring. Take a minute to study the sample guesses and scores to be sure you understand the scoring system.

Table 6-1. 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 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 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 codebreaker and letting the computer score each player. The player who guesses the secret code in the fewest tries wins the round.

Figure 6-1 shows a sample run of the program.

—Secrets of Codebreaking

Most players eventually come up with some kind of system for guessing. The Codebreaker has its own method too: the program makes its first

```
THE CODEBREAKER

ENTER A RANDOM NUMBER  335

ONE MOMENT PLEASE . . .

GUESS, KEEP SCORE, OR QUIT?
ENTER <G> <S> OR <Q> G

I HAVE A SECRET 4-DIGIT CODE,
CONSISTING OF THE SYMBOLS 'ABCD'.
ANY SYMBOL MAY REPEAT.
HERE ARE EXAMPLES: AAAA DCBA DACC

INPUT YOUR 4-DIGIT GUESS ABCD

HERE IS THE SCORING RECORD

GUESS    TRIAL     HIT(S)    MISS(ES)
NO.      CODE
 1       ABCD       3         0

INPUT YOUR 4-DIGIT GUESS BBCD

HERE IS THE SCORING RECORD

GUESS    TRIAL     HIT(S)    MISS(ES)
NO.      CODE
 1       ABCD       3         0
 2       BBCD       4         0
YOU HAVE GUESSED THE CODE IN  2  TRIES!

GUESS, KEEP SCORE, OR QUIT?
ENTER <G> <S> OR <Q> S

MAKE UP A SECRET 4-DIGIT CODE
CONSISTING OF THE SYMBOLS 'ABCD'.
ANY SYMBOL MAY REPEAT.
PRESS <RETURN> AFTER YOU HAVE WRITTEN
DOWN YOUR SECRET CODE
CAN I SCORE MYSELF (Y/N) N

MY GUESS IS AAAA
ENTER HITS, MISSES  1 , 0
*********************
```

Figure 6-1. Sample run of The Codebreaker

```
MY GUESS IS ABBB
ENTER HITS, MISSES  3 , 0
*
MY GUESS IS ABBC
ENTER HITS, MISSES  4 , 0
DID IT IN  3  TRIES!

GUESS, KEEP SCORE, OR QUIT?
ENTER <G> <S> OR <Q> Q


READY.
```

Figure 6-1. Sample run of The Codebreaker (*continued*)

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. The computer assumes the hypothesis is correct and scores each of its previous guesses against it. 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.

If you make a mistake in scoring, the program cannot test its hypothesis properly. Eventually, it tries all possible codes without finding one that is consistent with the previous scoring. In that case, the program asks you to check your scores and correct the error.

—The Program

The first block of the Codebreaker program prints a title:

```
10 PRINT CHR$(147);: REM CLEAR SCREEN
20 PRINT "THE CODEBREAKER"
30 PRINT
40 INPUT "ENTER A RANDOM NUMBER";X
50 X=RND(-ABS(X))
60 PRINT
```

```
70 REM
80 PRINT "ONE MOMENT PLEASE . . ."
90 REM
```

Lines 40 and 50 let you set the random number generator so the program won't start with the same secret code each time you run it.

Storing Hits, Misses, and Codes

The next lines construct a list of all possible codes:

```
100 LG=10
110 DIM P$(256),GU$(LG),S1(LG),S2(LG)
120 DG$="ABCD"
130 FOR P1=1 TO 4
140 FOR P2=1 TO 4
150 FOR P3=1 TO 4
160 FOR P4=1 TO 4
170 IX=(P1-1)*64 + (P2-1)*16 + (P3-1)*4 + P4
180 P$(IX)=MID$(DG$,P1,1) + MID$(DG$,P2,1)
    + MID$(DG$,P3,1) + MID$(DG$,P4,1)
190 NEXT P4,P3,P2,P1
```

LG is the maximum number of guesses you are allowed before the computer reveals the secret code. 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$(1) stores the first guess; S1(1) stores the number of hits assigned to that guess, and S2(1) stores the number of misses. DG$ contains the four characters that can be used in codes.

Lines 130-190 generate all possible codes in the following order: AAAA, AAAB, AAAC, AAAD, AABA, AABB, AABC, AABD, AACA, and so forth, up to DDDD. Those familiar with counting in different bases will recognize that the computer is counting from 0 to 255 in base 4 using A for 0, B for 1, C for 2, and D for 3.

Printing the Menu Options

The next lines print a menu on the screen:

```
200 PRINT CHR$(147);: REM CLEAR SCREEN
210 PRINT
220 PRINT "GUESS, KEEP SCORE, OR QUIT?"
230 INPUT "ENTER <G> <S> OR <Q>";C$
```

```
240 IF C$="G" THEN 280
250 IF C$="S" THEN 540
260 IF C$<>"Q" THEN 210
270 END
```

The menu offers you three options: guess (act as the codebreaker), score (act as the codemaker), or quit.

The Guess Option

In the next block of lines, the computer randomly selects a secret code and presents instructions for guessing:

```
280 GN=0
290 CR=INT(RND(1)*256) + 1
300 CD$=P$(CR)
310 PRINT CHR$(147): REM CLEAR SCREEN
320 PRINT "I HAVE A SECRET 4-DIGIT CODE,"
330 PRINT "CONSISTING OF THE SYMBOLS '";DG$;"'."
340 PRINT "ANY SYMBOL MAY REPEAT."
350 PRINT "HERE ARE EXAMPLES: AAAA DCBA DACC"
```

GN stores the latest guess number. (It is set at 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.

Checking Your Guesses

The following lines accept your guesses and score them until you guess correctly or run out of chances:

```
360 PRINT
370 INPUT"INPUT YOUR 4-DIGIT GUESS";GU$
380 IF LEN(GU$)<>4 THEN 360
390 GN=GN+1
400 GU$(GN)=GU$
410 A$=CD$
420 Q$=GU$
430 GOSUB 1210
440 S1(GN)=S1
450 S2(GN)=S2
460 GOSUB 1110
470 IF S1(GN)<>4 THEN 500
480 PRINT "YOU HAVE GUESSED THE CODE
    IN ";GN;" TRIES!"
490 GOTO 210
500 IF GN<LG THEN 360
510 PRINT "NO MORE GUESSES LEFT."
```

```
520 PRINT "MY SECRET CODE IS ";CD$
530 GOTO 210
```

The subroutine called in line 430 scores your guess. Upon return from the subroutine, S1 contains the number of hits, and S2, the number of misses. These values are saved in S1(GN) and S2(GN).

The subroutine called in line 460 prints a cumulative scoring record starting with your first guess and ending with your most recent one.

Line 470 determines whether you have guessed the secret code yet. When S1=4, all characters in your guess are hits and you have guessed the secret code.

If you haven't guessed the code correctly, line 500 determines whether you have reached LG, the maximum number of guesses allowed. If you haven't, the program prompts you to make your next guess.

The Score Option

In case you choose the "score" option, the following block of lines takes over, printing the instructions and prompting you to select a secret code:

```
540 PRINT CHR$(147): REM CLEAR SCREEN
550 PRINT "MAKE UP A SECRET 4-DIGIT CODE"
560 PRINT "CONSISTING OF THE SYMBOLS '";DG$;"'."
570 PRINT "ANY SYMBOL MAY REPEAT."
580 PRINT "PRESS <RETURN> AFTER YOU HAVE WRITTEN"
590 PRINT "DOWN YOUR SECRET CODE";
600 INPUT RT$
605 SS$="N"
610 INPUT "CAN I SCORE MYSELF (Y/N)";SS$
620 IF SS$<>"Y" THEN 640
630 INPUT "ENTER YOUR SECRET CODE ";CD$
```

Line 610 gives you the choice of scoring the computer's guesses or letting the computer score itself. To make this latter option possible, you must reveal your secret code to the computer (line 630). There's no cause for alarm, however; the part of the computer that guesses never "talks" to the part that scores.

The Computer's Initial Guess

Now the computer is ready to make its first guess:

```
640 GN=1
650 PN=1
660 PRINT
```

```
670 GU$(GN)=P$(PN)
680 PRINT "MY GUESS IS ";GU$(GN)
690 IF SS$<>"Y" THEN 750
700 A$=CD$
710 Q$=GU$(GN)
720 GOSUB 1210
730 PRINT "SCORE IS ";S1;" HIT(S) AND
    ";S2;" MISS(ES)."
740 GOTO 760
750 INPUT "ENTER HITS, MISSES"; S1,S2
760 IF (S1>=0) AND (S1<=4) AND (S2>=0) AND
    (S2<=4) AND (S1+S2<=4) THEN 790
770 PRINT "SCORING ERROR, PLEASE REDO"
780 GOTO 680
790 S1(GN)=S1
800 S2(GN)=S2
810 IF S1<>4 THEN 840
820 PRINT "DID IT IN ";GN;" TRIES!"
830 GOTO 210
```

GN is the guess number and is 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$( ). This is the arbitrary guess referred to previously.

Lines 700-740 perform the self-scoring procedure, while lines 750-780 perform the manual scoring procedure. Line 760 checks for impossible combinations of hits and misses.

Lines 790 and 800 save the hits and misses assigned to the latest guess. Line 810 determines whether the latest guess was incorrect (S1 <> 4). If it was incorrect, the program will attempt to guess again.

The Computer's Subsequent Guesses

The following lines allow the computer to come up with subsequent guesses based on previous scoring:

```
840 PN=PN+1
850 IF PN>256 THEN 1000
860 PRINT "*";
870 FL=0
880 FOR IH=1 TO GN
890 Q$=GU$(IH)
900 A$=P$(PN)
910 GOSUB 1210
920 IF S1=S1(IH) AND S2=S2(IH) THEN 950
```

```
930 IH=GN
940 FL=1
950 NEXT IH
960 IF FL=1 THEN 840
970 PRINT
980 GN=GN+1
990 GOTO 670
```

Line 840 increments the pattern number. When PN>256, all patterns have been tried without success, and a scoring error has been made. In that case, line 850 jumps to an error-handling routine. If PN<=256, P$(PN) becomes the computer's next hypothesis.

Line 860 prints an asterisk on the screen each time the program adopts a new hypothesis. This is to reassure you that the computer is working during the sometimes lengthy pauses.

Lines 880-950 test the hypothesis by reviewing the previous guesses, scoring each guess under the assumption that the hypothesis is correct, and comparing the resulting 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 (line 920), the program rejects the hypothesis and moves on to the next one (line 960). If a hypothesis produces no conflicts, it is accepted and used as the next guess (lines 980 and 990).

Scoring Errors

If the program tries all 256 possible codes without finding one consistent with your previous scoring, you have made a scoring error. The following lines let you review the scoring and back up to where the error occurred:

```
1000 PRINT
1010 PRINT "SCORING ERROR."
1020 PRINT "PLEASE REVIEW YOUR SCORING"
1030 PRINT "AND TYPE IN THE NUMBER OF THE"
1040 PRINT "IMPROPERLY SCORED GUESS."
1050 GOSUB 1110
1060 INPUT "IMPROPERLY SCORED GUESS=";MG
1070 IF MG<1 OR MG>GN THEN 1000
1080 GN=MG
1090 PN=1
1100 GOTO 680
```

The subroutine called in line 1050 prints the scoring record. Lines 1060 and 1070 let you specify the incorrectly scored guess. Line 1080 resets the guess number counter GN, and line 1090 resets the current pattern number PN. In line 1100, the program jumps back to the section that prints the computer's guess and asks you to score it. The computer then reasserts the guess GU$(GN) so you can score it again.

The Codebreaker program uses four subroutines in conjunction with its main program: one to print the scoring record, another to score guesses, a routine to modify a portion of a string, and a routine to search for a character within a string.

Scoring Record Subroutine

The following lines let you review the entire sequence of guesses and scores starting with the first guess:

```
1110 PRINT
1120 PRINT "HERE IS THE SCORING RECORD"
1130 PRINT
1140 PRINT "GUESS";TAB(8);"TRIAL";TAB(17);
     "HIT(S)";TAB(25);"MISS(ES)"
1150 PRINT "NO.";TAB(8);"CODE"
1160 FOR J=1 TO GN
1170 PRINT J;TAB(8);GU$(J);TAB(17);S1(J);
     TAB(25);S2(J)
1180 NEXT J
1190 RETURN
```

Scoring Subroutine

The scoring subroutine is made up of two parts. The first block finds all the hits (correctly placed characters):

```
1200 REM
1210 S1=0
1220 S2=0
1230 FOR J=1 TO 4
1240 IF MID$(Q$,J,1)<>MID$(A$,J,1) THEN 1360
1250 S1=S1+1
1260 ZA$=A$
1270 ZB$=" ": REM 1 SPACE IN QUOTES
1280 ZP=J
1290 GOSUB 1530
1300 A$=ZC$
1310 ZA$=Q$
1320 ZB$="X"
1330 ZP=J
```

```
1340 GOSUB 1530
1350 Q$=ZC$
1360 NEXT J
```

On entry to this subroutine, A$ contains the secret code and Q$ contains the guess. Lines 1240-1360 compare each character in A$ with the corresponding character in Q$. Whenever a match is found, the program increments the hit counter S1. In this case, the program must blot out the character that was a hit so it won't affect the scoring of misses later on. Lines 1260-1300 replace the hit character in A$ with a space, and lines 1310-1350 replace the hit character in Q$ with an X.

The second part of the scoring subroutine finds all the misses (incorrectly placed characters):

```
1370 FOR J=1 TO 4
1380 Z$=MID$(Q$,J,1)
1390 Q0=1
1400 Q1$=A$
1410 Q2$=Z$
1420 GOSUB 1600
1430 F=QF
1440 IF F=0 THEN 1510
1450 S2=S2+1
1460 ZA$=A$
1470 ZB$=" ": REM 1 SPACE IN QUOTES
1480 ZP=F
1490 GOSUB 1530
1500 A$=ZC$
1510 NEXT J
1520 RETURN
```

Lines 1370-1510 examine each character in the guess Q$ to see if that character can be found anywhere in the secret code A$. Remember that the hit characters have already been blotted out from both Q$ and A$.

Lines 1400-1430 search for character Z$ inside A$. In line 1430, F is the position at which Z$ is found; Z=0 indicates the character was not found. Each time a character is found, the program increments the "miss" counter S2 and blots out from A$ the character counted as a miss.

After checking all four characters in Q$, line 1520 ends the subroutine with a return to the main program.

String Replacement Subroutine

The following subroutine replaces a portion of a string. This function is used several times during scoring.

```
1530 ZC$="": REM NO SPACES IN QUOTES
1540 IF ZP=1 THEN 1560
1550 ZC$=LEFT$(ZA$,ZP-1)
1560 ZC$=ZC$+ZB$
1570 IF LEN(ZA$)-LEN(ZB$)-ZP+1=0 THEN RETURN
1580 ZC$=ZC$+RIGHT$(ZA$,LEN(ZA$)-LEN(ZB$)-ZP+1)
1590 RETURN
```

On entry to the subroutine, ZA$ contains the string to be changed; ZB$ contains the new information to be put into ZA$; and ZP contains the starting position for the replacement. Upon return from the subroutine, ZC$ contains the changed version of ZA$.

For a typical example, suppose ZA$="ABCD", ZB$=" ", and ZP=3. Upon return from the subroutine, ZC$="AB D".

String Search Subroutine

Here is the subroutine that searches for one string inside another string:

```
1600 QF=0
1610 IF LEN(Q2$)=0 THEN RETURN
1620 IF Q0+LEN(Q2$)-1>LEN(Q1$) THEN RETURN
1630 IF MID$(Q1$,Q0,LEN(Q2$))=Q2$ THEN 1660
1640 Q0=Q0+1
1650 GOTO 1620
1660 QF=Q0
1670 RETURN
```

On entry to the subroutine, Q1$ is the string to search through, Q2$ is the string to find, and Q0 is the position at which to begin the search. On return from the subroutine, QF is the position at which Q2$ begins in Q1$. QF=0 if Q2$ is not found in Q1$.

For a typical example, if Q1$="ABAB", Q2$="B", and Q0=3, the subroutine will end with QF=4.

—Testing and Using the Program

After entering the entire program and eliminating all typographical errors, test the scoring subroutine as follows:

Run the program. Select the Quit option from the menu. Now type in these lines without line numbers:

A$="BDBA"
Q$="BCAB"

```
GOSUB 1210
PRINT S1,S2
```

The computer should print the two numbers 1 and 2 (the number of hits and misses for guess "BCAB" when the secret code is "BDBA").

If you get some other values, check all the subroutines carefully for typographical errors.

When the program is running correctly, it should guess your secret code within four to six tries. The number of guesses required is determined by where the secret code is located in the computer's internal list of codes P$( ). With a little experimentation, you can find out which secret codes will take the computer the most tries to find.

This chapter is adapted from "The Code Breaker," by George Stewart, appearing in the December 1982 issue of *Popular Computing* magazine.

Chapter 7

Blackjack '84

Blackjack or Twenty-One is one of the most popular card games in gambling houses. The rules are simple, and winning is not too difficult. This chapter's program turns your computer into a Blackjack dealer so you can sharpen your skills without risking a thing—except, perhaps, a little pride.

The rules of Blackjack have been changed a little to keep the program from getting too long. This should be acceptable even to veteran Blackjack players, since there are dozens of variations of the game. This version is called Blackjack '84.

Object and Rules of Blackjack '84

You play Blackjack '84 against the computer, which is also referred to as the dealer. The object of the game is to acquire a hand that totals 21 or less without going over, or "busting." The hand with the highest total not exceeding 21 wins. Aces are worth 11 or 1 points; jacks, queens, and kings each have a value of 10; and the other cards are worth their index (2, 3, 4, and so forth). A card's suit (hearts, clubs, spades, and diamonds) has no effect on its value.

In playing against the computer you place bets using imaginary chips. At the start of the game, you have 100 chips. To start each hand, you must bet five chips, called the *ante*; this amount serves as the winnings pool. As play progresses, you may increase the size of the winnings pool.

The dealer and player both initially receive two cards. The player's cards are both visible, while only one of the dealer's cards is visible. In this way you can never tell precisely how good or bad the dealer's hand is. Since you are not playing against other bettors, where each player's cards are dealt face down so players cannot see each others' hands, having the values of both cards exposed doesn't matter. The dealer doesn't care what you have.

After totaling your first two cards, you have four options: increase your bet, receive another card (hit), stand on your present hand, or review the current status of the game.

You can continue hitting and betting until you bust or are satisfied with your total. If you bust, the round ends immediately, and the dealer takes the winnings without having to show his hand or draw additional cards. (That's one of the advantages of being dealer.)

If you stand, the dealer then takes a turn at improving his hand. However, the procedure for doing so is predetermined. If the dealer's total is less than 17, he must draw a card; if it is 17 or more, he must stand. A lack of flexibility is one of the disadvantages of being dealer and is your key to beating the computer.

After the dealer stands, the two hands are totaled and compared. If you win with a total of 21 (Blackjack), you receive triple the amount in the winnings pool. If you win with a total under 21, you receive double the amount in the pool. If you lose to the dealer or bust (draw a total over 21), the winnings pool goes to the dealer. If both hands have the same total, the hand containing the fewest cards (the lowest card count) wins. If the card counts are the same, the round is judged a tie; the bet remains on the table and a new hand is dealt.

Figure 7-1 shows a sample run of Blackjack '84.

—How the Deck Is Formed

The computer-dealer uses a standard 52-card deck, which is ordinary in every way except that it only exists in digital form in the computer's memory.

```
BLACKJACK '84

ENTER A RANDOM NUMBER  32050
ENTER YOUR NAME: SAM

SAM STARTS WITH  100  CHIPS.
THE ANTE IS  5
PRESS RETURN TO START
HERE'S THE FIRST ROUND . . .

SAM HAS 95 CHIPS LEFT
AFTER MAKING THE ANTE.

NEW HAND

SHUFFLING THE CARDS...STANDBY


DEALER'S HAND:
[??] [8♠]

SAM'S HAND:
[8♦] [A♣]

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: B
SAM HAS 95 CHIPS.
BET HOW MUCH NOW?  45

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: S

DEALER'S TURN: DEALER STANDS

PRESS RETURN TO REVEAL THE WINNER
SAM'S HAND:
[8♦] [A♣]
SAM'S SCORE IS  19

DEALER'S HAND:
[K♣] [8♠]
DEALER'S SCORE IS  18

SAM WINS  100
```

Figure 7-1. Sample run of Blackjack '84

```
SAM NOW HAS  150 CHIPS.

PLAY ANOTHER HAND? (Y/N) Y

SAM HAS 145 CHIPS LEFT
AFTER MAKING THE ANTE.

NEW HAND

DEALER'S HAND:
[??] [2♣]

SAM'S HAND:
[K♦] [9♥]

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: B
SAM HAS 145 CHIPS.
BET HOW MUCH NOW?  100

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: S

DEALER'S TURN: HIT
[7♠]

HIT
[10♣]

DEALER STANDS

PRESS RETURN TO REVEAL THE WINNER
SAM'S HAND:
[K♦] [9♥]
SAM'S SCORE IS  19

DEALER'S HAND:
[7♣] [2♣] [7♠] [10♣]
DEALER'S SCORE IS  26

DEALER BUSTS. SAM WINS  210

SAM NOW HAS  255 CHIPS.
```

Figure 7-1. Sample run of Blackjack '84 (*continued*)

```
PLAY ANOTHER HAND? (Y/N) Y

SAM HAS 250 CHIPS LEFT
AFTER MAKING THE ANTE.

NEW HAND

DEALER'S HAND:
[??] [6♦]

SAM'S HAND:
[7♦] [3♠]

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: H
[9♠]


BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: R
SAM HAS 250 CHIPS.
THE BET IS 5

DEALER'S HAND:
[??] [6♦]

SAM'S HAND:
[7♦] [3♠] [9♠]

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: B
SAM HAS 250 CHIPS.
BET HOW MUCH NOW?  100

BET, HIT, STAND, OR REVIEW CARDS?
ENTER B/H/S/R: S

DEALER'S TURN: HIT
[4♣]

HIT
[5♦]

DEALER STANDS
```

Figure 7-1. Sample run of Blackjack '84 (*continued*)

```
PRESS RETURN TO REVEAL THE WINNER
SAM'S HAND:
[7♦] [3♠] [9♣]
SAM'S SCORE IS  19

DEALER'S HAND:
[6♠] [6♣] [4♣] [5♣]
DEALER'S SCORE IS  21

DEALER WINS  105

SAM NOW HAS  150 CHIPS.
PLAY ANOTHER HAND? (Y/N) N
```

Figure 7-1. Sample run of Blackjack '84 (*continued*)

The deck is actually a 52-element array called D( ). D(52) is the top position on the deck; D(51) is one card down, and so forth. D(1) is the bottom position on the deck. The 52 distinct cards are represented by the numbers 0 through 51. Zero is the ace of hearts, 1 is the 2 of hearts, and so forth until 51, the king of spades. The computer shuffles the cards by storing the numbers 0 through 51 at random positions in the array D( ).

The computer keeps four additional lists. CP( ) keeps track of which cards are being held by the player or the dealer. F$( ) stores the 13 card names or indexes: ace, 2, 3, and so forth, through jack, queen, and king. S$( ) stores the four suits: hearts, diamonds, clubs, and spades. V( ) stores a numerical value assigned to each of the 12 indexes. For example, V(1) is 11, V(2) is 2, and V(13) is 10.

How does the computer translate the card numbers 0 to 51 into card indexes and suits?

Let's look at indexes first. Card indexes repeat in blocks of 13. These blocks correspond to the four suits; for example, 0, 13, 26, and 39 all correspond to aces; 1, 14, 27, and 40 all correspond to 2's; and so forth. The modulo 13 function expresses this correspondence nicely. Any card number can be expressed in modulo 13, which produces a number from 0 to 12. By adding 1, you get a number from 1 to 13 that corresponds to the 13 possible index names. This may be more easily seen with the following formulas:

$$Index\ number = Card\ number \text{ modulo } 13 + 1$$

$$Index\ name = F\$(Index\ number)$$

To determine a card's suit, we observe that card values 0-12 are hearts, 13-25 are diamonds, 26-38 are clubs, and 39-51 are spades. Dividing a card number by 13 and discarding the fractional portion of the quotient gives a number from 0 to 3. Adding 1 gives a number from 1 to 4, which corresponds to the four suits. The formulas to do this are

$$Suit\ number = \text{Integer part of } card\ number\ /\ 13 + 1$$

$$Suit\ name = S\$(Suit\ number)$$

—The Program

The first block of the program sets the screen color scheme, prints a title, and sets up certain values.

```
1 POKE 53281,1: REM SCREEN=WHITE
2 POKE 53280,1: REM BORDER=WHITE
3 NM$=CHR$(30): REM CHARACTERS=GREEN
4 PRINT NM$;: REM SET GREEN COLOR
10 PRINT CHR$(147);: REM CLEAR SCREEN
20 PRINT "BLACKJACK '84"
30 PRINT
40 INPUT "ENTER A RANDOM NUMBER ";X
50 X=RND(-ABS(X))
60 SC=100
70 AN=5
80 DS=17
```

NM$ is the code for green, the color that is used for most of the program's output. Lines 40 and 50 let you set the random number function (used later in the program) so the program won't shuffle the deck the same way every time you run it.

SC is the number of chips held by the player. Initially, this is a value of 100. AN is the automatic minimum bet, or ante, for each new hand. DS is the point at which the dealer stands. You may change any of these values to suit your preference.

Storing the First Round

The next program block creates the card lists and other arrays.

```
90 DIM D(52),CP(52),F$(13),S$(4),CL$(4),
    H(2,11),A(2),V(13),T(2),C(2),PN$(2)
100 PN$(1)="DEALER"
110 PN$(2)="PLAYER"
120 INPUT "ENTER YOUR NAME: ";PN$(2)
130 REM
140 FOR J=1 TO 13
150 READ F$(J)
160 NEXT J
170 DATA A,2,3,4,5,6,7,8,9,10,J,Q,K
180 FOR J=1 TO 4
190 READ SB,CB
195 S$(J)=CHR$(SB)
196 CL$(J)=CHR$(CB)
200 NEXT J
210 DATA 115,28,120,144,122,28,97,144
220 FOR J=1 TO 13
230 READ V(J)
240 NEXT J
250 DATA 11,2,3,4,5,6,7,8,9,10,10,10,10
```

In addition to the arrays discussed previously, we have CL$( ), H(,), A( ), T( ), and PN$( ).

CL$( ) stores the color code for each of the four suits. H(*player number, card number*) stores the contents of each hand. H(1,) refers to the dealer, and H(2,) refers to the player. For instance, H(1,2) is the dealer's second card and H(2,3) is the player's third card. The second element of H(,) is a card number from 0 to 51.

A(*player number*) stores the number of aces in each hand. T(*player number*) stores the total points in each hand. PN$(*player number*) stores the names of the dealer and player.

Lines 140-250 read in the card index names, suit names, color codes, and card values.

The following lines print an introduction to the first round:

```
260 PRINT
270 PRINT PN$(2) " STARTS WITH " SC " CHIPS."
280 PRINT "THE ANTE IS "AN
290 INPUT "PRESS RETURN TO START"; EN$
300 PRINT "HERE'S THE FIRST ROUND . . ."
```

Dealing a New Hand

Each time a new hand is dealt (except after a tie), the program executes the following block, which bets another ante:

```
310 IF SC>=AN THEN 350
320 PRINT "YOU CAN'T MAKE THE ANTE."
330 PRINT "GAME OVER."
340 END
350 PRINT
360 SC=SC-AN
370 PB=AN
380 PRINT PN$(2) " HAS" SC "CHIPS LEFT"
390 PRINT "AFTER MAKING THE ANTE."
400 PRINT
```

Line 310 checks to see whether the player has enough chips to make the ante. If not, the game ends. Lines 360 and 370 deduct the ante from the player's score SC and add it to the player's bet PB.

The next block starts a new hand by dealing two cards each to the dealer and player:

```
410 PRINT "NEW HAND"
420 SF=0
430 FOR CN=1 TO 52
440 CP(CN)=0
450 NEXT CN
460 T(1)=0
470 T(2)=0
480 C(1)=0
490 C(2)=0
500 FOR WH=1 TO 2
510 GOSUB 1490
520 GOSUB 1490
530 GOSUB 1730
540 GOSUB 1880
550 NEXT WH
```

SF is a control variable that determines whether the dealer's first card is dealt face down (SF = 0) or face up (SF = 1). During a game, the card is always face down; when the scores are revealed, it is face up.

Lines 430-450 empty the list of cards in use. Lines 460-490 set the hand totals and card counts equal to 0. The loop from 500 to 550 deals two cards to each player. (Throughout the program, the variable WH indicates whose hand is being hit, displayed, scored, and so forth — WH = 1 for the dealer and WH = 2 for the player.)

Bet, Hit, Stand, or Review

The player and dealer each have two cards now. The following block gives the player a chance to increase the bet, get another card, stand, or review the totals.

```
560 IF T(2)>21 THEN 1100
570 WH=2
580 PRINT
590 PRINT "BET, HIT, STAND, OR REVIEW CARDS?"
600 INPUT "ENTER B/H/S/R: ";YN$
610 IF YN$<>"B" THEN 680
615 B=0
620 PRINT PN$(2) " HAS" SC "CHIPS."
630 INPUT "BET HOW MUCH NOW? ";B
640 IF B>SC OR B<0 THEN 620
650 SC=SC-B
660 PB=PB+B
670 GOTO 580
680 IF YN$<>"H" THEN 750
690 GOSUB 1490
700 J=C(2)
710 GOSUB 1960
720 PRINT CN$
730 GOSUB 1730
740 GOTO 560
750 IF YN$<>"R" THEN 840
760 PRINT CHR$(147);
770 PRINT PN$(2)" HAS" SC "CHIPS."
780 PRINT "THE BET IS"PB
790 FOR WH=1 TO 2
800 GOSUB 1730
810 GOSUB 1880
820 NEXT WH
830 GOTO 570
840 IF YN$="S" THEN 860
850 GOTO 570
```

Line 560 checks the player's total, T(2). If T(2) is greater than 21, the player is busted. Each time the player receives a new card, the program returns to this line, rechecking the total for a bust.

If the player isn't busted yet, he or she gets to choose an option: bet, hit, review cards, or stand.

Lines 620-670 handle the betting option. Lines 690-740 handle the hitting option. The subroutine called in line 690 draws a card from the deck. The subroutine called in line 710 derives the index and suit of the card just drawn. Line 720 names the card just drawn.

Lines 760-830 review all the relevant information about the current round.

Dealer's Turn

When the player stands, the program gives the dealer a turn:

```
860 WH=1
870 PRINT
880 PRINT "DEALER'S TURN: ";
890 IF T(1)<DS THEN 920
900 PRINT "DEALER STANDS"
910 GOTO 990
920 PRINT "HIT"
930 GOSUB 1490
940 J=C(1)
950 GOSUB 1960
960 PRINT CN$
970 GOSUB 1730
980 GOTO 890
```

Line 890 determines whether the dealer has reached the number 17 (DS), at which he must stand. If the dealer's score is less than DS, he must draw a card (line 930). The subroutine called in line 950 identifies the card just drawn, and line 960 prints that information. The dealer continues drawing cards until the total reaches or exceeds DS.

The next block displays and totals up the player's and dealer's hands:

```
990 PRINT
1000 INPUT "PRESS RETURN TO REVEAL THE
     WINNER ";EN$
1010 SF=1
1020 WH=2
1030 PRINT CHR$(147);
1040 GOSUB 1880
1050 PRINT PN$(2)"'S SCORE IS " T(2)
1060 WH=1
1070 PRINT
1080 GOSUB 1880
1090 PRINT "DEALER'S SCORE IS " T(1)
```

In line 1010, SF is set to 1 so that the display-hand subroutine will show the dealer's first card (hidden until now).

Finding the Winner

The program now compares the two hands and determines the winner.

```
1100 PRINT
1110 IF T(2)<=21 THEN 1140
1120 PRINT PN$(2);" BUSTS. DEALER WINS ";PB
1130 GOTO 1380
1140 IF T(1)<>T(2) THEN 1170
1150 PRINT "SCORES ARE THE SAME."
1160 GOTO 1310
1170 IF T(2)<>21 THEN 1210
1180 PRINT "BLACKJACK! ";PN$(2);" WINS ";PB*3
1190 SC=SC + PB*3
1200 GOTO 1380
1210 IF T(1)<=21 THEN 1250
1220 PRINT "DEALER BUSTS. ";PN$(2);" WINS "; PB*2
1230 SC=SC + PB*2
1240 GOTO 1380
1250 IF T(2)<=T(1) THEN 1290
1260 PRINT PN$(2);" WINS "; PB*2
1270 SC=SC + PB*2
1280 GOTO 1380
1290 PRINT "DEALER WINS ";PB
1300 GOTO 1380
```

If the player is busted, the player loses the hand, regardless of the dealer's score (lines 1110 and 1120). If the player's and dealer's scores are the same, the program skips to a tie-resolution routine (lines 1150 and 1160). If the player has 21, he wins triple the amount bet (lines 1180 and 1190)—even if the dealer has busted.

If the player's score is higher than the dealer's, the player wins double the amount bet (lines 1260 and 1270).

Finally, if the dealer's score is higher than the player's, the player loses the amount bet (line 1290).

Resolving Ties

Here's the routine to resolve ties:

```
1310 IF C(2)=C(1) THEN 1360
1320 HC=-1*(C(1)>C(2)) + -2*(C(2)>C(1))
1330 T(HC)=0
1340 PRINT "LOWEST CARD COUNT WINS."
1350 GOTO 1170
1360 PRINT "STANDOFF."
1370 GOTO 430
```

This routine awards the win to the holder (player or dealer) of the fewest cards. Line 1320 calculates HC, the number of the player with the most cards. That player's total, T(HC), is set equal to 0, and the totals are compared again, forcing player HC to lose (lines 1330-1350).

If both players have the same card count, the bet remains the same and a new hand is dealt.

Starting a New Round

The last block of the main program displays the player's winnings and an offer to play again.

```
1380 PRINT
1390 PRINT PN$(2);" NOW HAS ";SC;"CHIPS."
1400 INPUT "PLAY ANOTHER HAND? (Y/N) ";YN$
1410 IF YN$="N" THEN END
1420 IF YN$<>"Y" THEN 1380
1430 FOR WH=1 TO 2
1440 FOR J=1 TO C(WH)
1450 CP(H(WH,J)+1)=0
1460 NEXT J,WH
1470 PRINT CHR$(147);
1480 GOTO 310
```

If the player agrees to play again, lines 1430-1460 remove each player's cards, one at a time, from the cards-in-use list. The program then jumps back to the new-hand routine.

Subroutines

The following lines draw a card from the deck and add it to player WH's hand:

```
1490 IF CR>0 THEN 1670
1500 PRINT
1510 REM
1520 PRINT "SHUFFLING THE CARDS...STANDBY"
1530 REM
1540 PRINT
1550 CA=52-C(1)-C(2)
1560 FOR J=1 TO CA
1570 D(J)=-1
1580 NEXT J
1590 FOR J=1 TO 52
1600 IF CP(J)=-1 THEN 1640
1610 CD=INT(RND(1)*CA)+1
```

```
1620 IF D(CD)<>-1 THEN 1610
1630 D(CD)=J-1
1640 NEXT J
1650 PRINT CHR$(147);
1660 CR=CA
1670 CV=D(CR)
1680 CP(CV+1)=-1
1690 CR=CR-1
1700 C(WH)=C(WH)+1
1710 H(WH,C(WH))=CV
1720 RETURN
```

Line 1490 determines whether any cards remain in the deck. When CR=0, all the cards have been dealt. In this case, the entire deck must be shuffled (all but the cards that are currently in use).

Lines 1550-1660 shuffle the cards that are available. CA is the number of cards available.

Lines 1670-1710 pull a card from the top of the deck and put it into the hand of player WH (lines 1700 and 1710).

Line 1720 returns to the main program.

The subroutine to total a player's hand is as follows:

```
1730 TT=0
1740 A(WH)=0
1750 FOR J=1 TO C(WH)
1760 CV=H(WH,J)
1770 VL=CV-INT(CV/13)*13+1
1780 IF VL=1 THEN A(WH)=A(WH)+1
1790 TT=TT+V(VL)
1800 NEXT J
1810 PRINT
1820 IF TT<=21 OR A(WH)<=0 THEN 1860
1830 TT=TT-10
1840 A(WH)=A(WH)-1
1850 GOTO 1820
1860 T(WH)=TT
1870 RETURN
```

Line 1730 sets a temporary subtotal TT to 0. Line 1740 sets the ace counter A(WH) to 0. The ace counter is needed because aces can be evaluated as either 11 or 1.

Lines 1750-1800 add up the values of all the cards in the hand of player WH. Line 1770 calculates the card value using CV modulo 13.

When the program reaches line 1820, it has a temporary total. If the total indicates a bust, it may still be possible to save the hand by evaluating the aces as 1's instead of as 11's. Line 1850 performs this evaluation. If there are any aces in such a hand, the program subtracts 10, giving the ace its optional value of 1 instead of the previously assigned value of 11.

The following lines constitute the display-hand subroutine:

```
1880 PRINT PN$(WH);"'S HAND:"
1900 FOR J=1 TO C(WH)
1910 GOSUB 1960
1920 PRINT CN$;
1930 NEXT J
1940 PRINT
1950 RETURN
```

The variable J counts from 1 to the number of cards in the hand; for each card, the subroutine called in line 1910 gets the card's index and suit name. Line 1920 displays this information.

The last subroutine in the program derives a card's index and suit based on its card number.

```
1960 CV=H(WH,J)
1970 VL=CV-INT(CV/13)*13+1
1980 SU=INT(CV/13)+1
1990 IF WH<>1 OR J<>1 OR SF<>0 THEN 2020
2000 CN$="[??] "
2010 GOTO 2030
2020 CN$="["+CL$(SU)+F$(VL)+S$(SU)+NM$+"] "
2030 RETURN
```

Line 1970 calculates CV modulo 13 to get the card's value. Line 1980 uses integer division to get the card's suit.

When the dealer's first card is being shown (WH = 1, J = 1), the variable SF in line 1990 determines whether the value will be revealed or masked with a string of question marks.

CN$ in line 2020 is a string composed of the card's index and its suit. CL$(SU) sets the suit color, and NM$ restores the "normal" green color. Line 2030 ends the subroutine with CN$ containing the card identification.

—Testing the Program

After typing in the program and removing all obvious typographical errors, test the card-shuffling subroutine by adding these lines:

```
281 WH=2
282 FOR QQ=1 TO 52
```

```
283 GOSUB 1490
284 PRINT CV+1; TAB(10);
285 J=1: GOSUB 1970: PRINT CN$
286 CP(CV+1)=0
287 C(WH)=0
288 NEXT QQ
289 STOP
```

Run the program. It should print the contents of a shuffled deck, showing card numbers and the corresponding indexes and suits. The

Table 7-1. Typical Contents of a Shuffled Deck

Card No.	Index and Suit	Card No.	Index and Suit
11	QUEEN OF HEARTS	13	ACE OF CLUBS
28	3 OF DIAMONDS	36	JACK OF DIAMONDS
15	3 OF CLUBS	50	QUEEN OF SPADES
38	KING OF DIAMONDS	49	JACK OF SPADES
8	9 OF HEARTS	14	2 OF CLUBS
7	8 OF HEARTS	42	4 OF SPADES
4	5 OF HEARTS	46	8 OF SPADES
40	2 OF SPADES	39	ACE OF SPADES
48	10 OF SPADES	44	6 OF SPADES
29	4 OF DIAMONDS	43	5 OF SPADES
22	10 OF CLUBS	2	3 OF HEARTS
37	QUEEN OF DIAMONDS	1	2 OF HEARTS
31	6 OF DIAMONDS	45	7 OF SPADES
0	ACE OF HEARTS	35	10 OF DIAMONDS
16	4 OF CLUBS	26	ACE OF DIAMONDS
30	5 OF DIAMONDS	47	9 OF SPADES
17	5 OF CLUBS	23	JACK OF CLUBS
25	KING OF CLUBS	51	KING OF SPADES
27	2 OF DIAMONDS	18	6 OF CLUBS
33	8 OF DIAMONDS	34	9 OF DIAMONDS
41	3 OF SPADES	21	9 OF CLUBS
24	QUEEN OF CLUBS	32	7 OF DIAMONDS
10	JACK OF HEARTS	6	7 OF HEARTS
3	4 OF HEARTS	5	6 OF HEARTS
9	10 OF HEARTS	19	7 OF CLUBS
20	8 OF CLUBS	12	KING OF HEARTS

results should be similar to the listing in Table 7-1, but the card sequence will be different.

Card number 0 corresponds to the ace of hearts, 1 to the 2 of hearts, and so forth, up to card 51, which corresponds to the king of spades.

If your listing contains all 52 cards and the pairings correspond to those in Table 7-1, you can be reasonably sure the program is playing with a full deck.

Delete lines 281-289 and start enjoying Blackjack '84!

Chapter 8

Billiard Practice

The Billiard Practice program turns your C-64's display into an electronic billiard table. This "table" is primarily for practicing and experimenting with different kinds of angle shots, but it can also be used in simplified games of billiards.

The table has no pockets and only two balls—a cue ball and an object ball. At the beginning of each round, the balls are spotted (positioned) at randomly chosen locations. You can shoot the cue ball at the object ball directly or bounce it off one or more of the rails.

You specify the direction of your shot with degrees. Imagine the degrees on the face of a clock: 0 degrees is at 3 o'clock; 90 degrees at 6 o'clock; 180 degrees at 9 o'clock; 270 degrees at 12 o'clock, and 360 degrees at 3 o'clock. Figure 8-1 shows several cuing directions and the corresponding angles.

The Billiard Practice program lets you check the angle you have selected before you actually shoot the ball. It extends a line from the ball through the angle you specify. The line stops at the rail or the object ball, whichever comes first.

When you shoot the ball, it travels in a straight line until it hits the object ball or strikes a rail. After striking a rail, the ball bounces in a

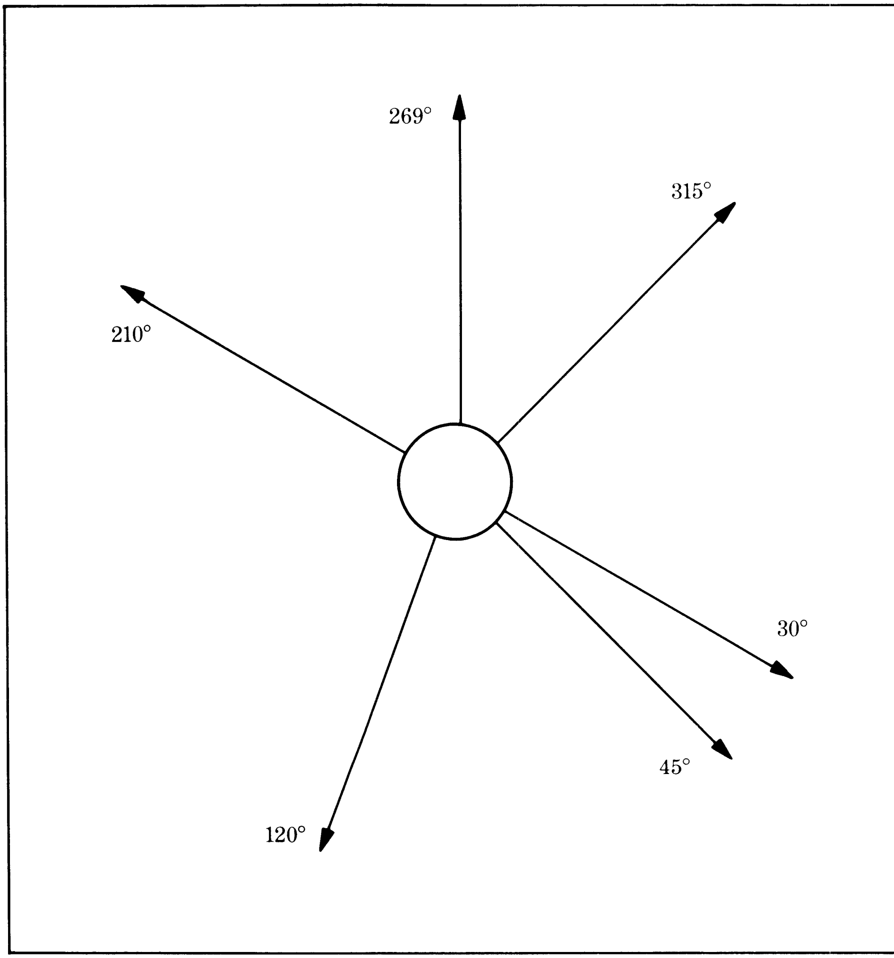


Figure 8-1. Degrees are used to specify cuing directions

different direction. That direction is determined by the law of physics which states that the angle of deflection equals the angle of inflection (see Figure 8-2).

Unlike real billiards, the electronic version is not affected by friction or gravity. The cue ball rolls at a constant speed until it hits the object ball, which stops it immediately.

It is possible to shoot the cue ball in such a direction that it will never hit the object ball. To prevent the ball from rolling indefinitely, you can set the maximum number of bounces allowed; the ball will

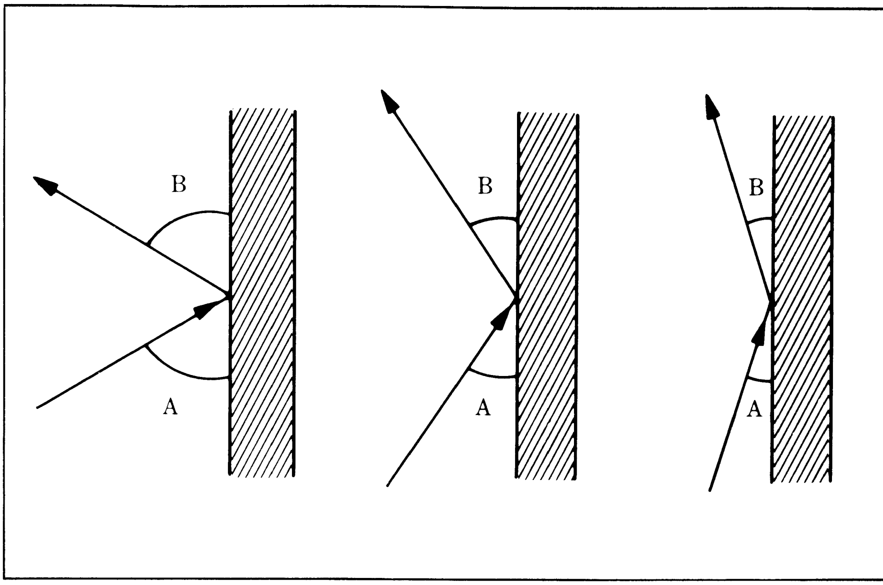


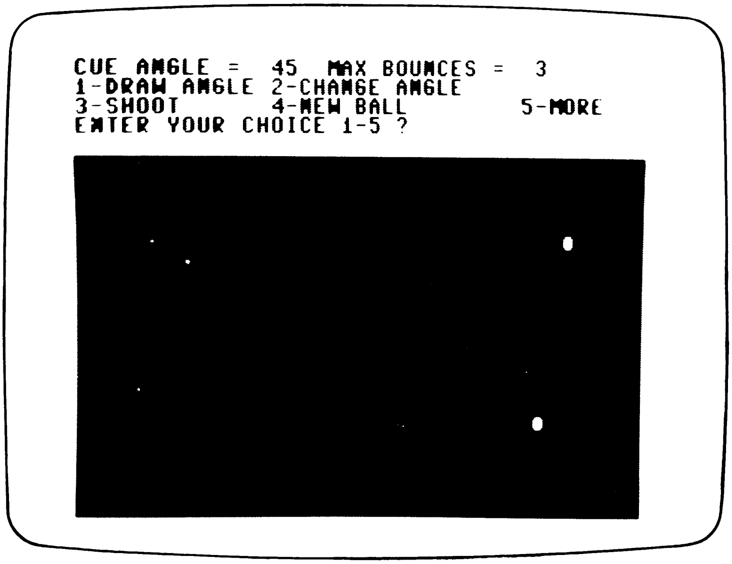
Figure 8-2. The angle of deflection (B) always equals the angle of inflection (A)

always stop after the specified number. You can also follow the course of the ball over an unlimited number of bounces. To do this, set 0 as the maximum number of bounces. In that case, the cue ball will stop only if it hits the object ball. Figure 8-3 shows the program in use.

The Program

The first block sets up the constants used to control the screen graphics:

```
10 CS$=CHR$(147)
12 HO$=CHR$(19)
16 RV$=CHR$(18): NM$=CHR$(146)
18 RC$=CHR$(17): LC=25: GOSUB 1590: VM$=SO$
20 RC$=CHR$(29): LC=40: GOSUB 1590: HM$=SO$
22 SM=256*PEEK(648)
24 CM=55296
26 BO=1: REM WHITE BORDER
28 FE=5: REM GREEN FELT
30 OC=7: REM YELLOW OBJECT BALL
32 CC=2: REM RED CUE BALL
34 DT=87: REM PATH INDICATOR
36 BL=81: REM BALL
```



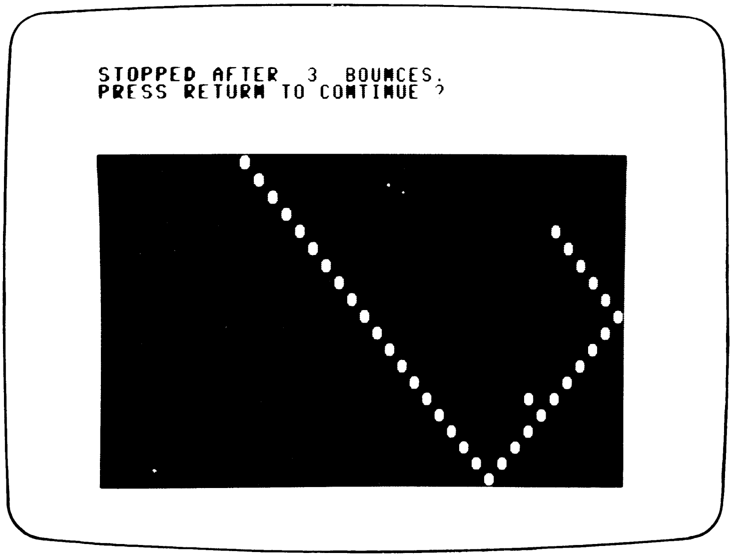


Figure 8-3. Sample screens of Billiard Practice

```
38 BO$=CHR$(5): REM PRINT BORDER COLOR
40 FE$=CHR$(30): REM PRINT FELT COLOR
42 RC$=CHR$(32): LC=40: GOSUB 1590: BL$=SO$
```

The subroutine called in lines 18, 20, and 42 returns a string of characters. As a result, VM$ contains 25 "cursor-down" codes, HM$ contains 40 cursor-right codes, and SO$ contains 40 blank spaces. HM$ and VM$ are used to control the cursor position, and SO$ is used to erase a line on the display.

SM is the start of the C-64 screen memory; storing a code 0-255 in screen memory puts a character on the screen. CM is the start of the C64 color memory; storing a color code 0-15 in color memory changes the color of the character at the corresponding location on the screen.

Setting Up the Tone Generators

The next block sets up the C-64's tone generators:

```
44 FOR R=54272 TO 54296: POKE R,0: NEXT R
46 DIM V(8)
48 FOR K=1 TO 7: V(K)=54271+K: NEXT K
50 V(8)=54296
52 READ FL,FH,PL,PH,WF,AD,SR,VL
54 DATA 1,2,3,4,5,6,7,8
56 POKE V(FL),100: POKE V(FH),30
58 POKE V(PL),0: POKE V(PH),10
60 POKE V(WF),65
62 POKE V(AD),0
64 POKE V(SR),0
66 POKE V(VL),15
```

The program uses tone generator 1, which is controlled by memory locations 54272 to 54278 and 54296. The array V( ) stores these memory addresses. The variables listed in line 52 store the eight indexes used to specify which address is needed. For instance, V(WF), where WF=5, is the address of the waveform control register. To set the waveform to a value WT, use POKE V(WF),WT, which is equivalent to POKE 54276, WT.

For a fuller explanation of the use of C-64 tone generators, refer to Chapter 7 of *Your Commodore 64* by John Heilborn and Ran Talbot (Osborne/McGraw-Hill, 1983).

Storing Math Constants and the Table Layout

The program next sets up several numeric constants and other control variables.

```
68 PI=4*ATN(1)
70 CF=2*PI/360
72 READ W0,WZ,L0,LZ
74 DATA 0,39,5,24
76 LX=LZ-L0+1
78 WX=WZ-W0+1
80 BD=1
100 CX=BD*BD
110 BC=BD*2
120 XX=WX-2*BC
130 YY=LX-2*BC
150 MB=3
```

PI is the ratio of a circle's circumference to its diameter. CF is the conversion factor for degrees to radians; it is needed because Commodore BASIC's trigonometric functions require that angles be measured in radians rather than degrees.

W0, WZ, L0, and LZ are the outlines of the table. BD is the ball's diameter measured in pixels (picture elements). CX is the diameter squared—a useful value later on when figuring whether the cue ball has hit the object ball. BC is the minimum distance required between the ball and the rails when the balls are initially spotted. XX and YY define the size of the area in which the balls may be spotted at the beginning of a round. MB in line 150 is the maximum number of bounces before the ball stops; the program lets you change this value before shooting the ball.

Figure 8-4 gives a pictorial representation of many of these variables.

Initializing the Screen

The next lines set the screen color, clear a five-line text window, and prompt the user to enter a random number:

```
160 POKE 53280,BO
170 POKE 53281,FE
180 PRINT CS$;BO$
190 GOSUB 1460: REM CLEAR TEXT WINDOW
200 INPUT "ENTER A RANDOM NUMBER ";RX
210 RX=RND(-ABS(RX))
```

The following block of lines randomly selects locations for the cue ball and object ball:

```
290 X=INT(RND(1)*XX)+BC
300 Y=INT(RND(1)*YY)+BC
310 X1=INT(RND(1)*XX)+BC
320 Y1=INT(RND(1)*YY)+BC
```

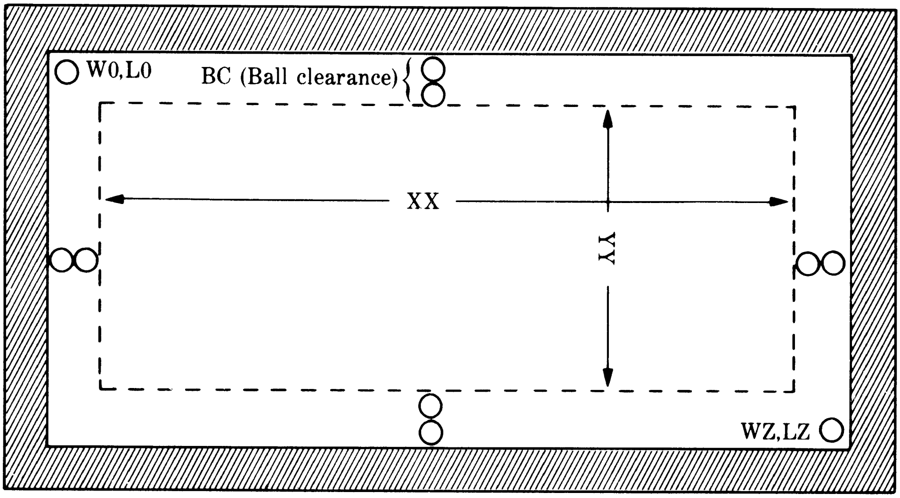


Figure 8-4. Billiard table mapped onto the X-Y coordinate system of a high-resolution graphics screen

```
330 IF (X1-X)*(X1-X)+(Y1-Y)*(Y1-Y)<BC*BC THEN 290
340 AG=45
```

The coordinate pairs X,Y and X1,Y1 specify the location of the cue ball and object ball respectively. Lines 290-320 ensure that the initial position will always be at least two ball diameters away from every rail. Look back to line 110, *ball clearance* (BC) = 2 × *ball diameter* (BD). Line 330 ensures that balls are separated from each other by at least two ball diameters. Line 340 sets the initial cuing angle to 45 degrees.

Spotting the Balls

The following lines put the cue and object balls on the table at their random locations X,Y and X1,Y1:

```
450 PRINT CS$
460 HC=CC: OB=BL
465 ZX=X
470 ZY=Y
475 GOSUB 1550
480 HC=OC: OB=BL
485 ZX=X1
490 ZY=Y1
495 GOSUB 1550
```

The cue ball is red and the object ball is yellow.

Menus

Using the text window at the top of the display, the program presents you with a menu of options.

```
510 GOSUB 1460
515 CP=0
520 GOSUB 1440
530 PRINT RV$; "CUE ANGLE = ";AG;"
    MAX BOUNCES = ";MB
535 CP=1:GOSUB 1440
540 PRINT RV$;"1-DRAW ANGLE 2-CHANGE ANGLE"
545 CP=2:GOSUB 1440
550 PRINT RV$;"3-SHOOT       4-NEW BALL
    5-MORE"
555 CP=3:GOSUB 1440
560 PRINT RV$;"ENTER YOUR CHOICE 1-5 "
563 PRINT RV$;
565 INPUT MO
570 IF MO<1 OR MO>5 THEN 510
580 ON MO GOTO 700,800,860,210,590
590 GOSUB 1460
600 PRINT RV$;"6-CHANGE MAX BOUNCE COUNT
    7-QUIT"
620 PRINT RV$;
625 INPUT MO
630 IF MO<6 OR MO>7 THEN 510
640 IF MO=7 THEN 670
```

Line 530 prints the current angle setting and maximum bounce limit.

Because of the limited space available for text, the menu is divided into two pages. The first page gives you five options: 1-DRAW ANGLE, 2-CHANGE ANGLE, 3-SHOOT, 4-NEW BALL POSITION, and 5-MORE (the next menu page). The second menu page gives you two additional options: 6-CHANGE THE MAXIMUM BOUNCE COUNT and 7-QUIT.

The following block of lines performs options 6 and 7:

```
650 CP=2:GOSUB 1440
652 PRINT RV$;
655 INPUT "NEW MAX BOUNCE COUNT (0=NO LIMIT) ";MB$
658 MB=VAL(MB$)
660 GOTO 510
670 PRINT CS$;
680 END
```

Checking the Cuing Angle

If you select option 1 (DRAW ANGLE), the following lines draw a dotted path extending from the cue ball at angle AG:

```
700 GOSUB 1230
710 PX=X+DX
720 PY=Y+DY
730 HC=CC: OB=DT
740 GOSUB 1340
750 PX=X+DX
760 PY=Y+DY
770 HC=FE
780 GOSUB 1340
790 GOTO 510
```

The subroutine called in line 700 calculates DX and DY, the X and Y increments that will produce a line from the cue ball at the specified angle AG. Figure 8-5 illustrates these values.

The coordinate pair PX,PY identifies the first point of the path to be drawn. Line 730 sets the color and shape of the dots. The subroutine called in line 740 draws a dotted line starting at PX,PY and ending at the object ball or a rail, whichever it hits first.

Lines 750-780 repeat the process exactly, except that now the background color number is used, which erases the dotted line. Line 790 jumps back to the menu.

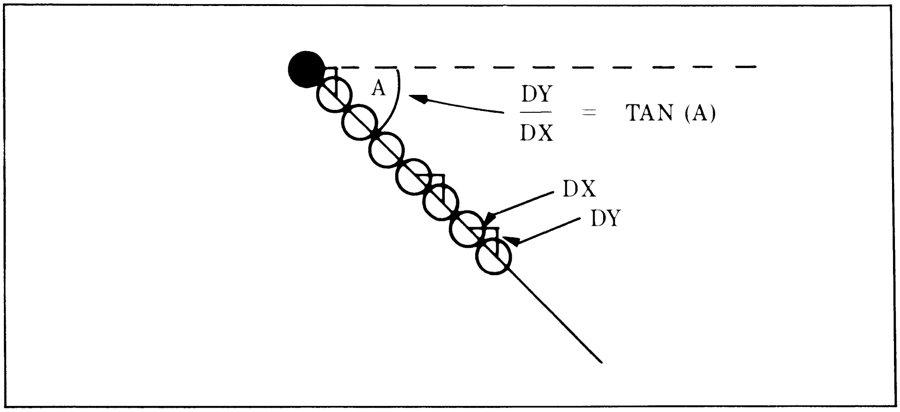


Figure 8-5. Calculating the X and Y increments that produce a desired direction

Changing the Cuing Angle

The following lines let you change the cuing angle:

```
800 CP=4:GOSUB 1440
802 PRINT RV$;
804 INPUT "ENTER NEW ANGLE 0.1-359.9 ";AG$
806 AG=INT(VAL(AG$)*10+.5)/10
810 IF AG<.1 OR AG>359.9 THEN 800
820 IF AG/90<>INT(AG/90) THEN 510
830 AG=AG+.1: REM TO AVOID VERTICAL AND HORIZONTAL
    ANGLES
840 GOTO 510
```

Lines 820-830 make sure that you do not select angles that are perpendicular to any of the rails, that is, angles of 90, 180, 270, or 360 degrees. Eliminating these angles from consideration simplifies the program's logic. Such angles aren't useful in bank shots anyway, since the ball always bounces straight back, recrossing its original location. If you are attempting a direct shot at the object ball and you need one of these angles, simply add 0.001 to the desired angle. The program will accept almost perpendicular angles, and the effect will usually be identical to that of truly perpendicular angles.

If the angle you enter is within limits, line 820 jumps back to the menu.

Shooting the Ball

The next block of lines shoots the ball:

```
860 NB=0
870 GOSUB 1230
880 HC=CC: OB=BL
890 PX=X
900 PY=Y
910 POKE V(AD),6
912 POKE V(FH),255
914 POKE V(VL),15
916 WT=129
918 GOSUB 1660
920 ZX=PX
922 ZY=PY
924 GOSUB 1550
930 QX=INT(PX)
940 QY=INT(PY)
950 IF QX<0 OR QX>=WX OR QY<0 OR QY>=LX THEN 1040
960 HC=FE: GOSUB 1550
```

```
970 HC=CC
972 ZX=QX
974 ZY=QY
976 GOSUB 1550
980 IF (X1-QX)*(X1-QX)+(Y1-QY)*(Y1-QY)<=CX
    THEN 1170
990 BX=PX
1000 BY=PY
1010 PX=PX+DX
1020 PY=PY+DY
1030 GOTO 930
```

NB keeps track of the number of bounces that have occurred. The subroutine called in line 870 calculates the X and Y increments, as shown in Figure 8-5.

Lines 910-918 produce a click to simulate the striking of the ball with the invisible cue stick. Lines 920-924 redraw the cue ball at its current location.

Line 950 determines whether the ball has hit one of the rails. If it has, the program jumps to another block of lines presented in the next section. If the ball has not hit a rail, lines 960 and 970 move the ball to its next location. Line 960 erases the old ball, and line 970 draws a new one in the new location.

To leave a trail of the cue ball's path on the screen, delete line 960. This will produce some interesting patterns on the screen, especially when the maximum bounce count is unlimited.

Line 980 determines whether the cue ball has hit the object ball. When the distance between the two balls is less than or equal to the ball diameter, the balls have hit. In that case, the program jumps to a block of lines shown next.

Hitting a Rail

When the ball hits a rail, the following lines take over:

```
1040 POKE V(AD),3
1042 POKE V(FH),8
1044 POKE V(VL),6
1046 WT=65
1048 GOSUB 1660
1050 NB=NB+1
1060 IF NB<>MB THEN 1120
1070 GOSUB 1460
1080 PRINT RV$;"STOPPED AFTER ";NB;" BOUNCES."
1090 GOTO 1200
```

```
1120 IF QX<0 OR QX>=WX THEN DX=-DX
1130 IF QY<0 OR QY>=LX THEN DY=-DY
1140 PX=BX+DX
1150 PY=BY+DY
1160 GOTO 930
```

Lines 1040-1048 make a short beep to simulate the bounce. Line 1050 increments the bounce counter, and line 1060 checks to see whether the number of bounces equals the maximum bounce limit. If it does, the ball stops and lines 1070-1090 print a message in the text window.

If the bounce count is not equal to the maximum limit, lines 1120 and 1130 make the necessary changes in DX and DY to effect the change in direction. If the ball has hit the left or right rail (QX < 0 or QX >= WX), the sign of DX is reversed. If the ball has hit the top or bottom rail (QY < 0 or QY >= LX), the sign of DY is reversed.

Lines 1140 and 1150 compute the next position PX,PY in the path of the cue ball. Line 1160 returns to a previous line in the shooting routine.

When the cue ball hits the object ball, the following lines produce a click and print a message in the text window:

```
1170 POKE V(AD),6
1172 POKE V(FH),240
1174 POKE V(VL),10
1176 WT=129
1178 GOSUB 1660
1180 GOSUB 1460
1190 PRINT RV$;"CONTACT AFTER ";NB;" BOUNCES."
1200 CP=1:GOSUB 1440
1205 PRINT RV$;
1210 INPUT "PRESS RETURN TO CONTINUE ";EN$
1215 HC=FE:GOSUB 1550
1220 GOTO 450
```

Subroutines

The first subroutine calculates DX and DY, as illustrated in Figure 8-5. The subroutine is used by the draw-angle routine and the shoot-ball routine.

```
1230 A=AG*CF
1240 TA=ABS(TAN(A))
1250 SX=SGN(COS(A))*BD
1260 SY=SGN(SIN(A))*BD
1270 IF TA<1 THEN 1310
1280 DY=SY
1290 DX=1/TA*SX
```

```
1300 RETURN
1310 DX=SX
1320 DY=TA*SY
1330 RETURN
```

Line 1230 converts the angle from degrees to radians. Line 1240 saves the tangent of the angle; this value tells us the ratio that must obtain between DX and DY in order to produce the desired angle.

Lines 1280 and 1290 calculate DX and DY for angles closer to the vertical direction than to the horizontal; lines 1310 and 1320 calculate DX and DY for angles closer to the horizontal direction than to the vertical.

The next block of lines is used by the draw-angle routine to extend a line from the cue ball to a rail or to the object ball:

```
1340 QX=INT(PX)
1350 QY=INT(PY)
1360 IF QX<0 OR QX>=WX OR QY<0 OR
     QY>=LX THEN RETURN
1370 IF (X1-QX)*(X1-QX)+(Y1-QY)*(Y1-QY)<=CX
     THEN RETURN
1380 ZX=QX
1384 ZY=QY
1386 GOSUB 1550
1390 PX=PX+DX
1400 PY=PY+DY
1410 GOTO 1340
```

On entry to this subroutine, QX,QY identifies the next point in the path. Before the point is plotted on the screen, line 1360 checks for impending collisions with each of the rails, and line 1370 checks for impending collisions with the object ball. In the case of an impending collision, the subroutine returns control to the main program.

On the other hand, if the ball is not about to be stopped by a rail or another ball, lines 1380-1400 plot the next point and calculate new coordinates PX,PY.

The next lines contain two subroutines that facilitate use of the text window:

```
1439 GOTO 1439
1440 PRINT HO$;LEFT$(VM$,CP);
1450 RETURN
1460 PRINT HO$;
1470 FOR LL=1 TO 5
1480 PRINT RV$;
1490 PRINT BL$;
```

```
1500 NEXT LL
1510 PRINT HO$;
1540 RETURN
```

The subroutine at 1440-1450 positions the cursor to the first column of line number CP. The subroutine at 1460-1540 erases the five-line text window at the top of the screen.

The following subroutine plots an object on the screen and sets its color:

```
1550 RL=ZX+W0+40*(ZY+L0)
1560 POKE CM+RL,HC
1570 POKE SM+RL,OB
1580 RETURN
```

HC is the color code for the object, and OB is the shape code. RL is the offset needed to indicate the intended position of the object within screen memory or color memory.

Here's the subroutine (referred to at the beginning of the program listing) that creates a repeating string of characters:

```
1590 SO$=""
1600 FOR K=1 TO LC
1610 SO$=SO$+RC$
1620 NEXT K
1630 RETURN
```

Upon return from this subroutine, SO$ contains a string of character RC$. The number LC is its length.

Finally, we have a subroutine to produce a sound:

```
1660 POKE V(WF),0
1670 POKE V(WF),WT
1680 RETURN
```

Line 1660 turns the sound generator off, and line 1670 starts it again using the waveform indicated by WT. The program uses a white-noise waveform for the clicks and a pure tone for the bounces. The program does not have to turn off the sound because it has previously set up a high decay rate; the sound tapers off by itself.

—Testing and Using the Program

When testing the program, omit line 960. That way, the cue ball will leave a trail showing where it's been.

When you run the program, your screen should resemble those shown in Figure 8-3. Try all of the menu options to verify that each of them works. Set the maximum bounce limit (MB) in line 250 to 10 or more, and angle the cue ball so that it won't immediately hit the object ball.

Remember that the program will not accept angles that are exact multiples of 90 degrees. It will add 0.1 to any such angles you enter. For most shots, the results will be the same as if you used the exact angle.

You may notice that the ball occasionally bounces away from a rail before it actually comes in contact with the rail. This happens when the next available position on a path doesn't allow enough room for the ball to be drawn without biting into the rail. Don't worry; the ball's subsequent positions are calculated correctly (even though the ball couldn't be drawn at the point of contact with the rail).

Another peculiarity sometimes arises when the cue ball hits the object ball. The cue ball may actually merge with the object ball on the screen. Of course, real billiard balls don't behave this way—they bounce away from each other. However, the program's simulation of billiards ends at the instant of contact, and the merging of the balls is just an interesting aftereffect. Ambitious readers may wish to enhance the program by allowing the balls to bounce apart realistically. Without gravity and friction to slow them, the collisions could go on forever.

—Suggested Games

One of the simplest games for one or two players is *Call the Shot*. Each player starts with new ball positions (menu option 4). Before shooting, the player specifies which rails the ball will bounce off of en route to the object ball. The player may check the angle using option 1 before shooting the 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 of bounces that will be used.

Another game is *Circles*. The goal is to encircle the object ball in the path of the cue ball without hitting it. This game requires that you delete line 960 and set the maximum number of bounces to four or five.

Finally, players may take turns at *One-upmanship*. Players start at level 0, meaning that they must hit the ball without using any bounces. Each player starts a turn with new ball positions. The players try to hit the object ball using the number of bounces corresponding to each level. If a player succeeds in hitting the object ball, he advances to the next level (the number of bounces required is increased by 1) and continues with new ball positions. A player continues shooting until he misses, at which time the turn passes to the other player.

By prior agreement, players may or may not be allowed to use option 1 to check their angles before shooting.

Chapter 9

Tic-Tac-Toe

Although the rules and strategies of tic-tac-toe are simple, setting up your computer to play well is no mean task. In this chapter, your computer plays the game to a win or a draw every time. Compared to a good human player, the tic-tac-toe program's only weakness is its occasional passivity: settling for a draw when a victory is possible.

In addition to making your computer a good tic-tac-toe player, this program exemplifies three techniques that are just as applicable to more complex games like checkers and chess:

- Prepared opening moves.
- Lookahead—checking the consequences of a proposed move by looking ahead to subsequent moves.
- Heuristics—selecting moves based on principles of good strategy.

Playing Tic-Tac-Toe

Tic-tac-toe is played on a 3×3 grid. Two players take turns marking cells on the grid. The starting player (player X) marks with an X and the second player (player O) marks with an O.

The first player to place three marks (X's or O's) in a row, column, or

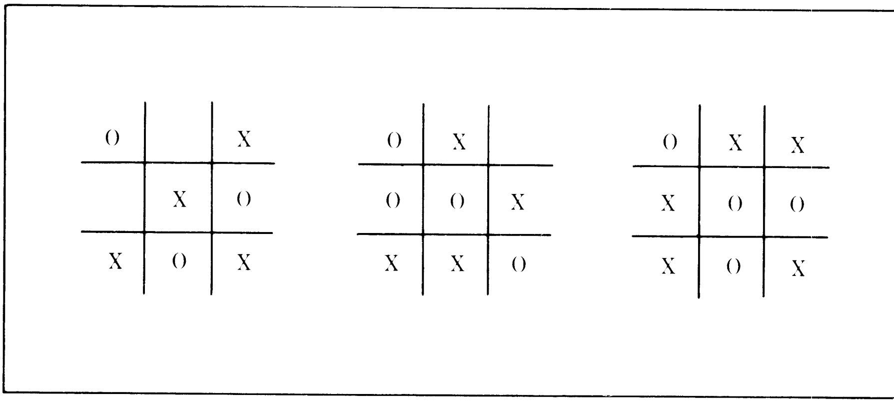


Figure 9-1. A win for player X, a win for player O, and a tie

diagonal wins. If all the cells are filled without either player winning, the game is a tie. (See Figure 9-1.) Before each subsequent game, players reverse their playing order, so that the second player becomes the starting player, and vice versa.

The simplest strategy for the game involves three steps:

1. If you can win on your next turn, do so.
2. Otherwise, if your opponent can win on his next turn, block him.
3. If neither condition is true, take any cell you can.

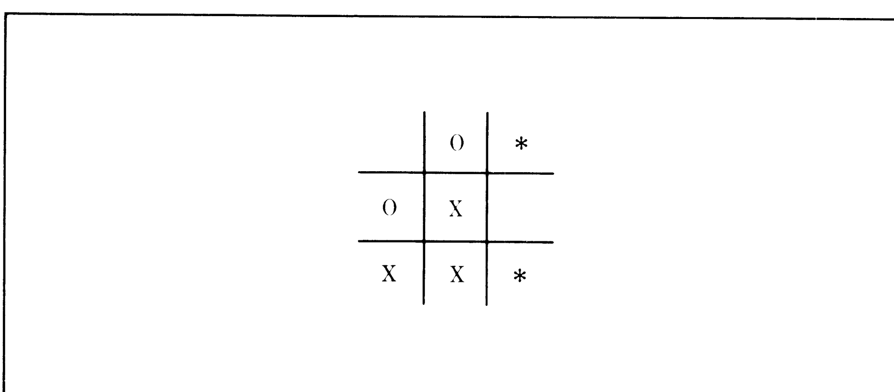


Figure 9-2. Player O is trapped; player X has two winning moves, indicated by asterisks

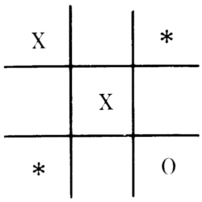


Figure 9-3. Player O can foil a trap by taking either safe cell, indicated by an asterisk.

It doesn't take a human player long to come up with some improvements or refinements for 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 9-2.) Your opponent will only be able to block one trap so you'll still have one winning opportunity.

Conversely, to avoid defeat at tic-tac-toe, you can prevent your opponent from setting such a trap. (See Figure 9-3.)

Preventing traps is not always easy. In some cases, you must look two turns ahead to spot a potential trap. Furthermore, player O's very first mark can set up a possible loss. Figure 9-4 shows the seven configurations that player O must avoid on his first turn.

—How the Program Plays Tic-Tac-Toe

In the following discussion the computer plays 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 five tests. The first two correspond to steps 1 and 2 of the strategy 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.

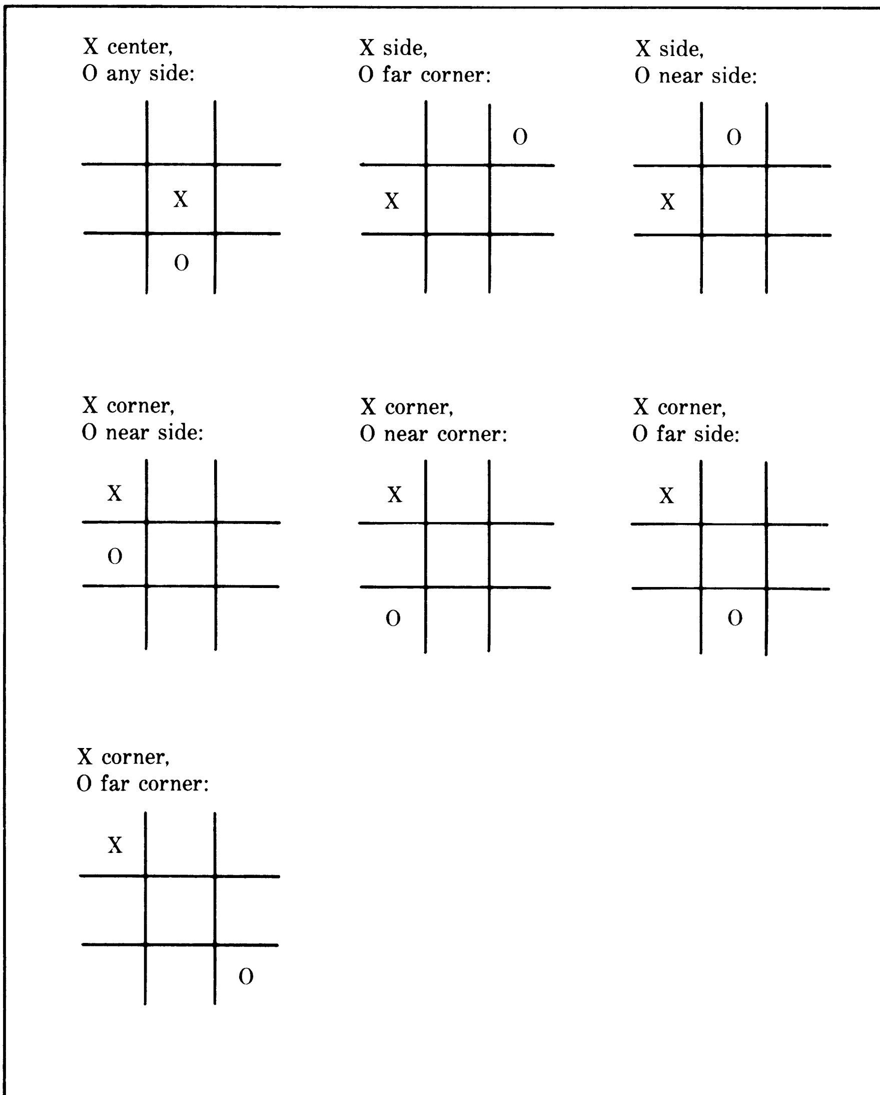


Figure 9-4. The seven losing positions for player O

2. If the program cannot find any winning marks, it checks whether it can block the opponent from winning 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 have passed test 4. It chooses the cell that has 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 aggresive strategy, hence the program's occasional willingness to settle for a draw when a win is possible.

—The Program

The first block resets the random number generator.

```
10 INPUT "ENTER A RANDOM NUMBER ";X
20 X=RND(-ABS(X))
```

Array Definitions

The next block creates several arrays and reads in certain data that is stored in the program:

```
30 DIM TC(3,3),OK(9,3),T(3,3),P(2),DI(4,2),
   PL(8,3),NW(2),P$(2)
40 FOR R=1 TO 3
50 FOR C=1 TO 3
60 READ T(R,C)
70 NEXT C,R
90 DATA 2,3,2,3,1,3,2,3,2
100 FOR DN=1 TO 4
110 FOR DV=1 TO 2
120 READ DI(DN,DV)
```

```
130 NEXT DV,DN
150 DATA 0,1,1,1,1,0,1,-1
160 FOR PN=1 TO 8
170 FOR PA=1 TO 3
180 READ PL(PN,PA)
190 NEXT PA,PN
210 DATA 1,1,1,1,1,2,1,1,3,1,2,3,1,3,4,1,3,3,
    2,1,1,3,1,1
220 RV$=CHR$(18): NM$=CHR$(146)
230 OC$(1)="-"
240 OC$(2)=CHR$(118): REM CROSS
250 OC$(3)=CHR$(119): REM CIRCLE
260 P$(1)="HUMAN"
270 P$(2)="COMPUTER"
280 SC(1)=0
290 SC(2)=0
300 TG=0
310 REM
320 REM
330 P(1)=1: REM 1ST PLAYER IS HUMAN
340 P(2)=2: REM 2ND PLAYER IS COMPUTER
350 GOTO 390
```

Refer to Figure 9-5 while reading the following explanations of the program's arrays.

Array TC(*row,column*) 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(*turn number, attribute*) keeps track of all the prospective cells that prevent the opponent from setting a trap on the next turn. T(*row,column*) 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(*player type*) keeps track of who the players are: *player type*=1 indicates a human, and *player type*=2 indicates the computer. Depending on how P(1) and P(2) are set, the program may play the computer against itself, the computer against a human, or it may allow two humans to play with no active involvement from the computer at all.

DI(*direction number, vector*) stores the direction increments of the four possible directions of a path. This same array was also used previously by the Hidden Words and Crossword Puzzle programs. PL(*path number, attribute*) stores information about the eight paths on a tic-tac-toe grid.

NW(*path number*) identifies paths that contain a specified number of one player's marks. P$(*player type*) stores the name assigned to each

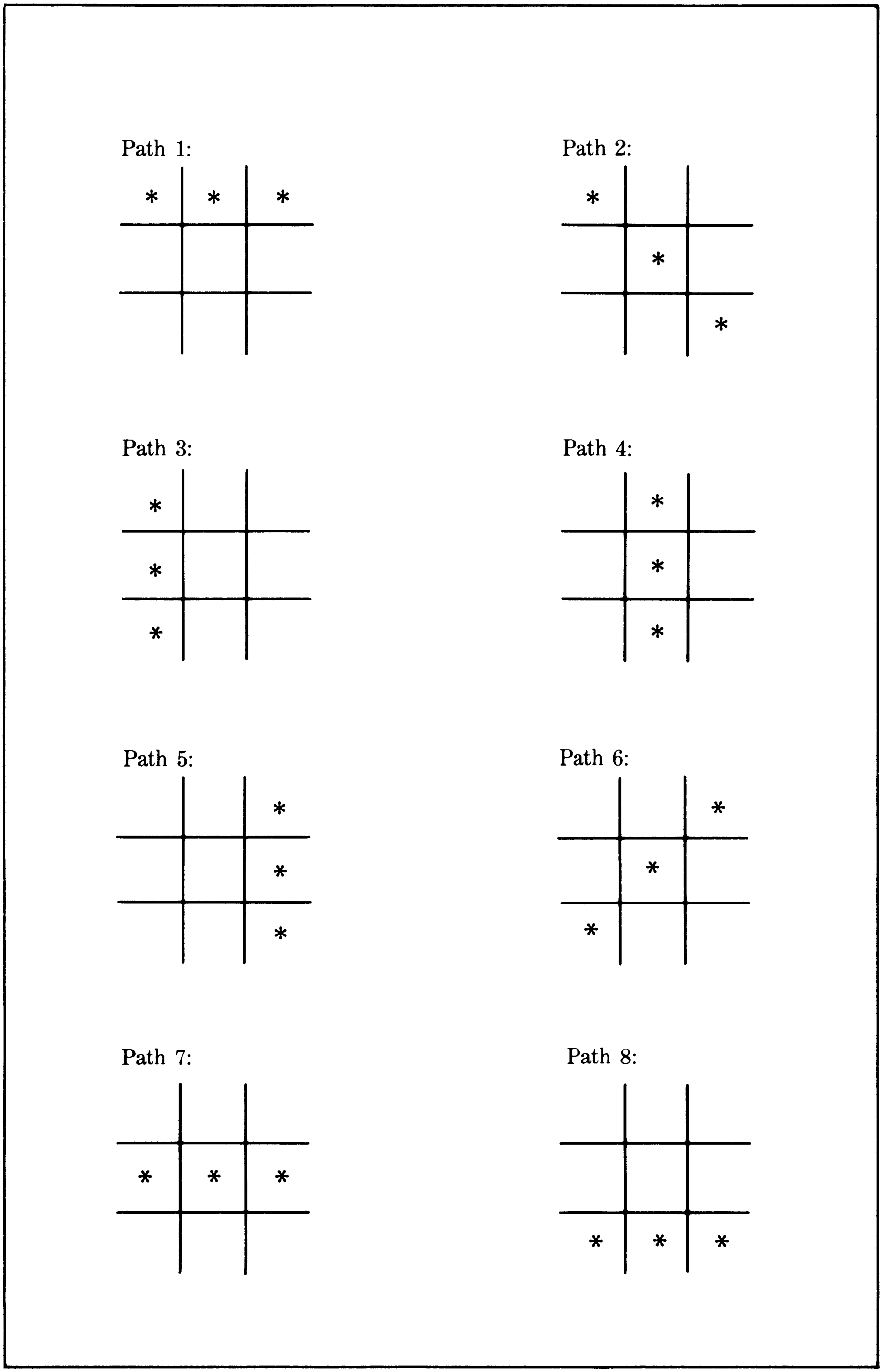
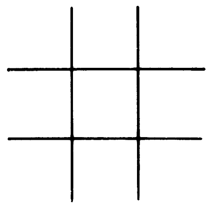


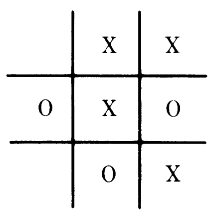
Figure 9-5. How several arrays and variables are used (*continued on next page*)

The array T(.) stores the type number of each cell: center=2, side=3



T(.)=			
	2	3	2
	3	1	3
	2	3	2

In the following situation, for player X, NW(1)=1 since path 1 contains 2 X's and no O's



T(.)=			
	0	1	1
	2	1	2
	0	2	1

Figure 9-5. How several arrays and variables are used (*continued*)

player type. "HUMAN" is used for player type 1, but you may change line 270 to use your own name instead. OC$(*character type*) stores the characters used when the grid is displayed: "X", "O", and "-". SC(*player type*) keeps track of how many wins each player has. The variable TG counts the number of tie games.

Lines 40-90 read the various cell types into T(,): 1=center, 2=corner, 3=side. Lines 100-150 read into DI(,) the direction vectors used to generate the eight possible tic-tac-toe paths. For path P, DI(P,1) is the row increment and DI(P,2) is the column increment.

Lines 160-210 read into PL(,) the attributes of the eight possible paths. For path P, PL(P,1) is the starting row, PL(P,2) is the starting column, and PL(P,3) is the direction.

Lines 330 and 340 determine who the players are. P(N)=1 means that player N is a human player, and P(N)=2 means that player N is the computer.

Starting a New Game

The next block of lines starts a new game:

```
360 PS=P(1)
370 P(1)=P(2)
380 P(2)=PS
390 PRINT CHR$(147)
400 PRINT "TIC-TAC-TOE"
410 MN=0
420 FOR R=1 TO 3
430 FOR C=1 TO 3
440 TC(R,C)=0
450 NEXT C,R
```

Lines 360-380 make the two players swap marks before each game. (Line 350 causes the program to skip these lines for the first game.) MN is the move number, initially 0. A complete move consists of two marks —one X and one O. Lines 420-450 empty the grid to get ready for a new game.

Getting the Next X or O

Now the program is ready to get a mark:

```
470 MN=MN+1
480 PN=1
490 P=0
500 GOSUB 2020
510 PRINT
520 PRINT P$(P(PN));" TO MARK AN ";OC$(PN+1)
530 ON P(PN) GOSUB 820,1000
```

First the move counter is incremented and the player number is set to 1 (lines 470 and 480). The subroutine called in line 500 prints the tic-tac-toe grid in its present state, and line 510 indicates whose turn it is to move.

Depending on the type of the current player (human or computer), line 530 calls one of two subroutines: one (at line 820) marks a cell from the keyboard; the other (at line 1000) marks a cell using program logic.

Evaluating the Results

After the human or computer makes a selection, the program evaluates its effect.

```
540 SL=3
550 SA=PN
560 GOSUB 2240
570 PRINT
580 IF N>0 THEN 640
590 IF MN=6 THEN 730
600 IF MN=5 THEN 690
610 IF PN=2 THEN 470
620 PN=PN+1
630 GOTO 490
640 P=NW(N)
650 GOSUB 2020
660 PRINT P$(P(PN));" WINS!"
670 SC(P(PN))=SC(P(PN))+1
680 GOTO 730
690 P=0
700 GOSUB 2020
710 PRINT "TIE GAME"
720 TG=TG+1
```

The subroutine called in line 560 searches all eight paths to see if the current player P(PN) has won. N>0 indicates a win; in that case, lines 640-680 announce the winner's name.

If N=0, the computer checks the turn number MN to see whether the game has ended in some other manner. There are only 9 cells in the grid, and each move puts two marks (an X and an O) on the grid. Ordinarily, MN can never exceed 5, since the X of move number 5 always takes the ninth cell (2+2+2+2+1=9 cells). However, if a player cancels the game, MN is set equal to 6. Line 590 detects that condition and jumps to the continuation menu.

When MN=5, the computer deduces a tie (there is no winner and all cells are marked) and announces that fact (line 600 and 690-720).

Otherwise, MN is less than 5, so the program gives the next player a turn.

The End of a Game

At the end of a game, the following lines print a continuation menu:

```
730 PRINT
735 EN=1
740 INPUT "ENTER 1 FOR NEW GAME, 2 TO QUIT ";EN
750 IF EN=1 THEN 360
760 IF EN<>2 THEN 730
770 PRINT
```

```
780 PRINT P$(1);" WON ";SC(1)
790 PRINT P$(2);" WON ";SC(2)
800 PRINT "TIE GAMES ";TG
810 END
```

If you elect to quit playing, lines 780-800 print the totals.

The two major subroutines "human's turn" and "computer's turn" are presented next.

Human's Turn

Here's the routine for the human's turn:

```
820 PRINT "WHICH CELL? ENTER ROW,COLUMN."
830 INPUT "(0,0 = NEW GAME) ";RM,CM
840 IF RM=0 AND CM=0 THEN 950
850 IF RM<1 OR RM>3 OR CM<1 OR CM>3 THEN 890
860 IF TC(RM,CM)=0 THEN 980
870 PRINT "NOT AVAILABLE"
880 GOTO 900
890 PRINT "INVALID MOVE"
900 PRINT "THE BOARD LOOKS LIKE THIS:"
910 P=0
920 GOSUB 2020
930 PRINT "NOW TRY AGAIN"
940 GOTO 820
950 PRINT "CANCELLED THAT GAME."
960 MN=6
970 RETURN
980 TC(RM,CM)=PN
990 RETURN
```

Lines 820 and 830 prompt the player to specify a cell in terms of its row and column number. Rows are numbered from top to bottom, columns from left to right.

If the player enters 0,0 for the row and column, the current game is canceled. Any other invalid row-column pair causes the program to reprint the current grid (lines 900-940) and repeat the prompt.

Given a valid row-column pair, line 860 determines whether that cell is empty: TC(RM,CM)=0. If the cell is empty, line 980 marks it. Line 990 returns to the main program.

Computer's Turn

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 lookahead logic.

Playing by the Book Here are the lines that handle the computer's first X or O:

```
1000 IF MN>1 THEN 1210
1010 IF PN<>1 THEN 1060
1020 GOSUB 2620
1030 RM=RT
1040 CM=CT
1050 GOTO 1990
1060 T=T(RM,CM)
1070 GOSUB 2620
1080 ON T GOTO 1090,1110,1140
1090 IF T(RT,CT)=3 THEN 1070
1100 GOTO 1180
1110 RT=2
1120 CT=2
1130 GOTO 1180
1140 ON T(RT,CT) GOTO 1180,1150,1170
1150 IF ABS(RT-RM)=2 OR ABS(CT-CM)=2 THEN 1070
1160 GOTO 1180
1170 IF ABS(RT-RM)=1 OR ABS(CT-CM)=1 THEN 1070
1180 RM=RT
1190 CM=CT
1200 GOTO 1990
```

If move number MN is greater than 1, line 1000 causes the program to jump to the lookaround program logic described in the next section. If MN=1 and PN=1, then it's time to make the first mark, an X. The subroutine called in line 1020 selects a cell at random, and lines 1030 and 1040 save the cell's address in variables RM and CM. Line 1050 jumps to the end of the computer's-turn subroutine.

Lines 1060-1200 take over when the computer is player O and the move number is 1. The general purpose of these lines is for player O to find a cell that avoids all seven losing game positions shown in Figure 9-4.

In line 1060, the variables RM and CM contain the row and column of the most recent move; in other words, they tell the program what cell contains an X. Line 1060 determines what type of cell—center, corner, or side—the X is in. Based on this information, the computer randomly selects an empty cell and checks to see whether that cell is safe given the location of the X. If the cell is not safe, the program randomly selects another cell and repeats the safety check.

The subroutine called in line 1070 randomly selects an empty cell T(RT,CT) as a candidate for player O's next move. Line 1080 jumps to the appropriate safety check depending on the type of cell that is already marked with an X.

Lines 1090-1100 handle the case of an X in the center; player O must not select a side cell (T=3).

Lines 1110-1130 handle the case of an X in the corner; player O must select the center cell.

Lines 1140-1170 handle the case of an X in the side; player O must not select a near side or the far corner.

Once the program has located a safe cell, lines 1180 and 1190 store its row and column address, and line 1200 jumps to the end of the computer's-turn subroutine.

Looking Ahead In the case of second and subsequent turns, the program no longer plays using prepared moves. It first checks to see whether it can win with one mark:

```
1210 IF MN>2 THEN 1240
1220 IF PN=2 THEN 1340
1230 GOTO 1500
1240 SA=PN
1250 SL=2
1260 GOSUB 2240
1270 IF N=0 THEN 1340
1280 M=INT(RND(1)*N)+1
1290 P=NW(M)
1300 GOSUB 2480
1310 RM=RO
1320 CM=CO
1330 GOTO 1990
```

Lines 1210-1230 check to see whether the computer is making its second mark. If it is, there's no point in looking for a winning cell yet (it takes three marks to fill a path). In the case of move 2 for player X, line 1230 jumps to a trap-prevention routine. In the case of move 2 for player O, line 1220 jumps to a trap-setting routine described later on.

Looking for a Winning Cell For move number MN equal to or greater than 3, lines 1240-1330 look for a winning cell. The subroutine called in line 1260 counts the number of unblocked paths containing at least two of player PN's marks. If N=0, there are none, so the program skips to the defensive move routine.

If N is greater than 0, then the array NW(,) lists the paths that contain winning cells. Line 1280 randomly selects one of these paths, and the subroutine called in line 1300 finds the row and column of the open cell in that path.

Now that the program has located a winning cell, lines 1310 and 1320 store its row and column address, and line 1330 jumps to the end of the computer's-turn subroutine.

Preventing Imminent Defeat If the program can find no winning cell, it next checks to see whether it must prevent its opponent from winning on his next turn:

```
1340 SA=3-PN
1350 SL=2
1360 GOSUB 2240
1370 IF N=0 THEN 1430
1380 P=NW(N)
1390 GOSUB 2480
1400 RM=RO
1410 CM=CO
1420 GOTO 1990
```

Line 1340 sets SA equal to the number of the opposing player (when PN is 1, ST is set to $3-1=2$; when PN is 2, ST is set to $3-2=1$). The subroutine called in line 1360 counts 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.

If N is not 0, there is at least one way for the opposing player to win on his next move. Lines 1380 and 1390 find the opponent's winning cell, and lines 1400 and 1410 store its row and column address so the computer can claim it. Line 1420 jumps to the end of the computer's-turn subroutine.

Setting a Trap If the computer still hasn't made a selection for player number PN, the computer now looks for a move that will trap the opponent and guarantee a win on the computer's next turn.

```
1430 IF MN=2 THEN 1500
1440 SA=PN
1450 GOSUB 2660
1460 IF N<>2 THEN 1500
1470 RM=RV
1480 CM=CV
1490 GOTO 1990
```

If the computer is making its second mark (MN=2), there is no way it can set a trap yet, so line 1430 causes the program to skip to the next logical block.

Otherwise, the program looks for a move that will create a trap. The subroutine called in line 1450 tests every empty cell to see which, if any, produces a trap. If N=2, the program has found such a cell, and lines 1470 and 1480 store the cell's row and column number so the computer can claim it. Line 1490 jumps to the end of the computer's-turn subroutine.

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:

```
1500 F=0
1510 FOR RM=1 TO 3
1520 FOR CM=1 TO 3
1530 IF TC(RM,CM)<>0 THEN 1760
1540 TC(RM,CM)=PN
1550 SA=PN
1560 SL=2
1570 GOSUB 2240
1580 IF N=0 THEN 1680
1590 IF MN=2 AND PN=1 THEN 1720
1600 P=NW(1)
1610 GOSUB 2480
1620 SA=3-PN
1630 TC(RO,CO)=SA
1640 SL=2
1650 GOSUB 2240
1660 TC(RO,CO)=0
1670 GOTO 1710
1680 IF MN=2 AND PN=1 THEN 1750
1690 SA=3-PN
1700 GOSUB 2660
1710 IF N=2 THEN 1750
1720 F=F+1
1730 OK(F,1)=RM
1740 OK(F,2)=CM
1750 TC(RM,CM)=0
1760 NEXT CM,RM
```

The variable F counts the number of safe cells (those that will prevent the opponent from setting a trap). In lines 1510-1700, the computer tries marking each empty cell in the grid (one at a time). For each cell marked, the program looks to see whether its opponent can set a trap.

In lines 1730 and 1740, 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 the heuristic method to choose among them:

```
1780 SL=2
1790 SA=3-PN
1800 FOR CN=1 TO F
1810 TC(OK(CN,1),OK(CN,2))=PN
1820 GOSUB 2240
1830 TC(OK(CN,1),OK(CN,2))=0
1840 OK(CN,3)=M
1850 NEXT CN
1860 IF F<>1 THEN 1890
1870 CN=1
1880 GOTO 1970
1890 SM=1
1900 FOR IT=2 TO F
1920 SM=IT
1930 NEXT IT
1940 CN=INT(RND(1)*F)+1
1950 IF OK(CN,3)=OK(SM,3) THEN 1970
1960 GOTO 1940
1970 RM=OK(CN,1)
1980 CM=OK(CN,2)
```

The program marks each safe cell (line 1810) and counts how many unblocked paths M remain. For each safe cell F, OK(F,3) stores the number of unblocked paths that remain when that cell is marked.

Lines 1900-1930 compare the results of these trial marks to see which marks result in the fewest number (SM) of unblocked paths. Lines 1940-1960 randomly pick safe cells until finding one that leaves SM unblocked paths.

Now that the program has located a suitable cell, lines 1970 and 1980 store its row and column address so the computer can claim it.

Ending the Computer's Turn The following lines end the computer's-turn subroutine.

```
1990 TC(RM,CM)=PN
2000 PRINT "COMPUTER TAKES ROW ";RM;" COLUMN ";CM
2010 RETURN
```

Line 1990 marks the player's number PN in grid location TC(RM,CM). Line 2000 announces the move, and the line 2010 returns to the main program.

Printing Subroutine

Here's the subroutine to print the tic-tac-toe board:

```
2020 QR=PL(P,1)
2030 QC=PL(P,2)
2040 DN=PL(P,3)
2050 QL=0
2060 PRINT
2070 FOR QI=1 TO 3
2080 PRINT SPC(3);
2090 FOR QJ=1 TO 3
2100 IF QL=3 OR QI<>QR OR QJ<>QC THEN 2180
2110 PRINT RV$;: REM REVERSE PRINTING
2120 PRINT OC$(TC(QI,QJ)+1);
2130 PRINT NM$;: REM NORMAL PRINTING
2140 QR=QR+DI(DN,1)
2150 QC=QC+DI(DN,2)
2160 QL=QL+1
2170 GOTO 2190
2180 PRINT OC$(TC(QI,QJ)+1);
2190 PRINT " ";: REM 1 SPACE IN QUOTES
2200 NEXT QJ
2210 PRINT
2220 NEXT QI
2230 RETURN
```

The subroutine is designed to highlight path P. This comes in handy whenever a game ends with a win: the computer highlights the winning path. The value of P (set before the subroutine is called) determines which path is highlighted. If P=0, no path is highlighted.

Line 2100 determines whether the next cell to be printed is part of the highlighted path. If it is, lines 2110 and 2120 print that cell flashing; otherwise, line 2180 prints it normally. Note that there is a single space in quotes in line 2190.

Auxiliary Subroutines

This subroutine analyzes the contents of all eight paths:

```
2240 N=0
2250 M=0
```

```
2260 FOR P=1 TO 8
2270 RU=PL(P,1)
2280 CU=PL(P,2)
2290 DN=PL(P,3)
2300 NF=0
2310 MF=0
2320 FOR CE=1 TO 3
2330 IF TC(RU,CU)=0 THEN 2380
2340 IF TC(RU,CU)=SA THEN 2370
2350 MF=MF+1
2360 GOTO 2380
2370 NF=NF+1
2380 RU=RU+DI(DN,1)
2390 CU=CU+DI(DN,2)
2400 NEXT CE
2410 IF NF<>SL OR MF>0 THEN 2440
2420 N=N+1
2430 NW(N)=P
2440 IF MF>0 THEN 2460
2450 M=M+1
2460 NEXT P
2470 RETURN
```

The variable N counts the number of unblocked paths containing at least SL of player SA's marks. Variable M counts the number of paths containing none of the other player's marks.

Upon return from this subroutine, the array NW(,) lists the path numbers of all unblocked paths containing at least SL of player SA's marks.

The following subroutine locates the first opening in path P:

```
2480 RO=0
2490 CO=0
2500 RT=PL(P,1)
2510 CT=PL(P,2)
2520 DN=PL(P,3)
2530 FOR CE=1 TO 3
2540 IF TC(RT,CT)<>0 THEN 2580
2550 RO=RT
2560 CO=CT
2570 CE=3
2580 RT=RT+DI(DN,1)
2590 CT=CT+DI(DN,2)
2600 NEXT CE
2610 RETURN
```

Upon return from the subroutine, RO is the row number of the open path and CO is the column number.

Here is the subroutine that randomly selects an empty cell:

```
2620 RT=INT(RND(1)*3)+1
2630 CT=INT(RND(1)*3)+1
2640 IF TC(RT,CT)<>0 THEN 2620
2650 RETURN
```

Upon return from the subroutine, RT is the open cell's row number and CT is the column number.

The last subroutine looks for an opportunity to set a trap (mark a cell that creates two winning threats for a player's next turn).

```
2660 FOR RB=1 TO 3
2670 FOR CB=1 TO 3
2680 IF TC(RB,CB)<>0 THEN 2780
2690 SL=2
2700 TC(RB,CB)=SA
2710 GOSUB 2240
2720 TC(RB,CB)=0
2730 IF N<2 THEN 2780
2740 RV=RB
2750 CV=CB
2760 CB=3
2770 RB=3
2780 NEXT CB,RB
2800 RETURN
```

On entry to the subroutine, SA is the number of the player trying 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 open cell that sets the trap.

—Using the Program

Figure 9-6 shows a sample run of the program.

It is very easy to modify the program. For example, you can set it so that two people can play against each other, rather than one person playing against the computer. Simply make these changes:

```
260 INPUT "ENTER THE NAME OF PLAYER 1: ";P$(1)
270 INPUT "ENTER THE NAME OF PLAYER 2: ";P$(2)
530 ON P(PN) GOSUB 820,820
```

Alternatively, you may find it interesting to watch the computer play against itself. Make these changes:

```
260 P$(1)="COMPUTER"
270 P$(2)="COMPUTER"
530 ON P(PN) GOSUB 1000,1000
```

```
ENTER A RANDOM NUMBER  25

TIC-TAC-TOE

     - - -
     - - -
     - - -

HUMAN TO MARK AN X
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  3 , 3


     - - -
     - - -
     - - X

COMPUTER TO MARK AN O
COMPUTER TAKES ROW  2  COLUMN  2


     - - -
     - O -
     - - X

HUMAN TO MARK AN X
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  2 , 3


     - - -
     - O X
     - - X

COMPUTER TO MARK AN O
COMPUTER TAKES ROW  1  COLUMN  3


     - - O
     - O X
     - - X

HUMAN TO MARK AN X
WHICH CELL? ENTER ROW,COLUMN.
```

Figure 9-6. Sample run of Tic-Tac-Toe (keyboard entries underlined)

```
(0,0 = NEW GAME)  3 , 1

    - - O
    - O X
    X - X

COMPUTER TO MARK AN O
COMPUTER TAKES ROW  3  COLUMN  2

    - - O
    - O X
    X O X

HUMAN TO MARK AN X
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  1 , 2

    - X O
    - O X
    X O X

COMPUTER TO MARK AN O
COMPUTER TAKES ROW  2  COLUMN  1

    - X O
    O O X
    X O X

HUMAN TO MARK AN X
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  1 , 1

    X X O
    O O X
    X O X
TIE GAME

ENTER 1 FOR NEW GAME, 2 TO QUIT  0
```

Figure 9-6. Sample run of Tic-Tac-Toe (keyboard entries underlined (*continued*)

```
ENTER 1 FOR NEW GAME, 2 TO QUIT  1

TIC-TAC-TOE

   - - -
   - - -
   - - -

COMPUTER TO MARK AN X
COMPUTER TAKES ROW  1  COLUMN  1

   X - -
   - - -
   - - -

HUMAN TO MARK AN O
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  2 , 1

   X - -
   O - -
   - - -

COMPUTER TO MARK AN X
COMPUTER TAKES ROW  2  COLUMN  2

   X - -
   O X -
   - - -

HUMAN TO MARK AN O
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  3 , 3

   X - -
   O X -
   - - O

COMPUTER TO MARK AN X
COMPUTER TAKES ROW  1  COLUMN  2
```

Figure 9-6. Sample run of Tic-Tac-Toe (keyboard entries underlined (*continued*)

```
    X X -
    O X -
    - - O

HUMAN TO MARK AN O
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  1 , 3

    X X O
    O X -
    - - O

COMPUTER TO MARK AN X
COMPUTER TAKES ROW  3  COLUMN  2

    X X O
    O X -
    - X O
COMPUTER WINS!

ENTER 1 FOR NEW GAME, 2 TO QUIT  1

TIC-TAC-TOE

    - - -
    - - -
    - - -

HUMAN TO MARK AN X
WHICH CELL? ENTER ROW,COLUMN.
(0,0 = NEW GAME)  0 , 0
CANCELLED THAT GAME.

ENTER 1 FOR NEW GAME, 2 TO QUIT  2

HUMAN WON  0
COMPUTER WON  1
TIE GAMES  1
```

Figure 9-6. Sample run of Tic-Tac-Toe (keyboard entries underlined (*continued*)

—How to Lower the Computer's IQ

After playing against the computer a while, you may find it frustrating that the computer never loses. If you play as well as the computer does, every game will end in a draw.

To add a little variety and uncertainty to the game, you can simplify the computer's playing strategy in several ways.

First you can eliminate the computer's prepared move for the first O by skipping that section of its logic. One change does this:

```
1010 REM  CHANGED FROM: IF PN<>1 THEN 1060
```

After you make this change, you'll notice the computer often stepping into the losing situations of Figure 9-4.

Alternatively or in addition, you can eliminate the computer's ability to set traps or detect them by making this change:

```
1500 GOTO 1020: REM CHANGED FROM: F=0
```

After you make this change, the computer's game playing will be reduced to the lowest level of strategy outlined at the beginning of this chapter. Even at this level, the computer may surprise you by making a (randomly) brilliant move.

Chapter 10

Quiz Master

Can you name the capital city of Illinois? What is the French word for acorn? How many grams are in an ounce? What baseball team won the pennant in 1968? Just about everybody needs to memorize something from time to time.

This chapter presents Quiz Master, a program that will help you learn information on any subject you choose. Quiz Master will ask you questions, check your answers, and keep your score until you have learned as much as you want. The program will even give hints to help you as you're learning.

What kinds of information can the program teach? Just about anything involving pairs of short facts: states and capitals, foreign language vocabulary, English and metric measures, ball teams and pennant years, events and dates, words and their synonyms, words and their antonyms, and so forth.

The program requires you to supply the data; this data is used to ask the questions as well as to give you the correct answers.

The data used is kept in what is commonly called a database. A database is a list of items that have something in common. It could be a list of names and addresses, metric and English conversions, and so on. We have included with the program two ready-to-use databases—the names of states and capitals and the names of French and English foods.

Like the data pairs, the database for the program also requires a title, two questions, and the number of data pairs there are. Here is a short example:

Title:	***Weights and Measures***
Question 1:	What is the English equivalent of . . . ?
Question 2:	What is the metric equivalent of . . . ?
Count:	6
Data pairs:	1 meter, 1.1 yard 1 liter, 1.06 U.S. quarts 1 kilogram, 2.2 pounds 0.9 meter, 1 yard 0.95 liter, 1 U.S. quart 454 grams, 1 pound

The title identifies the subject of the quiz. Questions 1 and 2 are used with the first and second items in each pair, thus allowing you to practice naming the second or first item of each pair. For example, a type 1 question might be: "What is the English equivalent of 1 meter?" and a type 2 question might be: "What is the metric equivalent of 2.2 pounds?"

The data pairs must be set up consistently so that the questions will always be applicable. In our example, the first item in each pair is a metric quantity and the second item is an English quantity.

—The Program

The first block resets the random number generator so the program will present you with a different series of questions each time you run it. The lines also clear the screen and set the color to green.

```
10 INPUT "ENTER A RANDOM NUMBER ";R
20 R=RND(-ABS(R))
25 PRINT CHR$(153);CHR$(147): REM LT GREEN &
   CLEAR SCREEN
```

Reading the Database

The next lines read in the database and print a title on the screen:

```
30 READ T$
40 PRINT T$
```

```
50 PRINT
60 DIM Q$(2)
70 READ Q$(1),Q$(2)
80 READ N
90 DIM L$(N,2),T(N),S(N)
100 FOR J=1 TO N
110 READ L$(J,1),L$(J,2)
120 NEXT J
125 NU$="": REM NO SPACES INSIDE QUOTES
126 RV$=CHR$(18): REM REVERSE ON
127 NR$=CHR$(146): REM REVERSE OFF
```

T$ contains the database title. The array Q$( ) contains the two questions. Array L$(,) holds the database. For instance, array element L$(1,2) contains the second item of the first data pair. Array S( ) is used to shuffle the questions so the computer won't ask them in the same order in case you repeat the quiz. NU$ is an empty or "null" string. There are no spaces inside the quotes.

Printing the Menu

Here are the lines to print the main menu:

```
130 TC=0
140 TH=0
150 FOR J=1 TO 2
160 PRINTJ; TAB(5); Q$(J)
170 NEXT J
180 C=0
185 INPUT "SELECT 1 OR 2 ";C
190 IF C<>1 AND C<>2 THEN 200
195 ON C GOTO 220,250
200 PRINT
210 GOTO 150
220 Q=1
230 A=2
240 GOTO 270
250 Q=2
260 A=1
```

Variable TC keeps track of the total correct answers during a single quiz. Lines 150-170 print the menu. Lines 180-210 get your selection and branch to the appropriate section of the program.

The variables Q and A, set in lines 220-260, keep track of which type of question you selected; a single routine handles both question types. For example, Q=2 and A=1 when you have selected question type 2.

Shuffling the Question Sequence

Now it's time for the program to shuffle the questions:

```
270 PRINT
280 PRINT "THERE WILL BE ";N;" QUESTIONS.
    STANDBY..."
290 PRINT
300 FOR J=1 TO N
310 T(J)=0
320 NEXT J
330 FOR J=1 TO N
340 R=INT(RND(1)*N)+1
350 IF T(R)=1 THEN 340
360 S(J)=R
370 T(R)=1
380 NEXT J
```

The lines set up a random sequence for asking the questions. Array S( ) contains the sequence. For any subscript N, S(N) specifies which data pair is used for the Nth question in the quiz.

Lines 300-320 set every element of T( ) to 0, indicating that none of the positions have been used yet. Lines 330-380 assign each data pair to a randomly selected position. If the position has already been assigned, that is, T(R)<>0 in line 350, the program selects another random number. The process continues until all of the data pairs have been assigned to a question in the quiz.

For instance, after line 380, the array S( ) might look like this:

S(1)=5
S(2)=1
S(3)=3
S(4)=2
S(5)=6
S(6)=4

indicating that the first question asked will involve data pair 5; the second question, data pair 1, and so forth.

Questions and Answers

Now comes the question-and-answer routine. First we present the lines that print the question and accept your answer:

```
390 FOR J=1 TO N
400 W1$=L$(S(J),0)
```

```
410 W2$=L$(S(J),A)
420 LA=LEN(W2$)
430 M$=NU$
440 FOR J1=1 TO LA
450 M$=M$+"*"
460 NEXT J1
470 PRINT
480 PRINT J;".  ";Q$(Q).
490 PRINT RV$;
500 PRINT W1$
510 PRINT NR$;
520 R$=NU$
525 INPUT R$
```

Line 390 causes J to count through each of the N questions. W1$ contains the word to be included in the question, and W2$ contains the correct answer. Lines 440-510 set up a mask M$ to be used in giving hints.

Lines 480-510 print the question, plugging in the appropriate word from the database. Line 525 inputs your answer.

Checking Your Response

The next lines evaluate your answer:

```
530 IF R$<>"/" THEN 560
540 J=N
550 GOTO 730
560 IF R$=W2$ THEN 660
570 IF M$=W2$ THEN 690
580 IF R$=NU$ THEN 600
590 PRINT "INCORRECT."
600 R$=NU$
605 INPUT "TYPE 1-HINT OR 2-GIVE UP ";R$
610 IF R$<>"1" THEN 640
620 GOSUB 800
630 GOTO 470
640 IF R$="2" THEN 690
650 GOTO 600
660 PRINT "CORRECT!"
670 TC=TC+1
680 GOTO 730
690 PRINT "THE CORRECT ANSWER IS"
700 PRINT RV$;
710 PRINT W2$
720 PRINT NR$;
730 NEXT J
```

If you type a slash "/" in response to the question, the program ends the quiz and prints your score up to that point. Otherwise, line 560 compares your answer with the correct answer.

If your answer is correct, the program jumps to a congratulation routine at line 660. Otherwise, the program prepares to give you a hint. Line 570 checks if the next hint completely reveals the answer; if it does, the program jumps to line 690, skipping the hint routine. Otherwise, the program asks whether you want to see the hint or give up. The subroutine called in line 620 performs the hint routine.

The following lines take over when you have tried all the questions or have stopped the quiz by answering with a "/".

```
740 PRINT
750 PRINT "YOU GOT ";TC;" CORRECT OUT OF ";N
760 PRINT "USING ";TH;" HINT(S)."
770 C=0
775 INPUT "TYPE 1-MORE PRACTICE 2-QUIT ";C
780 IF C=2 THEN END
790 GOTO 130
```

Here's the hint routine:

```
800 TH=TH+1
810 PRINT "EACH * STANDS FOR A MYSTERY LETTER."
820 PRINT "HERE IS YOUR HINT: ";
830 PRINT RV$;
840 PRINT M$
850 PRINT NR$;
860 R=INT(RND(1)*LA)+1
870 IF MID$(M$,R,1)<>"*" THEN 860
880 ZB$=MID$(W2$,R,1)
890 ZC$=NU$
900 IF R=1 THEN 920
910 ZC$=LEFT$(M$,R-1)
920 ZC$=ZC$+ZB$
930 IF LEN(M$)-LEN(ZB$)-R+1=0 THEN 950
940 ZC$=ZC$+RIGHT$(M$,LEN(M$)-LEN(ZB$)-R+1)
950 M$=ZC$
960 RETURN
```

Whenever you request a hint, the program increments the hints used total TH (line 800), gives you the hint M$ (lines 820-850), and modifies M$ to produce the next hint (lines 860-950).

Initially, the hint consists of a string of asterisks, one for each character in the answer. After each hint is given, a randomly selected asterisk is replaced with the character that belongs in that position. Line 860 randomly gets a character for position R, ranging from 1 to LA (the length of the answer). Line 870 determines whether that character has been revealed yet in the hint. If it has, the program tries another character position. If it hasn't, the program uncovers the corresponding character (lines 880-950).

The Database

The only thing missing from our program now is the database. This is where you customize the program. Store the database in DATA statements starting with line 970. Here is a short sample database, French and English foods. Input the lines so you can test the program:

```
970 DATA *** FRENCH/ENGLISH FOODS ***
980 DATA WHAT IS THE FRENCH WORD FOR...
990 DATA WHAT IS THE ENGLISH WORD FOR...
1000 DATA 24
1010 DATA ACORN, LE GLAND DU CHENE, APPLE,LA POMME
1020 DATA ASPARAGUS, LES ASPERGES, BEEF, LE BOEUF
1030 DATA BREAD, LE PAIN, BUTTER, LE BEURRE
1040 DATA CAULIFLOWER, LE CHOU-FLEUR, CHEESE,
     LE FROMAGE
1050 DATA DATE, LA DATTE, DOUGHNUT,
     LE PET DE NONNE
1060 DATA EGG, L'OUEF, EGGPLANT, LA AUBERGINE
1070 DATA FISH, LE POISSON, GINGERBREAD,
     LE PAIN D'EPICE
1080 DATA GRAPEFRUIT, LA PAMPLEMOUSSE, GRAPE,
     LE GRAIN DE RAISIN
1090 DATA HONEY, LE MIEL, LEMON, LE CITRON
1100 DATA MUTTON, LE MOUTON, PEACH, LA PECHE
1110 DATA SUGAR, LE SUCRE, SYRUP, LE SIROP
1120 DATA TURKEY, LE DINDON, YAM, L'IGNAME
```

Another useful database, states and capitals, is given next:

```
970 DATA *** STATES AND CAPITALS ***
980 DATA WHAT IS THE CAPITAL OF...
990 DATA WHAT STATE HAS THE CAPITAL CITY OF...
1000 DATA 50
1010 DATA ALABAMA, MONTGOMERY, ALASKA, JUNEAU
1020 DATA ARIZONA, PHOENIX, ARKANSAS, LITTLE ROCK
1030 DATA CALIFORNIA, SACRAMENTO, COLORADO, DENVER
1040 DATA CONNECTICUT, HARTFORD, DELAWARE, DOVER
1050 DATA FLORIDA, TALLAHASEE, GEORGIA, ATLANTA
1060 DATA HAWAII, HONOLULU, IDAHO, BOISE
```

```
ENTER A RANDOM NUMBER  12321

*** STATES AND CAPITALS ***

 1       WHAT IS THE CAPITAL OF...
 2       WHAT STATE HAS THE CAPITAL CITY OF...
SELECT 1 OR 2  1

THERE WILL BE  50  QUESTIONS. STANDBY...


 1 . WHAT IS THE CAPITAL OF...
[illegible]
NYC
INCORRECT.
TYPE 1-HINT OR 2-GIVE UP 1
EACH * STANDS FOR A MYSTERY LETTER.
HERE IS YOUR HINT: [illegible]

 1 . WHAT IS THE CAPITAL OF...
[illegible]
?
INCORRECT.
TYPE 1-HINT OR 2-GIVE UP 1
EACH * STANDS FOR A MYSTERY LETTER.
HERE IS YOUR HINT: [illegible]

 1 . WHAT IS THE CAPITAL OF...
[illegible]
ALBANY
CORRECT!

 2 . WHAT IS THE CAPITAL OF...
[illegible]
AUSTIN
CORRECT!

 3 . WHAT IS THE CAPITAL OF...
[illegible]
?
INCORRECT.
TYPE 1-HINT OR 2-GIVE UP 1
EACH * STANDS FOR A MYSTERY LETTER.
HERE IS YOUR HINT: [illegible]
```

Figure 10-1. Sample run of states and capitals quiz

```
 3 . WHAT IS THE CAPITAL OF...
[illegible]
?
INCORRECT.
TYPE 1-HINT OR 2-GIVE UP 1
EACH * STANDS FOR A MYSTERY LETTER.
HERE IS YOUR HINT: [illegible]

 3 . WHAT IS THE CAPITAL OF...
[illegible]
?
INCORRECT.
TYPE 1-HINT OR 2-GIVE UP 2
THE CORRECT ANSWER IS
[illegible]

 4 . WHAT IS THE CAPITAL OF...
[illegible]
?

YOU GOT  2  CORRECT OUT OF  50
USING  5  HINT(S).
```

Figure 10-1. Sample run of states and capitals quiz (*continued*)

```
1070 DATA ILLINOIS, SPRINGFIELD, INDIANA,
     INDIANAPOLIS
1080 DATA IOWA, DES MOINES, KANSAS, TOPEKA
1090 DATA KENTUCKY, FRANKFORT, LOUISIANA,
     BATON ROUGE
1100 DATA MAINE, AUGUSTA, MARYLAND, ANNAPOLIS
1110 DATA MASSACHUSETTS, BOSTON, MICHIGAN, LANSING
1120 DATA MINNESOTA, ST. PAUL, MISSISSIPPI, JACKSON
1130 DATA MISSOURI, JEFFERSON CITY, MONTANA, HELENA
1140 DATA NEBRASKA, LINCOLN, NEVADA, CARSON CITY
1150 DATA NEW HAMPSHIRE, CONCORD, NEW JERSEY,
     TRENTON
1160 DATA NEW MEXICO, SANTA FE, NEW YORK, ALBANY
1170 DATA NORTH CAROLINA, RALEIGH, NORTH DAKOTA,
     BISMARCK
1180 DATA OHIO, COLUMBUS, OKLAHOMA, OKLAHOMA CITY
1190 DATA OREGON, SALEM, PENNSYLVANIA, HARRISBURG
```

```
1200 DATA RHODE ISLAND, PROVIDENCE, SOUTH CAROLINA,
     COLUMBIA
1210 DATA SOUTH DAKOTA, PIERRE, TENNESSEE,
     NASHVILLE
1220 DATA TEXAS, AUSTIN, UTAH, SALT LAKE CITY
1230 DATA VERMONT, MONTPELIER, VIRGINIA, RIGHMOND
1240 DATA WASHINGTON, OLYMPIA, WEST VIRGINIA,
     CHARLESTON
1250 DATA WISCONSIN, MADISON, WYOMING, CHEYENNE
```

Figure 10-1 shows a sample run of the program using the states and capitals database.

Use your imagination to think up other possibilities. Just be sure to set up the database along the same lines as the examples.

Chapter 11

Speed Drills

This chapter's program, Speed Drills, can help you master situations like the following:

- 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 the answer comes to you.
- 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.

These are typical situations that require you to think fast on your feet. This program will help you to add, subtract, multiply, and divide quickly and easily.

The method used is drill and practice, but with a timer added. You specify the range of numbers to be used and the time limit per question. You can even set an error tolerance of 0 to 25 percent. This is used in case you're more interested in learning to make quick estimates than to calculate exact answers.

—Program Operation

The program starts by displaying the menu shown in Figure 11-1.

Items 1 through 4 determine what kind of drill is used: item 1 indicates that the current operation is addition; item 2 shows the respective ranges for the first and second operands, A and B (10 to 99 in both cases).

Item 3 lists the error tolerance: 0 percent, meaning that no error is allowed. An error tolerance of 25 would mean that your answer could be within 25 percentage points of the correct answer and still be counted as correct.

Item 4 gives the time limit (in seconds) for answering each question. A value of 0 means no time limit—you have all the time you want to answer each question.

To change any of the settings, enter the corresponding item number. To start the drill, press RETURN on an empty line.

When you start the drill, the program randomly chooses operands A and B according to the specified range and operation and displays an incomplete equation. For example:

$$50 + 85 =$$

Type in the answer (use DEL to erase errors). Press RETURN when you are finished. If time runs out before you press RETURN, the pro-

```
              MATH DRILLS

 1-OPERATION: A + B

 2-RANGES:  10 <=A<= 99 &  10 <=B<= 99

 3-ERROR TOLERANCE:  0 %

 4-TIME LIMIT:  10  SECONDS
 ----------------------------------------

 SELECT <1>-<4> TO CHANGE DRILL
   OR <RETURN> TO START
```

Figure 11-1. Start-up menu

gram will accept whatever you have typed in so far. The program will then tell you whether your answer is correct or close enough (within the specified error tolerance).

You can then press RETURN to continue with another question, C to set up new drill parameters, or S to stop. In the latter case, the program will print out your score.

—Program Listing

The first block sets up the screen colors and resets the random number generator so you will receive a different set of problems each time you run the program:

```
1 POKE 53280,1: REM WHITE BORDER
2 POKE 53281,1: REM WHITE SCREEN
3 PRINT CHR$(154);: REM LIGHT BLUE
10 INPUT "ENTER A RANDOM NUMBER ";R
20 R=RND(-ABS(R))
```

Line 10 prompts you to enter a randomly chosen number, which is used as the seed to determine the subsequent results of the RND function. If you enter the same number each time you run the program and if the drill settings are the same, you will receive the same series of problems.

Setting Up the Variables

The next lines set up arrays, counters, and other control variables:

```
30 DIM L(2),U(2),A(2),OD$(2),CU$(2)
40 OD$(1)="A"
50 OD$(2)="B"
60 EN$=CHR$(13)
70 BK$=CHR$(157)
80 DL$=CHR$(20)
88 S1$=" ": REM 1 SPACE INSIDE QUOTES
90 NU$="": REM NO SPACES INSIDE QUOTES
92 RC$=CHR$(17)
93 LC=25
94 GOSUB 1860
95 VM$=SO$
96 RC$=CHR$(102)
97 LC=40
98 GOSUB 1860
99 LD$=SO$
```

```
100 CL$=CHR$(147): REM CLEAR SCREEN
103 HO$=CHR$(19): REM HOME CURSOR
105 RV$=CHR$(18): REM REVERSE ON
107 NR$=CHR$(146): REM REVERSE OFF
108 CU$(1)=RV$+S1$+NR$+BK$
109 CU$(2)=S1$+BK$
110 VW=40
130 READ OP$,L(1),U(1),L(2),U(2),ER,TL
140 DATA   +,  10,  99,  10,  99, 0,10
145 GOSUB 1930: REM SET UP TONE REG'S
150 GOTO 420
160 KR=0
170 KQ=0
```

The arrays store the various drill settings and parameters for the two operands A and B. L( ) stores lower limits; U( ) stores upper limits; A( ), the values assigned to operands A and B; and OD$, the operand names "A" and "B".

Lines 40-109 store certain keyboard and video codes required for some special techniques used in the keyboard entry phase of the program.

EN$ is the RETURN character. BK$ is the cursor-left character. DL$ is the DEL character; the command PRINT DL$ causes the cursor to back up and erase the preceding character. S1$ is a single space and NU$ is a null (empty) string.

Lines 92-95 store 25 consecutive cursor-down codes in VM$; the string is used to position the cursor on the screen in later parts of the program. Lines 96-99 store 40 consecutive dashes in LD$.

The screen control codes CL$, HO$, RV$, and NR$ are explained by remarks in the program. CU$(1) and CU$(2) are solid and reverse blocks; they are used to create a blinking cursor effect during the actual math drill.

Line 110 stores your display's width (characters per line). Line 130 reads the initial settings for the current operation, the lower and upper operand limits, the error tolerance, and the time limit. Change any of the values in DATA line 140 to make the drill start with the type of problem you want. If you change the data, be sure to list your new data in the proper order:

140 DATA *operator, lower limit for A, upper limit for A, lower limit for B, upper limit for B, error tolerance, time limit*

The subroutine called in line 145 sets up the tone generator, which is used to signify time out during a drill. Line 150 jumps to a block that checks the validity of the math operator (the first item in line 140).

Lines 160 and 170 return the counters to zero for *correct answers* (KR) and *questions attempted* (KQ).

Main Menu

The following lines present the main menu:

```
180 PRINT CL$
190 PRINT SPC((VW-11)/2)"MATH DRILLS"
200 PRINT
210 PRINT "1-OPERATION: A ";OP$;" B"
220 PRINT
230 PRINT "2-RANGES: ";L(1);"<=A<=";U(1);
240 PRINT "& ";L(2);"<=B<=";U(2)
250 PRINT
260 PRINT "3-ERROR TOLERANCE: ";INT(ER*100+.5);"%"
270 PRINT
280 PRINT "4-TIME LIMIT: ";TL;" SECONDS"
290 PRINT LD$
320 PRINT
330 PRINT "SELECT <1>-<4> TO CHANGE DRILL"
340 PRINT "  OR <RETURN> TO START ";
350 S=0
360 INPUT S
370 IF S<0 OR S>4 THEN 180
380 PRINT
390 ON S+1 GOTO 760,400,490,660,720
```

Lines 210-280 display the current drill parameter settings. Lines 360-390 get your selection and respond accordingly.

Changing the Arithmetic Operator The following routines take care of the four options presented in the main menu. Here's option 1 (change operator):

```
400 PRINT "SELECT AN OPERATION: + - * / ";
405 OP$=NU$
410 INPUT OP$
420 Q1$="+-*/": REM NO SPACES INSIDE QUOTES
430 Q2$=OP$
440 QP=1
450 GOSUB 1780
460 IP=QF
470 IF IP=0 THEN 400
480 GOTO 180
```

The operator you enter is stored in OP$. The subroutine called in line 450 ensures that OP$ is among the valid operators (+, −, *, and /).

Changing the Limits Here's menu option 2 (change operand ranges):

```
490 FOR J=1 TO 2
500 PRINT "LOWER LIMIT FOR ";OD$(J);
510 INPUT L(J)
520 IF L(J)>=0 THEN 550
530 PRINT "MUST BE > OR = 0"
540 GOTO 500
550 PRINT "UPPER LIMIT FOR ";OD$(J);
560 INPUT U(J)
570 IF J=1 OR IP<4 OR U(J)>0 THEN 610
580 PRINT "FOR DIVISION, UPPER LIMIT OF B"
590 PRINT "MUST BE > 0"
600 GOTO 550
610 IF L(J)<=U(J) THEN 640
620 PRINT "MUST BE < OR = ";L(J)
630 GOTO 550
640 NEXT J
650 GOTO 180
```

Lines 520-540 require that the lower limits for both operands be nonnegative. Lines 570-600 require that operand B's lower limit be greater than 0 when the operation is division; this prevents an attempt to divide by 0. Lines 610-630 require that the upper limit of an operand be greater than or equal to the operand's lower limit.

Setting the Error Tolerance Option 3 sets the error tolerance:

```
660 PRINT "ENTER ERROR TOLERANCE, 0-25%"
670 PRINT "(0 = NO MARGIN OF ERROR) ";
680 ER=0
685 INPUT ER
690 IF ER<0 0 OR ER>25 THEN 660
700 ER=ER/100
710 GOTO 180
```

Line 690 ensures that the tolerance you enter is between 0 and 25 percent. Line 700 converts the percentage into a decimal ratio.

Resetting the Time Limit The next block handles option 4 (reset time limit):

```
720 PRINT "ENTER TIME LIMIT, 0-120 SECONDS"
730 INPUT "(0=NO LIMIT) ";TL
740 IF TL<0 OR TL>120 THEN 720
750 GOTO 180
```

Starting the Drill

If you select the start drill option, the program continues with the following block:

```
760 IF IP<>2 OR U(1)>=L(2) THEN 800
770 PRINT "ADJUST RANGES SO THAT"
780 PRINT "UPPER LIMIT A IS > OR = LOWER LIMIT B"
790 GOTO 490
800 IF IP<4 OR U(2)>0 THEN 850
810 PRINT "UPPER LIMIT FOR OPERAND B (DIVISOR)"
820 PRINT "MUST BE >0. ENTER NEW UPPER LIMIT ";
830 INPUT U(2)
840 IF U(2)<=0 THEN 810
```

These lines make a final check of the operand ranges to ensure that subtraction problems will always produce a nonnegative result (760-790) and that division by zero is not attempted (800-840).

Now the program generates random values for operands A and B:

```
850 FOR J=1 TO 2
860 N1=L(J)
870 N2=U(J)
880 GOSUB 1750
890 IF J=2 AND IP=4 AND NR=0 THEN 880
900 A(J)=NR
910 NEXT J
```

The subroutine called in line 880 gets a random value between N1 and N2 inclusive. The subroutine is called twice, once for each operand, with N1 and N2 set accordingly. The random values for A and B are stored in A(1) and A(2) respectively. Line 890 prevents an attempt at division by 0 and gets a new divisor if necessary.

The following lines compute the correct answer, depending on which operation has been selected:

```
920 ON IP GOTO 930,950,980,1000
930 R=A(1)+A(2)
940 GOTO 1010
950 IF A(1)<A(2) THEN 850
960 R=A(1)-A(2)
970 GOTO 1010
980 R=A(1)*A(2)
990 GOTO 1010
1000 R=A(1)/A(2)
1010 TR=ABS(R*ER)
```

IP ranges from 1 to 4, depending on which operation has been selected. Line 920 selects the appropriate program logic. The result of the operation is stored in R.

Whichever operation has been selected, the program continues at line 1010, which uses the error tolerance ER to compute the allowable error. For example, given an error tolerance of 10 percent (ER=0.1) and a correct answer of 34, the allowable error is 3.4.

Displaying the Problem

The program has the answer figured out now, so it is ready to display the problem:

```
1020 PRINT CL$
1022 CP=11
1024 GOSUB 1910
1025 PRINT LD$
1026 CP=5
1027 GOSUB 1910
1030 PRINT SPC(3); A(1);OP$;A(2);"= ";
1040 TI$="000000": REM ZERO TIMER
1050 CO=1
1060 G$=NU$
1063 POKE 198,0: REM EMPTY KEYBOARD BUFFER
1065 W=1
1070 PRINT CU$(SGN(W+1)+1);: REM PRINT CURSOR
1075 W=-W: REM SWITCH CURSORS FOR NEXT TIME
1080 TF=0: REM TIME-OUT FLAG
```

Lines 1020-1025 clear the screen and divide it into two windows. Line 1030 prints the problem. Line 1040 resets the C-64's timer to zero. The variable CO, initialized in line 1050, keeps track of the cursor position on the display. G$ will hold your answer in string form; line 1060 initially sets it to a null or empty string value.

Line 1070 prints a character that serves as the cursor position indicator. Line 1180 resets the time-out flag TF to zero, indicating that you haven't run out of time yet.

Inputting Your Answer

The next lines handle your keyboard input during the timed portion of the program.

```
1110 GET K$
1120 IF K$=NU$ THEN 1380
1160 REM
```

For timed input, the program cannot use INPUT, which causes the computer to stop and wait. During the wait, the program would not be able to check the timer. Instead, the program uses the GET statement, which gets a character, if it is available, but does not stop and wait for one.

Line 1110 tries to get the character into K$. If no key is available, K$ is set equal to the null string. In that case, line 1120 causes the computer to go directly to the check-timer routine starting at line 1380.

If K$ is not a null string, the following block determines whether it is an acceptable character for the program:

```
1170 IF K$=EN$ THEN 1420
1180 IF K$=DL$ AND CO>1 THEN 1270
1190 IF (K$<"0" OR K$>"9") AND K$<>"." THEN 1380
1240 G$=G$+K$
1250 CO=CO+1
1260 GOTO 1340
1270 IF CO<>2 THEN 1300
1280 G$=NU$
1290 GOTO 1310
1300 G$=LEFT$(G$,CO-2)
1310 CO=CO-1
1330 PRINT CU$(2);:REM ERASE CURSOR
1340 PRINT K$;
```

Line 1170 checks whether you have pressed RETURN, signaling that your answer is ready. Line 1180 checks whether you have pressed DEL to delete a character; if you have, lines 1270-1330 handle it.

Line 1190 checks to see whether you have entered one of the other allowable characters. If you have, program block 1240-1260 adds that character to your input field G$, increments the position counter CO, and jumps to the character display statement at line 1340.

After completing each keyboard input cycle, the program updates the timer:

```
1380 TM=VAL(TI$)
1390 IF TL=0 OR TM<TL THEN 1070
1400 TF=1
```

Line 1380 converts the C-64's timer value into a number TM. If TM is less than the time limit or the time limit is 0, line 1390 jumps back to the keyboard input routine. Otherwise, your time is out, so line 1400 sets the time out flag.

Checking Your Answer

When you press RETURN or time runs out, the program evaluates your answer:

```
1420 G=VAL(G$)
1430 KQ=KQ+1
1440 PRINT CU$(2): REM ERASE CURSOR
1442 CP=8
1444 GOSUB 1910
1450 IF TF=0 THEN 1470
1460 GOSUB 2010: REM SOUND BUZZER
1465 PRINT " TIME'S UP"
1470 IF ABS(G-R)<=TR THEN 1520
1490 PRINT " INCORRECT. ";
1500 PRINT " CORRECT ANSWER IS ";R
1510 GOTO 1590
1520 KR=KR+1
1530 IF R<>G THEN 1570
1540 PRINT
1550 PRINT " CORRECT"
1560 GOTO 1590
1570 REM
1580 PRINT "CLOSE ENOUGH! THE EXACT ANSWER IS ";R
```

Line 1420 converts G$ (your input) into a numeric value. Line 1430 updates the questions-attempted counter. Line 1440 erases the blinking cursor. Line 1442-1444 position the cursor to line 8 before the program prints its evaluation message. If time elapsed before you pressed RETURN, lines 1460-1465 sound a "buzzer." At this point, whatever you have typed in so far is accepted as your answer.

Line 1470 checks to see whether your answer is close enough to the correct answer. If it is not, lines 1490-1510 are run; otherwise, lines 1520-1580 are performed.

Continuation Menu

The following block prints a continuation menu offering three options: continue, change drill, or stop.

```
1590 CP=14
1595 GOSUB 1910
1597 PRINT " PRESS..."
1598 PRINT
1600 PRINT SPC(8);RV$;" RETURN ";NR$;" TO CONTINUE"
1605 PRINT
1610 PRINT SPC(8);RV$;" C ";NR$;" TO CHANGE DRILL"
```

```
1615 PRINT
1620 PRINT SPC(8);RV$;" S ";NR$;" TO STOP"
1623 POKE 198,0: REM EMPTY KEYBOARD BUFFER
1625 GET CQ$
1630 IF CQ$=EN$ THEN 760
1640 IF CQ$="S" THEN 1670
1650 IF CQ$="C" THEN 180
1660 GOTO 1625
1670 PRINT CL$
1680 CP=6
1690 GOSUB 1910
1700 REM
1705 PRINT " YOUR SCORE:"
1710 PRINT
1715 PRINT " PROBLEMS TRIED: ";KQ
1720 PRINT
1725 PRINT " ANSWERED CORRECTLY: ";KR
1730 PRINT
1735 PRINT " SCORE: ";INT(KR/KQ*100+.5) ;"%"
1740 PRINT
1745 END
```

Lines 1590-1595 position the cursor inside the lower screen "window." Lines 1597-1620 print the three options. Lines 1625-1660 get your answer and respond accordingly.

If you selected the Stop option, lines 1705-1745 print your cumulative scores and end the program.

Subroutines

The following subroutine returns a random integer between N1 and N2 (inclusive):

```
1750 NG=N2-N1+1
1760 NR=INT(RND(1)*NG)+N1
1770 RETURN
```

NR is the random integer.

Here is the string search subroutine:

```
1780 QF=0
1790 IF Q2$=NU$ THEN RETURN
1800 IF QP+LEN(Q2$)-1>LEN(Q1$) THEN RETURN
1810 IF MID$(Q1$,QP,LEN(Q2$))=Q2$ THEN 1840
1820 QP=QP+1
1830 GOTO 1800
1840 QF=QP
1850 RETURN
```

On entry to the subroutine, Q1$ is the string to be searched, Q2$ is the string to find, and QF is the starting position. Upon return from the subroutine, QF is the position at which Q2$ begins in Q1$. QF=0 indicates that the string was not found or that Q2$ was an empty string.

The next subroutine builds up a string of consecutive characters:

```
1860 SO$=""
1870 FOR K=1 TO LC
1880 SO$=SO$+RC$
1890 NEXT K
1900 RETURN
```

The subroutine stores the number LC of character RC$ in the string SO$.

These next lines position the cursor to a specified display row:

```
1910 PRINT HO$;LEFT$(VM$,CP);
1920 RETURN
```

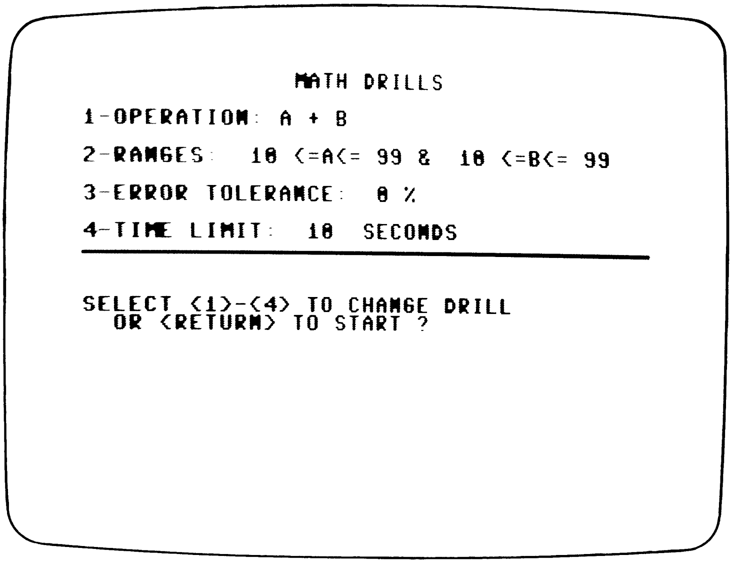
CP is the destination row number ranging from 0 (top row) to 24 (bottom row).

The following subroutine initializes the C-64's tone registers to produce a time out buzzer sound. However, these lines do not actually produce the sound:

```
1930 FOR R=54272 TO 54295
1932 POKE R,0
1934 NEXT R
1936 POKE 54296,15
1940 BR=54272
1950 FOR R=BR TO BR+6
1960 READ VV
1970 POKE R,VV
1980 NEXT R
1985 REM  F1   F2   P1   P2   WF   AD   SR
1990 DATA 195, 16, 15, 15, 00, 00, 240
2000 RETURN
```

Finally, here's the subroutine to sound the buzzer for about one second.

```
2010 POKE BR+4,0
2020 POKE BR+4,65
2030 REM POKE 54296,15
2040 FOR J=1 TO 300
2050 NEXT J
2060 POKE BR+4,0
2070 RETURN
```



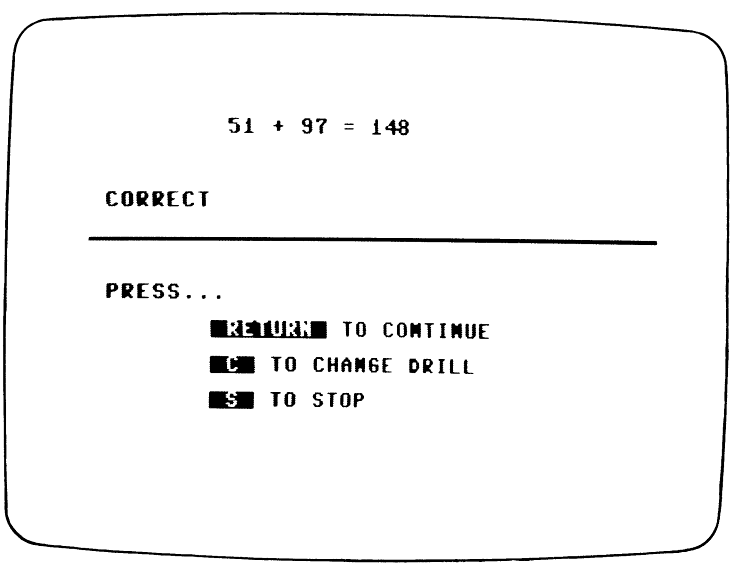
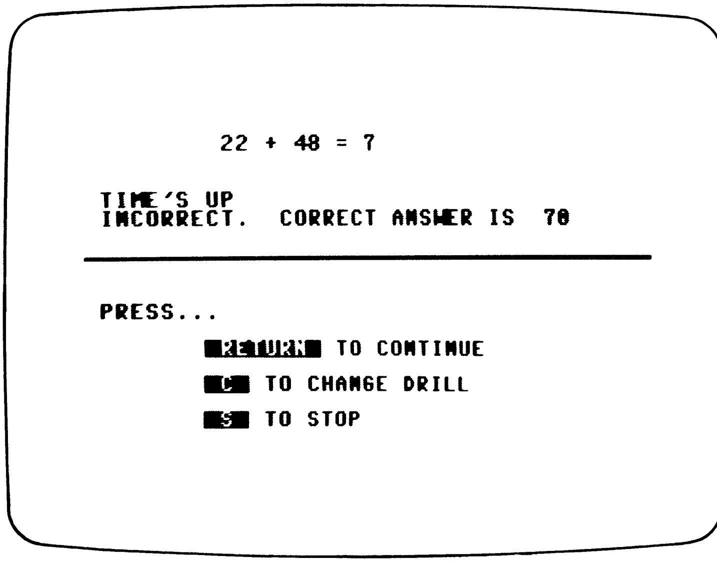


Figure 11-2. Sample run of Speed Drills



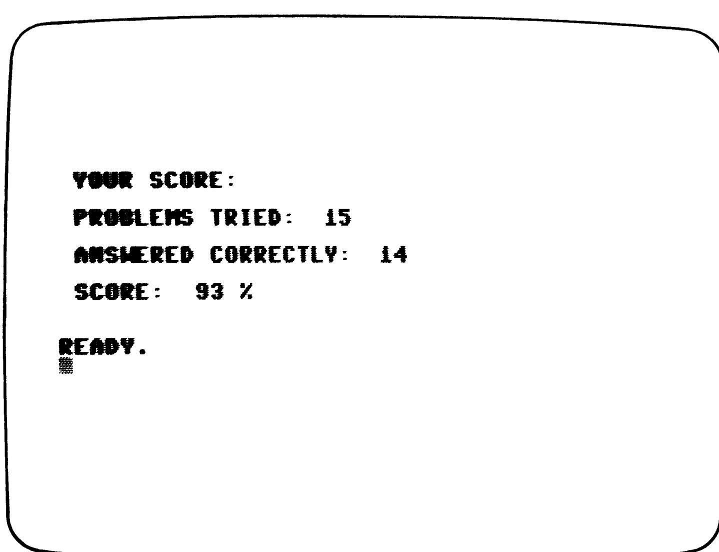


Figure 11-2. Sample run of Speed Drills (*continued*)

—Hints for Using the Program

Figure 11-2 shows a sample run of the program.

When you first run the program, select a time limit of 0 (no limit) and then start the drill. Practice using the keyboard input routine, testing RETURN and DEL and all the acceptable input characters (digits and decimal point). Occasionally there will be a slight delay between typing a character and seeing it on the display.

When you have the feel of the input process, reset the drill with a timer setting of 200. If you can get the correct answer 75 percent of the time, reduce the time limit. Once you can answer 75 percent of the problems within the new time limit, reduce the time limit again. Repeat the process for various operations and operand ranges.

Now you're ready for life in the fast lane!

Chapter 12

Text Scanner

What constitutes good writing? Many factors are involved, and some of them (style, for instance) are quite difficult to evaluate with a computer. Other writing elements, however, are easily measured by the computer's statistical powers. This chapter presents Text Scanner, a program that measures average sentence length and average word length.

With this program you'll be able to measure your own writing as well as your favorite passages from literature. You can compare the analyses of samples from scientific journals, newspapers, computer manuals, and so forth. With these results, you can draw your own conclusions about the effects of sentence and word lengths on readability.

The program can read word-processed documents stored in disk files as well as samples you type in at the keyboard.

In addition to its practical applications, Text Scanner illustrates several useful techniques for text processing.

—How Text Scanner Works

Here is the overall structure of the program. It starts with the general outline shown in Figure 12-1.

The outline is written in a near-English form called pseudo-code.

```
Main program:
  Initialization:
    Sentence_delimiters are: period, exclamation point, question mark, and
          end_of_text_marker
    Word_delimiters are: sentence_delimiters, space, hyphen, end_of_line,
          and double_quote
    Letters are: upper and lower case alphabet
    Sentence-, word-, character-, and letter-counters = 0
    End initialization block
  Print title and menu
  Analyze text
  Print statistics
  End main program

Analyze_text
  End_of_text=false
  Do until end_of_text=true:
    Analyze_a_sentence
    If end_of_sentence=true then add 1 to sentence_counter
    End do-block
  End analyze_text procedure

Analyze_a_sentence
  End_of_sentence=false
  In_a_sentence=false
  Do until end_of_sentence=true or end_of_text=true:
    Analyze_a_word
    If end_of_word=true then add 1 to word_counter
    End do-block
  End analyze_a_sentence procedure

Analyze_a_word
  End_of_word=false
  In_a_word=false
  Do until end_of_word=true or end_of_sentence=true or
        end_of_text=true:
    Get character C
    If C=end_of_text_marker then end_of_text=true
    If C is a letter of alphabet then do:
      If in_a_word=false then in_a_word=true
      If in_a_sentence=false then in_a_sentence=true
      Add 1 to letter_counter
      End if-block
    If C is a word-delimiter then do:
      If in_a_word=true then end_of_word=true
      If C is a sentence-delimiter and in_a_sentence=true then
          end_of_sentence=true
      End if-block
    If end_of_text=false then add 1 to character_counter
    End do-block
  End analyze_a_word procedure
```

Figure 12-1. Program description in structured pseudo-code

The form of the pseudo-code emphasizes the program's logical structure. This structure is made up of a main program with three auxiliary procedures. In the BASIC program presented in the next sections, the procedures are treated as subroutines. In the pseudo-code outline, an underscore character is used to connect words that correspond to a definite entity (a variable or a procedure).

Text Scanner counts sentences, words, and letters. The task sounds simple, but it is actually quite complex.

To count sentences, the program looks for sentence delimiters—a period, an exclamation point, or a question mark. But what if the text contains an ellipsis—three periods in a row? Or suppose a very dramatic passage ends with two exclamation points. The program cannot assume that a sentence has ended each time it reads a delimiter.

To count words, the program looks for word delimiters—a space, a hyphen, or a carriage return. The program must again watch for sequences of delimiters; otherwise, two hyphens (--) used as a dash would appear to mark two words instead of one.

The answer to these problems is that the program must keep track of whether it is in the middle of a sentence or a word. For example, if a delimiter is encountered in the middle of a word, then the end of the word has been found. But if the delimiter is not in the middle of a word when reached, the delimiter has no effect on the word total.

This explains the use of the true/false variables in the in_a_sentence, in_a_word, end_of_text, end_of_sentence, and end_of_word procedures (refer to Figure 12-1).

Other punctuation also requires special consideration. Apostrophes, for example, must not be treated as word delimiters; otherwise, *didn't* would be treated as two words. These punctuation marks should also be ignored in counting the length of a word.

—The Program

The BASIC program is presented in logical blocks. Type them in as you go along. The first block defines certain delimiters and returns various counters used in the program to zero. The first block appears as follows:

```
10 PRINT CHR$(147)
20 S1$=" ": REM 1 SPACE INSIDE QUOTES
30 NU$="": REM NO SPACES INSIDE QUOTES
40 ET$=NU$
```

```
50 EL$=CHR$(13)
60 CR=0
70 EL=1
80 NS=0
90 NW=0
100 NC=0
110 NT=0
130 PRINT
```

ET$ is an arbitrarily chosen end-of-text character. We've assigned it the value of a null string " ", but any character may be used. When the program reads this character, it sets the end_of_text=true. EL$ is the end-of-line character stored when you press RETURN. It counts as a word delimiter.

CR stores the number of characters left in the text input buffer. EL is another status variable used in the keyboard input logic. NS, NW, NC, and NT count the number of sentences, words, characters, and letters that have been read.

Printing the Title and Menu

The next block prints a title and menu:

```
140 PRINT "THE TEXT SCANNER"
150 PRINT
160 PRINT "INPUT FROM: 1-KEYBOARD  2-DISK"
165 INPUT IM
170 IF IM<>1 AND IM<>2 THEN 160
180 IF IM=1 THEN 220
```

Before using option 2 (input text from disk), you must put a text file on the disk using a word processing program. The file should contain the same type of information as might be entered from the keyboard. Carriage returns will be treated as word delimiters. Other control characters like line feeds, tabs, and form feeds will have no effect on the program's analysis.

If you select option 2, the following lines let you specify the input file name:

```
200 INPUT "VIEW DISK FILE DIRECTORY (Y/N) ";YN$
202 IF YN$="Y" THEN GOSUB 2900
204 PRINT
205 INPUT "NAME OF THE INPUT FILE ";FI$
210 OPEN 1,8,2,FI$+",SEQ,READ"
```

Analyzing the Input Text

The next block performs the analyze_text procedure:

```
220 ET=0
230 GOSUB 370
240 IF ES=0 THEN 260
250 NS=NS+1
260 IF ET=0 THEN 230
```

At the beginning of this routine, ET is set equal to false (throughout the program, 0 indicates false and 1 indicates true). The subroutine called in line 230 analyzes a word. Line 240 verifies whether a sentence has ended during the execution of the analyze_word subroutine. If so, the sentence counter is incremented. The program next checks to see if the end of text was reached. If not (ET=0, or false), the program jumps back to line 230 to repeat the analyze_a_sentence procedure.

If the end of text has been reached (ET=1), the program continues with the next block, which prints the statistics and ends the program:

```
270 PRINT
280 PRINT "SENTENCES: ";NS
290 PRINT "WORDS: ";NW
300 IF NS=0 THEN 330
310 SA=INT(NW/NS*100+.5)/100
320 PRINT "AVERAGE SENTENCE LENGTH: ";SA;" WORDS"
330 IF NW=0 THEN 360
340 WA=INT(NT/NW*100+.5)/100
350 PRINT "AVERAGE WORD LENGTH: ";WA;" LETTERS
360 PRINT "TOTAL CHARACTERS READ: ";NC
365 END
```

The calculations for averaging sentence and word lengths are simple:

$$\textit{Average sentence length} = \textit{words/sentences}$$

and

$$\textit{Average word length} = \textit{letters/words}$$

Lines 300 and 330 prevent division by zero in case no words or no sentences were found in the text. Line 310 calculates the average sentence length rounded to two decimal places, and line 340 calculates the average word length rounded to two decimal places.

Analyzing a Sentence

The following block performs the analyze_a_sentence subroutine:

```
370 ES=0
380 IS=0
390 GOSUB 440
400 IF EW=0 THEN 420
410 NW=NW+1
420 IF ES=0 AND ET=0 THEN 390
430 RETURN
```

First the end_of_sentence and in_a_sentence flags are set equal to 0 (false). Then line 390 calls the analyze_a_word subroutine. If a word was ended during the execution of the subroutine, the program adds 1 to the word total. If the program has found neither the end of a sentence nor the end of the text (ES=0 and ET=0), it goes back to the analyze_a_word subroutine.

If a sentence has ended or the end of text has been reached, the subroutine ends and returns control to the main program.

Analyzing a Word

Here is the analyze_a_word subroutine:

```
440 EW=0
450 IW=0
460 ON IM GOSUB 600,770
470 IF C$=ET$ THEN ET=1
480 IF C$=ET$ OR C$<"A" OR C$>CHR$(122) OR (C$>"Z"
    AND C$<CHR$(97)) THEN 530
490 IF IW=0 THEN IW=1
500 IF IS=0 THEN IS=1
510 NT=NT+1
520 GOTO 560
530 IF C$="." OR C$="!" OR C$="?" OR C$=S1$
    THEN 540
532 IF C$<>"-" AND C$<>EL$ THEN 560
540 IF IW=1 THEN EW=1
550 IF (C$="." OR C$="!" OR C$="?") AND IS=1
    THEN ES=1
560 IF ET=1 THEN 590
570 NC=NC+1
580 IF EW=0 AND ES=0 AND ET=0 THEN 460
590 RETURN
```

The end_of_word and in_a_word flags are set to zero (false) in lines 440 and 450. Then the program gets a character from the text buffer. Line 460 gets a character from either the keyboard or the disk file, depending on the value of IM you specified previously.

Upon return from either of the subroutines (at 600 or 770), C$ contains the character. If the end of text was reached, C$ is equal to the special end-of-text character ET$; in that case, line 470 sets the end_of_text flag to 1 (true).

Line 480 determines whether C$ is a letter. If it is, lines 490-510 make the necessary changes to the in_a_word flag, in_a_sentence flag, and letter_counter.

Lines 530-532 determine whethr C$ is a letter. If it is, line 540 checks the status of the in_a_word flag; if the flag is 1 (true), the delimiter ends the word, so the end_of_word flag EW is set to 1.

Line 550 checks whether C$ is a sentence delimiter and changes the end_of_sentence flag EF if appropriate.

Line 570 adds 1 to the character_count unless C$ is the end-of-text character.

If the end_of_word, end_of_sentence, and end_of_text flags are all 0 (false), the program jumps back to get another character. If any of them is true, the subroutine ends and returns to the analyze_a_sentence procedure.

Inputting From the Keyboard

You must type a quotation mark at the beginning of each line of text you enter. Otherwise, Commodore BASIC's input routine will stop at the first comma you type.

The keyboard input subroutine allows you to enter text without worrying a great deal about line breaks: you can press RETURN after any word or sentence. Be sure not to press RETURN in the middle of a word, because RETURN counts as a word delimiter. To end keyboard entry, press RETURN on an empty line.

```
600 C$=ET$
610 IF CR>0 THEN 730
620 IF EL=1 THEN 660
630 C$=EL$
640 EL=1
650 GOTO 760
660 PRINT
```

```
670 PRINT "TYPE A QUOTE, THEN ENTER TEXT"
680 B$=NU$
690 INPUT B$
700 BL=LEN(B$)
710 CR=BL
720 IF CR=0 THEN 760
730 EL=0
740 C$=MID$(B$,BL-CR+1,1)
750 CR=CR-1
760 RETURN
```

C$ is initially set to the end-of-text marker. The subroutine will return with this character only if you press RETURN on an empty line or if you type the end-of-text character somewhere inside a line.

The subroutine draws characters one at a time from a buffer B$. When the buffer is empty (CR=0), the program prompts you to enter another line. However, before doing this, the program must account for the RETURN you pressed to end the line. Line 630 sets C$ to this character and jumps to the end of the subroutine.

However, suppose you have pressed RETURN on an empty line to signify the end of text. In this case only, EL is set equal to 0 (line 730) so that the next time the program tries to read the buffer, line 620 will discover that EL=0 and will not try to get another line of input.

Inputting From a Disk File

The following lines read from a text file:

```
770 GET#1,C$
771 PRINT C$;: REM SHOW EACH CHARACTER AS ITS READ
772 IF ST=0 THEN 780
774 IF ST<>64 THEN PRINT "FILE ";FI$;" IS NOT
    AVAILABLE."
776 C$=ET$
780 IF C$=ET$ THEN CLOSE 1
790 RETURN
```

Line 770 gets a character from the disk file. In case of a file input error other than end of file, line 774 prints an error message along with the current statistics. If the character is the predesignated end-of-file marker ET$, or if the end of file is reached, line 780 closes the file. Line 790 ends the subroutine.

Disk Directory

Here's the subroutine to read a disk directory:

```
2900 PRINT "LOADING DIRECTORY..."
2901 OPEN 1,8,4,"$,SEQ,READ"
2902 IW=0
2910 IF ST=64 THEN 2980
2920 GET#1,A$
2922 IF LEN(A$)=0 THEN 2902
2930 IF A$>CHR$(31) AND A$<CHR$(122) THEN 2938
2932 IF IW=0 THEN 2910
2934 IW=0
2935 PRINT
2937 GOTO 2910
2938 IW=1
2939 PRINT A$;
2940 GOTO 2910
2980 CLOSE 1
2985 PRINT
2990 RETURN
```

Refer to Chapter 5 for an explanation of the logic used (the line numbers are identical).

—Using the Program

Passages from *Scientific American* magazine, William Faulkner, and Ernest Hemingway were run through the program. Here are the results:

	Scientific American	**Faulkner**	**Hemingway**
Average sentence length	14.00	20.55	16.92
Average word	4.76	3.92	4.46

Where does your writing fall on the scale?

Chapter 13

Guess My Word

How do you learn to spell a list of words when no one's around to call them out to you? One way is to memorize the entire list and practice writing it. The problem with this method is that it encourages you to learn the words in a fixed sequence. Later you may draw a blank when trying to spell a word out of sequence. This chapter's program, Guess My Word, offers an interesting alternative.

Given a word list that you provide, the program randomly selects a word and prompts you to guess what it is.

The program is almost identical to the game Hangman but without the hangman imagery. In our program, a panic meter shows the number of incorrect guesses you make for each word. The object of the game is to guess the word before the meter reading goes off the scale.

How Guess My Word Works

The program first gives a clue as to how long the word is: it displays one hyphen for each letter. The player then attempts to guess the letters of the word. Each time the player guesses a letter correctly, the program fills in the corresponding blank or blanks.

Each time the player guesses incorrectly, the panic level on the meter increases. The program allows the player to make 10 wrong guesses before the meter fills up; however, you can easily modify the program to increase or decrease the number of incorrect guesses allowed.

Your word list can consist of a group of words on a given topic or it may be a collection of words that are hard to spell. (The sample run of this program shown later in the chapter uses a list of elements as an example.)

You have a great deal of freedom to select words for the list. Include as many words as you wish, subject to your computer's memory limitations. The only limitation is that words can be no longer than 19 letters.

—The Program

The first part of the program sets up a large number of constants. The explanation for most of the lines is provided in REM (remark) statements at the end of each line.

```
14 TC$=CHR$(31): REM TEXT COLOR, BLUE
16 RD$=CHR$(28): REM RED
18 GC$=CHR$(30): REM GREEN
20 NF=10: REM NUMBER OF ERRORS ALLOWED
21 IF NF<1 OR NF>13 THEN PRINT "ERROR: NF OUT
   OF RANGE IN LINE 20": STOP
22 S1$=" ": REM 1 SPACE INSIDE QUOTES
24 NU$="": REM NO SPACES INSIDE QUOTES
```

NF in line 20 determines the number of incorrect errors you can make before the program gives you the answer. You can set this to any value between 1 and 13, inclusive. Line 21 ensures that NF is within range before continuing.

S1$ is a single space. NU$ is simply a null or empty string; there are no spaces inside the quotes.

The next block builds up some longer string constants:

```
26 RC$=CHR$(17): REM CURSOR DOWN 1
28 LC=24
30 GOSUB 1920
32 CD$=SO$: REM CURSOR DOWN 24
34 RC$=CHR$(29): REM CURSOR RIGHT 1
36 LC=40
38 GOSUB 1920
40 CR$=SO$: REM CURSOR RIGHT 40
```

```
42 RC$=CHR$(157): REM CURSOR LEFT 1
44 LC=40
46 GOSUB 1920
48 CF$=SO$: REM CURSOR LEFT 40
50 RC$=CHR$(183): REM HIGH DASH
52 LC=NF*3-2
54 GOSUB 1920
56 TL$=SO$: REM TOP LINE OF BOX
58 RC$=CHR$(175): REM LOW DASH
60 LC=NF*3-2
62 GOSUB 1920
64 BL$=SO$: REM BOTTOM LINE OF BOX
68 RC$=CHR$(32)
70 LC=40
72 GOSUB 1920
74 SS$=SO$: REM 40 BLANK SPACES
```

The subroutine called in line 30 and in several other lines creates a string of repeating characters. LC is the length of the string, and RC$ is the character that is repeated. The variables CD$, CR$, and CF$ are used to control the cursor position.

The following lines creates the panic meter—a single string of graphic and cursor control characters:

```
76 BX$=CHR$(111)+LEFT$(TL$,NF*3-2)
77 BX$=BX$+CHR$(112): REM TOP OF BOX
78 BX$=BX$+CHR$(17)+LEFT$(CF$,NF*3)
80 BX$=BX$+CHR$(108)+LEFT$(BL$,NF*3-2)
81 BX$=BX$+CHR$(186): REM BOTTOM OF BOX
82 VW=40
84 BB=INT((VW-NF*3)/2): REM BOX COL.
86 BA=18: REM BOX ROW
88 HO$=CHR$(19): REM HOME CURSOR
90 CS$=CHR$(147): REM CLEAR SCREEN
92 LL=(VW-26)/2
```

VW is the display width. BA and BB are the row and column locations for the panic meter on the screen display. LL is the column location for the available character list (the alphabet) on the screen display.

Reading the Word List

The next block reads in the word list and certain other data.

```
310 READ TL$
320 READ NW
330 DIM WL$(NW),WU(NW),RV$(2)
```

```
332 RV$(1)=CHR$(18): REM REVERSE ON
334 RV$(2)=CHR$(146): REM REVERSE OFF
340 FOR J=1 TO NW
350 READ WL$(J)
360 WL$(J)=LEFT$(WL$(J),19): REM LIMIT LENGTH TO 19
370 WU(J)=0
380 NEXT J
410 AZ$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
430 WT=0: REM WORDS TRIED
440 WC=0: REM WORDS GUESSED
```

Lines 310-380 read the vocabulary title TL$, the number of words NW, and the words themselves WL$( ). WU( ) keeps track of which words have been used during the running of the program. Line 360 ensures that none of the words exceeds 19 characters. Longer words would upset the carefully formatted display used during each round of the word-guessing game.

Displaying the Title and Instructions

The next lines print a title and brief instructions on the screen.

```
442 POKE 53280,1: REM WHITE BORDER
444 POKE 53281,1: REM WHITE BACKGROUND
450 PRINT CS$;TC$
460 INPUT "ENTER A RANDOM NUMBER ";R
470 R=RND(-ABS(R))
475 PRINT CS$
480 PRINT TAB((VW-21)/2);"GUESS MY WORD"
490 PRINT
500 PRINT "GUESS ONE LETTER AT A TIME"
510 PRINT
520 PRINT "IF YOU MAKE ";NF;" WRONG GUESSES,"
530 PRINT "I WIN. IF YOU GUESS ALL THE LETTERS"
540 PRINT "OF THE WORD, YOU WIN."
542 PRINT
550 PRINT "THE SUBJECT IS ";RV$(1);TL$;NR$
560 C1=23
562 C2=(VW-13)/2
564 PT$="PRESS ANY KEY"
570 GOSUB 1510: REM PROMPT FOR ANY KEY
```

Line 550 prints the subject of the vocabulary list. In the example used in this chapter, the vocabulary contains a list of chemical elements, so the title is "Elements."

The subroutine called in line 570 prompts the user to press any key and then waits until the key has been pressed.

Starting the Game

The next lines control the game play and the continuation menu:

```
580 PRINT CS$
640 GOSUB 770: REM PLAY ONE ROUND
645 PRINT CS$;LEFT$(CD$,12);
650 IF WT=NW THEN 720
660 INPUT "PLAY AGAIN (Y/N) ";YN$
670 IF YN$="N" THEN 730
680 IF YN$="Y" THEN 580
690 GOTO 660
720 PRINT "NO MORE WORDS LEFT."
730 PRINT "WORDS TRIED: ";WT
740 PRINT "CORRECT ANSWERS: ";WC
760 END
```

The subroutine called in line 640 plays one round of the game. Upon return from the guess-a-word subroutine, line 650 checks if any words remain. When WT (words tried)=NW (number of words), no words remain, so lines 720-760 end the game.

Lines 660-690 give the player a chance to continue playing or to quit. If the player elects to quit, lines 730 and 740 print the total words tried and total correct answers.

Guess-A-Word Subroutine

This subroutine is the heart of the program. As usual, it will be presented in several blocks.

```
770 GOSUB 1934: REM DRAW BOX
780 F=0: REM NUMBER OF ERRORS
800 BF=0: REM BLANKS FILLED
810 LA$=AZ$
820 WT=WT+1
830 WN=INT(RND(1)*NW)+1
840 IF WU(WN)=1 THEN 830
850 WU(WN)=1
860 W$=WL$(WN)
870 LE=LEN(W$)
872 BC=(VW-(LE+1)*2)/2
880 C2=BC
890 C1=2
900 GOSUB 1930
910 FOR LP=1 TO LE
920 PRINT S1$;"-";
930 NEXT LP
```

```
940 PRINT LEFT$(CD$,1)
960 PRINT LEFT$(CR$,LL);LA$
```

Lines 780 and 800 set the error and blanks-filled counters to 0. Line 810 sets the letters-available list LA$ to include the entire alphabet. The program keeps an updated list of letters available on the screen to help the player remember which letters have been tried.

Line 820 increments the words-tried counter WT. Line 830 randomly selects a word number WN. Line 840 checks the list of words used WU( ); if WU(WN)=1, the word has already been used, so the program goes back to line 830 for another word number.

In line 860, W$ stores the selected word. Lines 870-930 set up the word clue, which consists of a single hyphen for each letter of the word. BC is the starting column for the clue.

Line 960 prints the list of letters available—at this point, all 26 letters of the alphabet.

Inputting a Letter The next lines prompt the player to guess a letter:

```
970 C1=7
972 C2=(VW-15)/2
974 GOSUB 1930
980 PRINT "GUESS A LETTER ";
990 C2=POS(0)
1000 CL$=LA$
1010 GOSUB 1690
```

The subroutine called in line 1010 waits for the player to type one of the characters in CL$; since CL$=Q1$, the subroutine waits for the player to type one of the available letters.

Upon return from the subroutine, the program has the player's guess stored in LA$. It removes the guessed letter from LA$ so that letter won't be tried again:

```
1020 C2=0
1030 GOSUB 1930
1032 PRINT SS$;
1040 L$=C$
1050 ZA$=LA$
1060 ZB$=S1$
1070 ZP=QF
1080 GOSUB 1840
1090 LA$=ZA$
1100 C1=4
1110 C2=LL
```

```
1112 GOSUB 1930
1120 PRINT LA$
```

Line 1030 erases the GUESS A LETTER prompt. Lines 1040-1090 remove the letter guessed from the list of letters available, LA$. Actually, the letter L$ is replaced with a blank space S1$. Line 1120 prints the updated LA$.

Displaying Correct Letters The program next locates every occurrence of the guessed letter L$ in the secret word W$:

```
1130 SP=1
1140 LF=0
1150 Q1$=W$
1160 Q2$=L$
1170 Q0=SP
1180 GOSUB 1770
1190 IF QF=0 THEN 1270
1200 LF=LF+1
1210 C2=BC+1+(QF-1)*2
1220 C1=2
1222 GOSUB 1930
1230 PRINT L$;
1240 SP=QF+1
1250 GOTO 1170
```

The variable LF keeps track of the number of times the letter occurs in W$. Each time a letter is found, line 1230 prints it in the corresponding blank space in the clue.

Checking for Completed Words After counting and marking all occurrences, the program considers several possibilities:

```
1270 IF LF=0 THEN 1400
1280 BF=BF+LF
1290 IF BF<LE THEN 970
1300 WC=WC+1
1310 C1=7
1320 C2=(VW-7)/2
1330 GOSUB 1930
1340 PRINT GC$;"GOOD!!!";TC$
1350 C1=23
1360 C2=(VW-13)/2
1370 PT$="PRESS ANY KEY"
1380 GOSUB 1510
1390 RETURN
```

If LF=0, no occurrences were found, so the program continues at the incorrect guess block (line 1400). If LF>0, line 1280 adds LF to the blanks-filled total. In line 1290 if BF<LE, all blanks have not yet been filled, so the program goes back to line 970 to prompt the player for another guess. Otherwise, all blanks are filled, so the player has guessed the entire word correctly.

Line 1300 increments the words-correct total. Line 1340 prints a congratulatory message, and lines 1350-1380 print a continuation message.

Handling an Incorrect Guess Here is the block that takes over when the player guesses an incorrect letter:

```
1400 F=F+1
1410 GOSUB 1960
1420 IF F<NF THEN 970
1430 C1=7
1440 C2=(VW-32)/2
1450 GOSUB 1930
1460 PRINT RD$;"NO MORE TURNS LEFT. ANSWER IS..."
1470 PRINT LEFT$(CR$,(VW-LE)/2);GC$;W$;TC$;
1480 GOTO 1350
```

Line 1400 adds 1 to the error total. Line 1410 increments the panic meter. In line 1420 if F<NF, the player still has chances remaining, so the program jumps back to line 970 to get another guess.

Otherwise, no chances remain, so line 1460 prints the correct answer. The program then jumps back to the continuation message block.

Unrestricted Character-Input Subroutine

This subroutine displays a blinking prompt message and waits until a character is pressed before returning control to the line that invoked the subroutine:

```
1510 POKE 198,0: REM EMPTY KEYBOARD BUFFER
1512 W=1
1518 PRINT RV$(SGN(W+1)+1);: REM SWITCH
     FOREGROUND/BACKGROUND
1520 W=-W: REM NEXT TIME DO OPPOSITE
1530 GOSUB 1930
1540 PRINT PT$;
1550 GET C$
1560 FOR XX=1 TO 30
1570 NEXT XX
1580 IF C$=NU$ THEN 1518
```

```
1590 GOSUB 1930
1600 PRINT RV$(2);LEFT$(SS$,LEN(PT$));
1620 RETURN
```

The C-64 has a keystroke buffer that allows you to enter information even before the computer requests it. Such an advanced entry might be confusing in this application, so line 1510 erases any keystrokes before continuing.

Lines 1512 to 1520 set the display for normal or reverse color, depending on the value of W. The subroutine called in line 1930 positions the cursor to row C1, column C2 (set previously), and line 1540 prints the prompt stored in PT$. Line 1550 checks the keyboard for an available character; if no key has been pressed, the program reprints the prompt—this time in the opposite coloration (reverse/normal). If a key has been pressed, line 1590-1600 erases the prompt and line 1620 ends the subroutine.

Restricted Character-Input Subroutine

The next subroutine also gets a single keystroke; however, unlike the previous subroutine, this one will only accept characters from a specified set:

```
1690 PT$=S1$
1692 GOSUB 1510
1700 IF C$=S1$ THEN 1690
1710 Q0=1
1720 Q1$=CL$
1730 Q2$=C$
1740 GOSUB 1770
1750 IF QF=0 THEN 1690
1760 RETURN
```

On entry to the subroutine, CL$ contains the acceptable character list and C1,C2 specifies the row and column where the prompt should appear.

The subroutine sets the prompt equal to a single space and then calls the unrestricted character input subroutine. Upon return from that subroutine, C$ contains the character pressed. The program refuses to accept a space (S1$) or any character not contained in the list CL$. Lines 1710-1750 determine whether C$ is in the list CL$.

String Search

The following subroutine searches for one string inside another. The subroutine is called by several other parts of the program.

```
1770 QF=0
1780 IF Q0+LEN(Q2$)-1>LEN(Q1$) THEN RETURN
1790 IF MID$(Q1$,Q0,LEN(Q2$))=Q2$ THEN 1820
1800 Q0=Q0+1
1810 GOTO 1780
1820 QF=Q0
1830 RETURN
```

On entry to the subroutine, ZA$ is the string to be searched, ZB$ is the string to search for, and Q0 is the starting position for the search. Upon return from the subroutine, QF is the position at which ZB$ is found in ZA$. Q0=0 indicates the string is not found.

Midstring Replacement Subroutine

This subroutine replaces a portion of a string. It is used to blot out characters from the letters-available string, ZA$=LA$. Each time the player guesses a letter from LA$, this subroutine replaces that letter in LA$ with the character ZB$=S1$.

```
1840 ZC$=NU$
1850 IF ZP=1 THEN 1870
1860 ZC$=LEFT$(ZA$,ZP-1)
1870 ZC$=ZC$+ZB$
1880 IF LEN(ZA$)-LEN(ZB$)-ZP+1=0 THEN 1900
1890 ZC$=ZC$+RIGHT$(ZA$,LEN(ZA$)-LEN(ZB$)-ZP+1)
1900 ZA$=ZC$
1910 RETURN
```

On entry to the subroutine, ZP is the starting position for the replacement, ZA$ is the string to be modified, and ZB$ is the new contents to be plugged into a portion of ZA$. On return from the subroutine, ZA$ has the same length as it did initially, but a portion of its contents starting at position ZP are replaced by the contents of ZB$.

Repeating Characters

Here's the subroutine that builds up a string of repeating characters:

```
1920 SO$=NU$
1922 FOR K=1 TO LC
1924 SO$=SO$+RC$
1926 NEXT K
1928 RETURN
```

RC$ is the character to be repeated, LC is the length of the string, and SO$ is the resultant string.

Cursor Positioning

The next two lines move the cursor to any character position on the screen:

```
1930 PRINT HO$;LEFT$(CD$,C1);LEFT$(CR$,C2);
1932 RETURN
```

On entry to the subroutine, C1 and C2 are set to the destination row and column locations. C1 can range from 0 to 23 and C2 from 0 to 39. In line 1930, HO$ homes the cursor to the upper-left corner of the screen; CD$ and CR$ move the cursor down to the specified row and right to the specified column.

Drawing the Panic Meter

These lines draw the panic meter with a panic level of 0 (no errors made yet):

```
1934 C1=BA-1
1936 C2=INT((VW-11)/2)
1938 GOSUB 1930
1940 PRINT RD$;"PANIC METER";
1942 C1=BA
1944 C2=BB
1946 GOSUB 1930
1948 PRINT BX$;TC$;
1966 RETURN
```

First the program prints a label, PANIC METER, over the meter. C1 is set to the row just over the box, and C2 is set to the starting column for the label. Lines 1946-1948 print the meter (referred to in the program's remarks as a box).

Setting the Panic Level

Whenever the player guesses an incorrect letter, the following routine increases the panic meter reading:

```
1968 C1=BA
1970 C2=BB+(F-1)*3
1972 GOSUB 1930
1974 PRINT RD$;RV$(1);LEFT$(SS$,3);
1976 PRINT CHR$(17);LEFT$(CF$,3);
1978 PRINT LEFT$(SS$,3);RV$(2);TC$;
1980 RETURN
```

The panic meter occupies two rows and NF × 3 columns on the display. Line 1974 fills the appropriate portion of the top row, and line 1978 fills in the appropriate portion of the bottom row.

The Data

The next lines give the title of the word list, the number of words it contains, and the words themselves.

```
2050 DATA ELEMENTS
2060 DATA 10
2070 DATA HYDROGEN,HELIUM,OXYGEN,NITROGEN
2080 DATA CARBON,CHLORINE,SODIUM,FLUORINE,
     NEON,ARGON
```

—Hints for Using the Program

Figure 13-1 shows a sample use of the program.

Guess My Word may be used for spelling practice or for vocabulary building. For spelling practice, select difficult words that exemplify a group of phonetic rules you want to learn. For vocabulary building, select a group of unfamiliar words relating to a single topic.

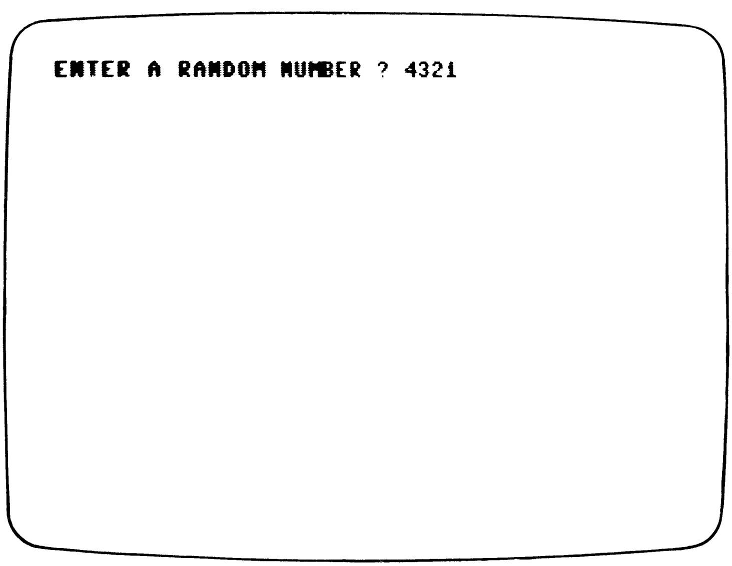
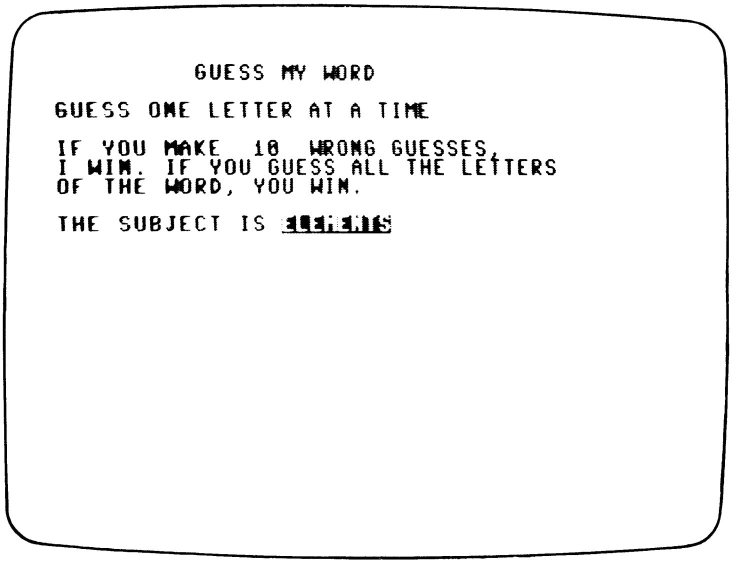


Figure 13-1. Sample run of Guess My Word



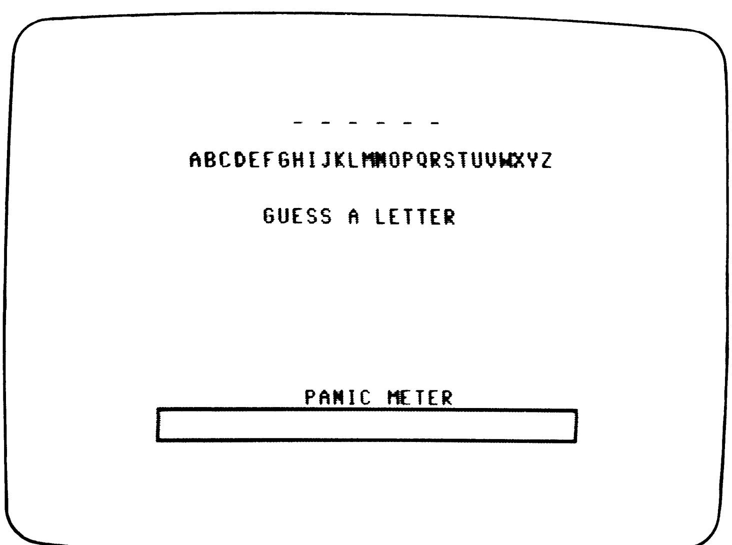
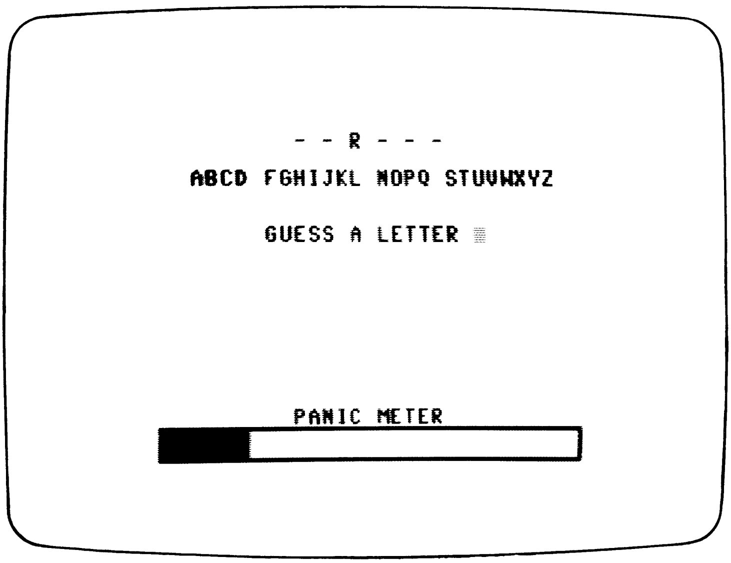


Figure 13-1. Sample run of Guess My Word *(continued)*



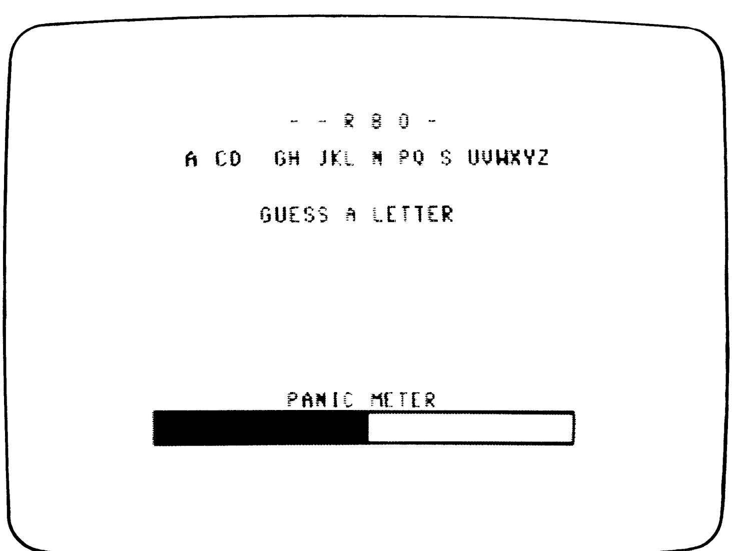
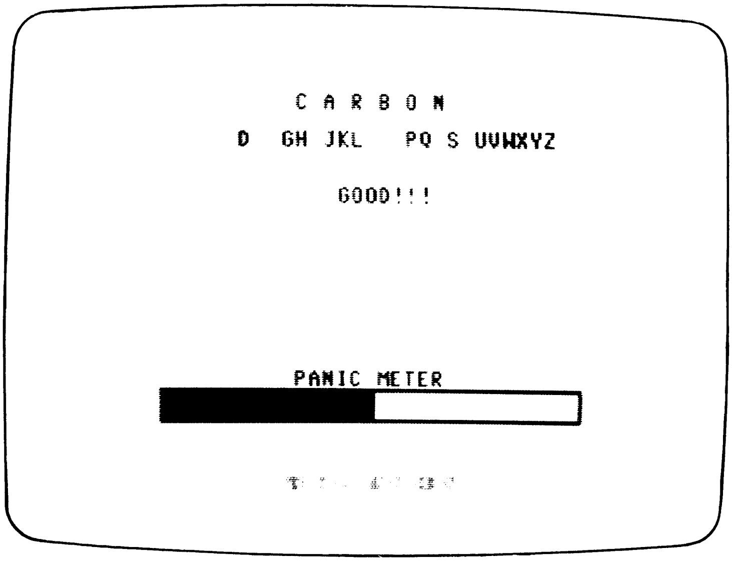


Figure 13-1. Sample run of Guess My Word *(continued)*



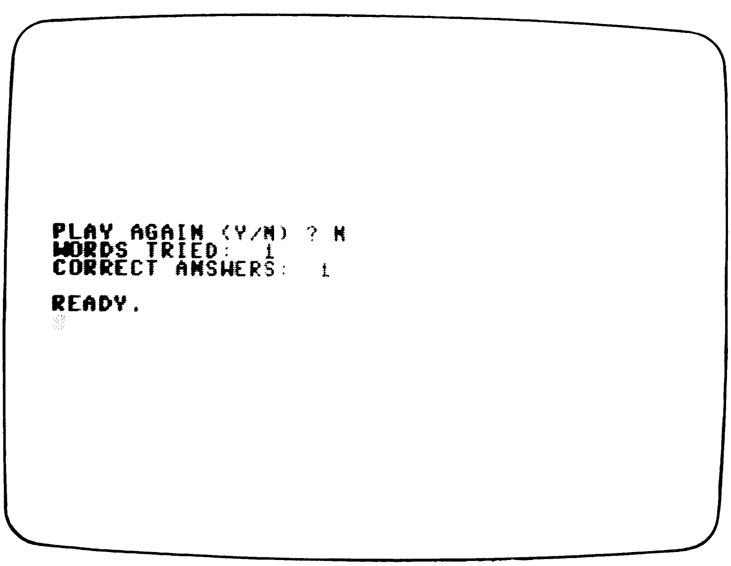


Figure 13-1. Sample run of Guess My Word *(continued)*

Be sure to set up the word list in the order shown: the title, the word-count, and then the list of words. Begin each numbered data line with DATA, and do not put any spaces inside the words.

Two individuals can play a game using this program by taking turns guessing letters. A player gets one point for each letter guessed correctly. When a guessed letter occurs more than once in a word, the player gets an extra point for each extra occurrence.

Chapter 14

Poetry Generator

A computer 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 generate randomly selected words that fall into a grammatical skeleton. The result will sometimes pass for a real poem, while other times the computer outputs a silly but entertaining parody of a poem.

The vocabulary and poem structure that you provide have everything to do with the quality of the final result. While randomly chosen vocabularies produce interesting and surprising results, adding more structure gives the poems greater coherence.

The Poetry Generator program is quite simple; it makes no pretensions to having artificial intelligence. However, if you put some thought into compiling the vocabulary and designing the formats, you can count on getting some amusing and even amazing results.

To illustrate, the favorite words and verse formats of three poets were fed into the Poetry Generator. Figure 14-1 shows the results; the poems were edited for obvious grammatical errors.

How the Poetry Generator Works

Two data structures determine the type of poems produced: the vocabulary, stored at the end of the program, and the poem formats, entered from the keyboard when you run the program.

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 14-1. The Poetry Generator produced these verses using the words and formats of William Shakespeare, Emily Dickinson, and Robert Frost

Along with each word you include in the vocabulary, you must indicate the part of speech using the eight category codes listed in Table 14-1. For example, RED should be identified as category 2 (adjective), FALLS as category 5 (intransitive verb), and HITS as category 6 (transitive verb).

Creating a poem format is quite simple. First make up a sample poem. Then replace each variable word (each word that you want the program to fill in) with the appropriate code. Leave the other words and punctuation as they are. As an example, suppose you take the following impressionistic triplet as a model verse.

Table 14-1. Word Categories Used in the Poetry Generator

Category (example)	Code Number
Noun (mountain)	1
Adjective (frosty)	2
Adverb (happily)	3
Preposition (into)	4
Intransitive verb (remain)	5
Transitive verb (take)	6
Subordinate conjunction (if)	7
Coordinate conjunction (and)	8

THE DEWDROP HANGS FROM A TWIG
IN LATE WINTER —
A WINDOW INTO SPRING

The grammatical skeleton for that verse is

THE *noun intransitive verb preposition* A *noun*
preposition adjective noun —
A *noun preposition noun*

The corresponding poem format is

THE 1 5 4 A 1
4 2 1 —
A 1 4 1

We simply substitute a code number for each italicized word. Notice that we include the articles (THE and A) and the dash as fixed elements; they will appear "as is" in every random poem produced according to this format.

One further detail about poem format. To tell the program to end a line, you include the special code 9. With this in mind, the actual poem format you would specify to the program is

THE 1 5 4 A 194 2 1 —9A 1 4 19

When the program is reading a poem format, it replaces each number 1 to 8 with a randomly chosen word from the corresponding category. Each time the program encounters a 9, it ends the line and starts a new one. Any other characters in the poem format remain in their places in the final poem.

The Poetry Generator does not check for subject and verb agreement or the proper spelling of inflected words. For example, if your vocabulary includes verbs in the third person singular and your format includes a plural subject, you may end up with results like

THE GLUM BULL AND THE BLUE MOON
STALKS THE REBELLIOUS HIGHWAY

Accepting such minor imperfections keeps the program short and simple. Don't hesitate to edit the poems for grammatical correctness. After all, even real poets occasionally need a little help.

—The Program

The first block sets certain C-64 features and resets the random number generator so that a different series of poems is produced each time you run the program.

```
10 POKE 53280,1
20 POKE 53281,1
22 PD=4: REM DEVICE NUMBER OF PRINTER
24 DV=1: REM DEFAULT DEVICE=TV
26 MP=1: REM DEFAULT NUMBER OF POEMS
28 NU$="": REM NO SPACES INSIDE QUOTES
30 INPUT "ENTER A RANDOM NUMBER ";R
40 R=RND(-ABS(R))
50 PRINT "ONE MOMENT..."
```

If your printer has a device number other than 4, change line 22 accordingly.

The next lines read the vocabulary list, which is stored in DATA statements at the end of the program. You can include any number of words in the vocabulary as long as the last word is followed by a slash (/). The program achieves this flexibility by reading the word list twice: once to see how many words are in each category and a second time to put the words into the appropriate data structures.

Surveying the Word List

The following lines make the first pass through the vocabulary list:

```
60 DIM N(8)
70 READ W$
80 IF W$="/" THEN 120
90 READ T
100 N(T)=N(T)+1
110 GOTO 70
120 DIM W1$(N(1)),W2$(N(2)),W3$(N(3)),W4$(N(4))
130 DIM W5$(N(5)),W6$(N(6)),W7$(N(7)),W8$(N(8))
```

Line 70 reads each word and line 90 reads the corresponding category code.

The array N( ) stores the total number of words in each category. For instance, N(1) is the number of nouns (category 1). The program continues reading words until it encounters the end-of-data marker, a slash (/). You must end the word list with this symbol.

After reading all the words, the program creates separate arrays for each type of word. In lines 120 and 130, each array is dimensioned to hold the number of words in the corresponding category. For instance, W1$( ) is designed to hold the the N(1) nouns that your vocabulary list contains.

Reading in the Vocabulary

The next block of lines rereads the vocabulary list, this time putting each word into the appropriate array.

```
140 RESTORE
150 READ W$
160 IF W$="/" THEN 430
170 READ T
180 ON T GOTO 190,220,250,280,310,340,370,400
190 K1=K1+1
200 W1$(K1)=W$
210 GOTO 150
220 K2=K2+1
230 W2$(K2)=W$
240 GOTO 150
250 K3=K3+1
260 W3$(K3)=W$
270 GOTO 150
280 K4=K4+1
```

```
290 W4$(K4)=W$
300 GOTO 150
310 K5=K5+1
320 W5$(K5)=W$
330 GOTO 150
340 K6=K6+1
350 W6$(K6)=W$
360 GOTO 150
370 K7=K7+1
380 W7$(K7)=W$
390 GOTO 150
400 K8=K8+1
410 W8$(K8)=W$
420 GOTO 150
```

Line 150 reads the word W$, and line 170 reads its corresponding category number T. Depending on the value of T, line 180 selects the appropriate logic to put W$ into the correct array. The counter variables K1, K2, and so on ensure that words are added to successive array locations within each category. By the time the program reads the end-of-data marker in line 160, all the words have been placed into the appropriate arrays.

The Menu

The next lines print a title and instructions and prompt you to enter the poem format.

```
430 NL$=CHR$(13)
440 F$="POEM 9 9 2 1 4 A 2 1 9 THE 1 3 5.99
450 PRINT CHR$(147)
460 PRINT SPC(13)"THE C-64 POET"
470 PRINT
480 PRINT "FORMAT CODES:"
490 PRINT "  1-NOUN  2-ADJECTIVE"
500 PRINT "  3-ADVERB  4-PREPOSITION"
510 PRINT "  5-INTRANSITIVE VERB"
520 PRINT "  6-TRANSITIVE VERB"
530 PRINT "  7-SUBORDINATE CONJUNCTION"
540 PRINT "  8-CONJUNCTION"
550 PRINT "  9-NEW LINE"
560 PRINT "  ALL OTHER CHARACTERS ARE USED AS-IS"
570 PRINT "THE CURRENT FORMAT IS"
580 PRINT CHR$(18); F$; CHR$(146)
590 PRINT "ENTER A NEW FORMAT OR PRESS <RETURN>"
600 F1$=""
610 INPUT F1$
```

```
620 IF F1$<>NU$ THEN F$=F1$
630 INPUT "HOW MANY POEMS ";MP
640 PRINT "OUTPUT TO: 1-TV  2-PRINTER"
650 INPUT "SELECT 1 OR 2 ";DV
660 IF DV<>1 AND DV<>2 THEN 640
670 IF DV=1 THEN 680
672 OPEN 1,PD
674 CMD 1
```

Line 430 stores the control code for a new line (a carriage return, ASCII 13). Line 440 assigns an initial value to the poem format.

Lines 570-590 print the current format. Line 600 prompts you to enter a new format line or press RETURN, which leaves the existing format line.

Line 630 asks you to specify the number of poems MP to be generated; each poem will be different (except for random coincidences).

Lines 640-674 let you specify what output device to use for the poem.

Poetry Generation Logic

The following block of lines generates MP poems using the poem format F$:

```
680 FOR J=1 TO MP
690 FOR K=1 TO LEN(F$)
700 S$=MID$(F$,K,1)
710 IF S$>="1" AND S$<="9" THEN 740
720 OW$=S$
730 GOTO 920
740 ON VAL(S$) GOTO 750,770,790,810,830,
    850,870,890,910
750 OW$=W1$(INT(RND(1)*N(1))+1)
760 GOTO 920
770 OW$=W2$(INT(RND(1)*N(2))+1)
780 GOTO 920
790 OW$=W3$(INT(RND(1)*N(3))+1)
800 GOTO 920
810 OW$=W4$(INT(RND(1)*N(4))+1)
820 GOTO 920
830 OW$=W5$(INT(RND(1)*N(5))+1)
840 GOTO 920
850 OW$=W6$(INT(RND(1)*N(6))+1)
860 GOTO 920
870 OW$=W7$(INT(RND(1)*N(7))+1)
880 GOTO 920
890 OW$=W8$(INT(RND(1)*N(8))+1)
900 GOTO 920
```

```
910 OW$=NL$
920 PRINT OW$;
930 NEXT K
940 PRINT
950 NEXT J
```

Lines 680-950 constitute a repetitive procedure or "loop." During each pass through the loop, the program produces one poem. The larger loop contains a smaller one: lines 690-930. This smaller loop examines each character of the format and takes appropriate action depending on whether the character is a category number, an end-of-line code, or a literal.

Here's a summary of the logic that evaluates each character S$ of the poem format F$:

1. If the character is a category code, select a word at random from the appropriate category, store that word in OW$, and go to Step 4.
2. If the character is the end-of-line code, store NL$ in OW$, and go to Step 4.
3. Otherwise, store the character in OW$.
4. Print OW$.

The variable S$ holds the character of the format that is currently under examination. Line 710 determines whether S$ is a one of the preset codes.

If it is not either of those, the character is treated as a literal and is immediately assigned to the output variable OW$ (line 720). If S$ is a category code from 1 to 8 or the end-of-line code 9, line 740 selects the appropriate logic for each specific category.

Consider the case of S$="1" as an example. All eight categories are handled similarly.

Lines 750 and 760 handle the case of S$="1", which ndicates a noun. Recall that N(1) is the number of words in category 1. Accordingly, line 750 gets a random number from 1 to N(1) and uses that number as a pointer to one of the words in W1$( ). The randomly chosen word is stored in OW$, and the program jumps to line 920, which prints OW$.

Line 930 causes the program to loop back for the next character of the format until all of its characters have been handled.

Displaying the Continuation Menu

After all the poems have been printed, the following lines print a continuation menu:

```
960 IF DV=1 THEN 980
970 PRINT#1,
975 CLOSE 1
980 PRINT "CONTINUE OR QUIT"
990 CQ$="C"
995 INPUT "TYPE C OR Q ";CQ$
1000 IF CQ$="C" THEN 450
1010 IF CQ$<>"Q" THEN 980
1020 END
```

The Data

Store your vocabulary list in DATA lines starting with 1030. For the sake of testing the program, use this special list:

```
1030 DATA NOUN,1,ADJ,2,ADV,3,PREP,4
1040 DATA BE-VERB,5,DO-VERB,6,SUB.CONJ.,7,CONJ.,8
1050 DATA /
```

When using this list, the program should print NOUN whenever the format calls for a noun, ADJ whenever the format calls for an adjective, and so forth.

After running the program with this test list, type in the vocabulary list given in Figure 14-2.

```
1030 REM 66 NOUNS
1040 DATA RIVER, 1, WATER, 1, POOL, 1, MIRROR, 1,
     SCENE, 1, BUBBLE, 1
1050 DATA FALL, 1, YEAR, 1, MAZE, 1, DANCE, 1,
     FLIGHT, 1, PICTURE, 1
1060 DATA STREAM, 1, HEART, 1, MIND, 1, WATERFALL,
     1, BED, 1, COURSE, 1, SHADOW, 1
1070 DATA FORM, 1, IMAGE, 1, SCREEN, 1, CHILD, 1,
     GARDEN, 1, STRAND, 1
1080 DATA PEBBLE, 1, SAND, 1, FLOWER, 1, MOTHER,
     1, TIME, 1, SPOT, 1
1090 DATA IMAGINATION, 1, LIFE, 1, STONE, 1,
     BOWER, 1, SUMMER, 1, MEADOW, 1
```

Figure 14-2. Sample vocabulary for the Poetry Generator

```
1100 DATA THUNDERSTORM, 1, GRASSHOPPER, 1,
     CYCLONE, 1, ROOT, 1, WOOL, 1
1110 DATA WILDERNESS, 1, NIGHT, 1, BRIDE, 1, BODY,
     1, SPRING, 1, SEED, 1, MILK, 1
1120 DATA SURFACE, 1, THICKET, 1, ARROW, 1,
     MANTLE, 1, WILDERNESS, 1
1130 DATA SUNLIGHT, 1, SAND-DUNE, 1, TRAIN, 1,
     CLOUD, 1, RAIN, 1
1140 DATA KEY, 1, WINDOW, 1, TREE, 1, MUSIC, 1,
     SNOW, 1, MOUNTAIN, 1, FEATHER, 1
1150 DATA VOICE, 1, TWILIGHT, 1, EARTH, 1, DOOM,
     1, ACCEPTANCE, 1, TIME, 1, TRUTH, 1,
     PATIENCE, 1, FACT, 1, FEAR, 1, SILENCE, 1,
     BIRD, 1, YEAR, 1, WHISPER, 1, FEAT, 1, HOPE, 1
1160 DATA INNUMERABLE, 2, IRREVOCABLE, 2
     MICROSCOPIC, 2, PURE, 2, FAIL, 5, UNDAUNTED, 2
     DIMINUTIVE, 2, SINGLE, 2, MERE, 2
1170 DATA FAITHFULLY, 3, MARVELLOUSLY, 3, ALWAYS, 3,
     HEARS, 6, SUBTRACTS, 6, RECEIVES, 6
1180 DATA DARES, 6, FAILS, 5, FORSAKES, 6, AT, 4,
     OF, 4, BY, 4, ABOVE, 4, UNDER, 4, FROM, 4,
     AND, 8, BEFORE, 7
1190 REM 17 INTRANSITIVE VERBS
1200 DATA REVOLVES, 5, BREAKS, 5, WATCHES, 5,
     SCREAMS, 5, FADES, 5
1210 DATA FLOWS, 5, TALKS, 5, RETURNS, 5, RUNS, 5,
     EXISTS, 5, NODS, 5, LIVES, 5
1220 DATA REMEMBERS, 6, REMEMBERS, 5, REMAINS, 5,
     WONDERS, 5, VANISHES, 5, WISHES, 5
1230 REM 15 PREPOSITIONS
1240 DATA IN, 4, ON, 4, BESIDE, 4, WITH, 4, FROM, 4,
     TO, 4, OVER, 4, UNDER, 4, BY, 4
1250 REM 5 CONJUNCTIONS
1260 DATA AND, 8, OR, 8, BUT, 7, WHILE, 7,
     BECAUSE, 7
1270 REM 31 ADJECTIVES
1280 DATA TURNING, 2, DARK, 2, SUSPENDED, 2,
     UNCHANGING, 2, FRAIL, 2,
1290 DATA RIPPLING, 2, SAME, 2, EACH, 2, FORMER, 2,
     REFORMING, 2, LOW, 2
1300 DATA ROCKY, 2, INTANGIBLE, 2, GLASSY, 2,
     SHIMMERING, 2, SECRET, 2
1310 DATA PAST, 2, RED, 2, YELLOW, 2, ALONE, 2,
     LITTLE, 2
```

Figure 14-2. Sample vocabulary for the Poetry Generator *(continued)*

```
1320 DATA CRACKED, 2, DARKENED, 2, FADED, 2,
     VARNISHED, 2, FOREIGN, 2
1330 DATA WHITE, 2, WILD, 2, CEASELESS, 2, GRAY, 2,
     AGELESS, 2
1340 REM 15 TRANSITIVE VERBS
1350 DATA CHASES, 6, DECEIVES, 6, IGNORES, 6,
     SAVES, 6, MAKES, 6, GIVES, 6
1360 DATA SEVERS, 6, OPENS, 6, CLOSES, 6, SEES, 6,
     INSTRUCTS, 6, STRIKES, 6
1370 DATA MARKS, 6, FILTERS, 6, PASSES, 6
1380 REM 10 ADVERBS
1390 DATA ONCE, 3, TWICE, 3, NEVER, 3, STILL, 3,
     ONCE, 3, AGAIN, 3, SHYLY, 3
1400 DATA FAST, 3, EVER, 3, NEVER, 3, HAPPILY, 3,
     SILENTLY, 3
1410 DATA BALL, 1, HAT, 1, RED, 2, BLUE, 2, FAST, 3,
     SLOWLY, 3
1420 DATA UNHAPPILY, 3, HITS, 6, EATS, 6, DRINKS, 6,
     LISTENS, 5, IN, 4
1430 DATA ON, 4, OFF, 4, AND, 7, OR, 7, BUT, 7,
     WHILE, 8, BECAUSE, 8
1440 DATA /
```

Figure 14-2. Sample vocabulary for the Poetry Generator *(continued)*

Figure 14-3 shows a sample run of the program using this vocabulary.

—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 into DATA lines starting with line 1030. Remember that the last line of your vocabulary list must be

line number DATA /

substituting an appropriate line number for the italicized words.

Experiment with various formats. Try including prefixes, suffixes,

```
                THE C-64 POET

FORMAT CODES:
  1-NOUN  2-ADJECTIVE
  3-ADVERB  4-PREPOSITION
  5-INTRANSITIVE VERB
  6-TRANSITIVE VERB
  7-SUBORDINATE CONJUNCTION
  8-CONJUNCTION
  9-NEW LINE
  ALL OTHER CHARACTERS ARE USED AS-IS
THE CURRENT FORMAT IS
[illegible]
ENTER A NEW FORMAT OR PRESS <RETURN>
"99A THOUGHT, BY THE COMMODORE99IF THE 2 1 5 3,
   9WHO 6 THE 2 AND 2 1?999"
HOW MANY POEMS  3
OUTPUT TO: 1-TV  2-PRINTER
SELECT 1 OR 2  1


A THOUGHT, BY THE COMMODORE

IF THE RIPPLING SCREEN RETURNS SLYLY,
WHO CHASES THE MICROSCOPIC AND PURE WATER?




A THOUGHT, BY THE COMMODORE

IF THE FORMER WATERFALL NODS FAITHFULLY,
WHO DECEIVES THE MERE AND TURNING HEART?




A THOUGHT, BY THE COMMODORE

IF THE BLUE SHADOW WATCHES NEARLY,
WHO MARKS THE SHIMMERING AND DARK WATERFALL?
```

Figure 14-3. Sample run of the Poetry Generator

```
CONTINUE OR QUIT
TYPE C OR Q C


                THE C-64 POET

FORMAT CODES:
  1-NOUN  2-ADJECTIVE
  3-ADVERB  4-PREPOSITION
  5-INTRANSITIVE VERB
  6-TRANSITIVE VERB
  7-SUBORDINATE CONJUNCTION
  8-CONJUNCTION
  9-NEW LINE
  ALL OTHER CHARACTERS ARE USED AS-IS
THE CURRENT FORMAT IS
[illegible]
ENTER A NEW FORMAT OR PRESS <RETURN>

HOW MANY POEMS  3
OUTPUT TO: 1-TV  2-PRINTER
SELECT 1 OR 2  1
POEM

 MICROSCOPIC BEACH IN A BLUE GARDEN
 THE HAT SILENTLY WATCHES.


POEM

 BLUE WATER FROM A ROCKY MIND
 THE SCREEN FAITHFULLY NODS.


POEM

 ROCKY MAZE ON A BLUE SCREEN
 THE BALL TWICE EXISTS.


CONTINUE OR QUIT
TYPE C OR Q C
```

Figure 14-3. Sample run of the Poetry Generator (*continued*) and inflectional endings for special effects. For instance, the format fragment

6ING THE 1S OF YOUR 19

might generate

WALKING THE RIVERS OF YOUR MIND

On the other hand, it might equally well generate the less exciting

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 15

Electronic Loom

The Electronic Loom program turns your computer screen into a grid on which you can create colorful designs. You specify the length and width of the design (it must fit on your display), and the program creates patterns by combining any characters your computer can display and print using any of the eight primary C-64 colors. Common punctuation marks and other symbols work well for giving a rough approximation of real loom work.

If you're interested in weaving, you can use the program as a planning aid to visualize patterns before weaving them on a loom. You can associate some characters with specific weaving techniques. For example, a block of hyphens might represent a plain weave, a block of alternating hyphens and equal signs might represent a twill weave with its characteristic diagonal pattern, and a repeating pattern might represent a satin weave (see Figure 15-1).

You can also use the program to create any number of designs—flags, cartoons, and so forth.

The best way to understand the program's operation is to look at its menu.

```
Plain

----------------------
----------------------
----------------------
----------------------
----------------------
----------------------

Twill

-=-=-=-=-=-=-=-=-=-=-=
=-=-=-=-=-=-=-=-=-=-=-
-=-=-=-=-=-=-=-=-=-=-=
=-=-=-=-=-=-=-=-=-=-=-
-=-=-=-=-=-=-=-=-=-=-=
=-=-=-=-=-=-=-=-=-=-=-

Satin

=====-=====-=====-====
-=====-=====-=====-===
=-=====-=====-=====-==
==-=====-=====-=====-=
===-=====-=====-=====-
====-=====-=====-=====
```

Figure 15-1. Sample weave patterns

THE ELECTRONIC LOOM

1 – VIEW

2 – PRINT

3 – CHANGE A ROW

4 – CHANGE A COLUMN

5 – COPY A ROW

6 – COPY A COLUMN

7 – COPY A BLOCK

8 – FILL

9 – END

The first option shows the latest version of the design. When you start, the design is filled with hyphens. The second option lets you print the current design. The third and fourth options let you change the contents of a single row or column. Changing a row is comparable to replacing one horizontal thread with another. Changing a column, however, is comparable to replacing a vertical thread—something you can't do on a real loom.

The fifth and sixth options let you copy one row or column to another row or column. The seventh option copies a block (a rectangle). This is especially handy when you've created a picture or a pattern inside the space of several rows and columns. You can duplicate the picture by copying the rectangle that contains it to another part of the grid. The eighth option lets you fill all areas of the design with a single character. Option nine ends the program.

—The Program

The first block initializes the display and certain other constants:

```
1 TX$=CHR$(5)
2 CS$=CHR$(147)
3 POKE 53280,0
4 POKE 53281,0
5 S1$=" ": REM 1 SPACE INSIDE QUOTES
6 NU$="": REM NO SPACES INSIDE QUOTES
7 RV$=CHR$(18)
8 NR$=CHR$(146)
9 S9$=CHR$(8): S6$=CHR$(15): REM PRINTER
  LINE SPACING
10 DV=4: REM PRINTER DEVICE NUMBER
20 VS=25: REM DISPLAY SIZE
30 VL=VS-2: REM USABLE DISPLAY LENGTH
40 VW=40: REM DISPLAY WIDTH
50 TB$=S1$+"-=+><X."
60 NC=LEN(TB$)
70 IL=INT(VW/8)
80 CL$=NU$
82 FOR C=1 TO 8
84 READ CL
86 CL$=CL$+CHR$(CL)
88 NEXT C
90 DATA 144,5,28,159,156,30,31,158
```

Line 9 sets the line-spacing codes used for the Commodore MPS-801 printer. If you have a different printer, see "Using the Program" at the end of this chapter. Line 10 sets the printer device number. If your printer has a different device number, change line 10 accordingly. Line 20 sets the display size to 25, indicating the display holds 24 lines. Line 40 sets the display width to 40.

Line 50 stores in TB$ all the characters that may be used to make up the loom design. You may change the characters; simply include your selections inside the quotes. It's a good idea to have a space as one of the characters; using the blank space, you'll be able to erase parts of the design. Include as many characters as you wish inside the quotes; however, 15 characters is probably the most you would need. Line 70 calculates the number of 8-character fields that will fit on each display line.

Lines 80-90 store the eight color codes in CL$.

Inputting the Design Grid Size

The next block of lines prints a title and prompts you to specify the size of the design grid:

```
100 PRINT CS$;TX$;SPC((VW-19)/2);
102 TL$="THE ELECTRONIC LOOM"
104 FOR J=1 TO LEN(TL$)
105 L$=MID$(TL$,J,1)
106 CO$=MID$(CL$,INT(RND(1)*7)+2,1)
107 PRINT CO$;L$;
108 NEXT J
109 PRINT TX$;
110 PRINT
120 INPUT "HOW MANY ROWS? ";RX
125 IF RX<1 THEN 120
130 INPUT "HOW WIDE IS EACH ROW? ";WX
135 IF WX<1 THEN 130
140 IF WX<VW-2 THEN 170
150 PRINT "WIDTH MUST BE LESS THAN ";VW-2
160 GOTO 130
170 DIM D$(RX)
180 C$=TX$+"X"+TX$: REM INITIAL STITCH TYPE
190 GOSUB 570
```

Given a display width of VW, the maximum design width is VW−3. There is no logical limit to the number of rows because the program breaks the design into pages. However, if you specify too many rows, your computer will run out of memory when it tries to create the pattern array D$( ) in line 170. Each element of D$( ) corresponds to one row of the design.

Line 180 sets the initial grid-fill character to a white "X", and the subroutine called in line 190 fills D$( ) with the character in C$.

Displaying the Menu

The next lines print the main menu:

```
200 PRINT CS$;"TYPE IN THE NUMBER OF YOUR CHOICE"
210 PRINT
```

```
220 PRINT "1-VIEW"
230 PRINT "2-PRINT"
240 PRINT "3-CHANGE A ROW"
250 PRINT "4-CHANGE A COLUMN"
260 PRINT "5-COPY A ROW"
270 PRINT "6-COPY A COLUMN"
280 PRINT "7-COPY A BLOCK"
290 PRINT "8-FILL"
300 PRINT "9-END"
310 Q=0
312 INPUT Q
315 IF Q<1 OR Q>9 THEN 200
320 ON Q GOSUB 630,1230,790,1000,1290,1340,
    1450,340,380
330 GOTO 200
```

The nine options are treated as subroutines. Line 320 selects the appropriate subroutine depending on your selection of Q.

On completion of the subroutine you select, line 330 causes the program to jump back to the start of the main menu.

Filling the Grid

The next lines handle option 8 (fill the grid):

```
340 PRINT "SELECT THE FILL CHARACTER"
350 GOSUB 390
360 GOSUB 570
370 RETURN
380 END
```

The subroutine called in line 350 prints the list of available design characters and gets your selection. The subroutine called in line 360 fills D$( ) with the character you select. Line 370 returns to the main program. Line 380 ends the program (option 9).

Choosing a Design Character

Here's the subroutine that prints the design characters and accepts your selection. It is used during completion of several menu options:

```
390 FOR J=1 TO NC
395 NC$=RIGHT$(S1$+STR$(J),2)+S1$
400 PRINT NC$;
405 PRINT MID$(TB$,J,1);;
410 IF J-INT(J/IL)*IL=0 THEN 425
415 PRINT SPC(4);
```

```
420 GOTO 435
425 PRINT
430 PRINT
435 NEXT J
440 PRINT
445 PRINT "SELECT A CHARACTER (ENTER 1-";NC;") ";
450 INPUT J
455 IF J<1 OR J>NC THEN 390
460 C$=MID$(TB$,J,1)
465 FOR J=1 TO 8
470 NC$=RIGHT$(S1$+STR$(J),2)+S1$
475 PRINT NC$;
480 PRINT MID$(CL$,J,1);RV$;S1$;NR$;TX$;
485 IF J-INT(J/IL)*IL=0 THEN 500
490 PRINT SPC(4);
495 GOTO 510
500 PRINT
505 PRINT
510 NEXT J
515 PRINT
520 INPUT "SELECT A COLOR (ENTER 1-8) ";J
525 IF J<1 OR J>8 THEN 390
530 C$=MID$(CL$,J,1)+C$+TX$
560 RETURN
```

Line 390 counts from 1 to NC (the number of design characters available). For each character, lines 395 and 400 print the character number, and line 405 prints the corresponding character. Line 410 causes the program to skip to a new line after IL characters have been displayed on a single line.

After all NC characters have been displayed, lines 440-455 get your selection. Lines 460-530 get your color selection in a similar fashion. C$ contains a color code and the character followed by the text color code. Line 560 returns with your selection stored in C$.

The following lines fill the entire design with C$:

```
570 FOR J=1 TO RX
580 D$(J)=NU$
590 FOR K=1 TO WX
600 D$(J)=D$(J)+C$
610 NEXT K,J
620 RETURN
```

Line 580 sets D$(J) equal to an empty string, and line 600 repeatedly adds C$ to D$(J) until K characters have been added. This process is done for every row D$(J) of the design.

Viewing the Design

The next block of lines handles option 1 (view the design):

```
630 PRINT CS$;
640 GOSUB 1680
650 FOR J=1 TO RX
660 PRINT RV$;
680 PRINT RIGHT$(S1$+STR$(J),2);
690 PRINT NR$;
700 PRINT D$(J)
710 IF J-INT(J/VL)*VL>0 OR J=RX THEN 750
720 INPUT "PRESS <RETURN> FOR MORE ";Q$
730 PRINT CS$;
740 GOSUB 1680
750 NEXT J
760 INPUT "PRESS <RETURN> TO CONTINUE ";Q$
770 PRINTCS$;
780 RETURN
```

The subroutine called in line 640 prints a line of column headings to help you reference specific areas of the design. The loop from 650 to 750 is repeated once for each row of the design.

Line 700 prints the row. Line 710 checks if the current display page is full; if it is, line 720 prompts you to press RETURN for the next page of the design.

Changing a Row

Here's the logic for option 3 (change a row):

```
790 INPUT "SPECIFY THE ROW TO MODIFY ";R
800 IF R<1 OR R>RX THEN 790
810 GOSUB 1680
820 PRINT RV$;
830 PRINT RIGHT$(S1$+STR$(R),2);
840 PRINT NR$;
850 PRINT D$(R)
860 INPUT "ENTER FIRST COLUMN TO BE CHANGED ";C1
870 INPUT "ENTER LAST COLUMN TO BE CHANGED ";C2
880 IF C1<1 OR C1>C2 OR C2>WX THEN 850
890 GOSUB 390
900 F$=NU$
910 FOR J=1 TO C2-C1+1
920 F$=F$+C$
930 NEXT J
940 ZA$=D$(R)
```

```
950 ZB$=F$
960 ZP=(C1-1)*3+1
970 GOSUB 1810
980 D$(R)=ZA$
990 RETURN
```

After you specify the row to be modified, the program prints its current contents along with a column heading and row label.

The subroutine lets you insert a single character into one or more contiguous columns on the selected row. Lines 860 and 870 prompt you to specify the starting and ending columns, and the subroutine called in line 890 gets your character selection.

Lines 900-930 build F$, a string of character C$ that fills the column range specified. Lines 940-980 plug F$ into the row starting at column C1. Refer to line 960. Since each character occupies three columns (two color codes plus one character code), the actual starting location of the character is (C1−1)×3+1. The subroutine called in line 970 performs the actual modification of the row contents.

Changing a Column

The following lines handle option 4 (change a column):

```
1000 INPUT "SPECIFY THE COLUMN TO MODIFY ";C
1010 IF C<1 OR C>WX THEN 1000
1020 FOR J=1 TO RX
1030 PRINT RV$;
1040 PRINT RIGHT$(S1$+STR$(J),2);
1050 PRINT NR$;
1060 PRINT MID$(D$(J),(C-1)*3+1,3)
1070 IF J-INT(J/VL)*VL>0 OR J=RX THEN 1100
1080 INPUT "PRESS <RETURN> FOR MORE ";Q$
1090 PRINT CS$;
1100 NEXT J
1110 INPUT "ENTER THE FIRST ROW TO BE CHANGED ";R1
1120 INPUT "ENTER THE SECOND ROW TO BE CHANGED ";R2
1130 IF R1<1 OR R1>R2 OR R2>RX THEN 1110
1140 GOSUB 390
1150 FOR J=R1 TO R2
1160 ZA$=D$(J)
1170 ZB$=C$
1180 ZP=(C-1)*3+1
1190 GOSUB 1810
1200 D$(J)=ZA$
1210 NEXT J
1220 RETURN
```

The logic to modify a column is similar to that for modifying a row except that the program cannot address a column quite so simply. For each row R, D$(R) represents the row. In contrast, for each column C, the program must look at the Cth element of every row in the range specified.

Line 1000 gets the column number and lines 1110 and 1120 get the range of rows to be modified in that column. Lines 1150-1210 plug character C$ into the appropriate location of every row in the range specified. Again the apparent column number C must be adjusted to give the real column number: $(C1-1)\times3+1$. See line 1180.

Printing the Design

The following lines handle option 2 (print the design):

```
1230 OPEN 1,DV: REM OPEN PRINTER CHANNEL
1240 FOR J=1 TO RX
1250 PRINT#1,S6$;D$(J);S9$
1260 NEXT J
1270 PRINT#1,S6$: REM RESTORE NORMAL SPACING
1275 CLOSE 1
1280 RETURN
```

Line 1230 activates the printer. Line 1250 prints the contents of row J; the line is repeated once for each row in the design. S9$ causes the listing to be printed at nine lines per inch.

Copying a Row and Columns

The logic to copy one row (option 5) is quite straightforward:

```
1290 INPUT "SPECIFY THE SOURCE-ROW ";R1
1300 INPUT "SPECIFY THE DESTINATION-ROW ";R2
1310 IF R1<1 OR R1>RX OR R2<1 OR R2>RX THEN 1290
1320 D$(R2)=D$(R1)
1330 RETURN
```

After you specify the source row (the row to use as the original copy) and destination row (the row to be changed into a copy of the original), line 1320 makes the change.

The subroutine to copy a column (option 6) is a little more complex because of the difficulty of addressing a column:

```
1340 INPUT "SPECIFY THE SOURCE-COLUMN ";C1
1350 INPUT "SPECIFY THE DESTINATION-COLUMN ";C2
```

```
1360 IF C1<1 OR C1>WX OR C2<1 OR C2>WX THEN 1340
1370 FOR J=1 TO RX
1380 ZA$=D$(J)
1390 ZB$=MID$(D$(J),(C1-1)*3+1,3)
1400 ZP=(C2-1)*3+1
1410 GOSUB 1810
1420 D$(J)=ZA$
1430 NEXT J
1440 RETURN
```

The loop from line 1370 to line 1430 copies the Jth character of the source column into the Jth position of the destination column. The subroutine called in line 1410 performs the actual character replacement.

Copying a Block

Here's the logic for option 7 (copy a block):

```
1450 PRINT "SOURCE BLOCK:"
1460 PRINT " ENTER ROW & COL NO'S OF THE..."
1470 INPUT " UPPER LEFT CORNER (R1,C1) ";R1,C1
1480 INPUT " LOWER RIGHT CORNER (R2,C2) ";R2,C2
1490 IF R1<1 OR C1<1 OR R2<R1 OR C2<C1 OR C2>WX
     OR R2>RX THEN 1450
1500 PRINT
1510 PRINT "DESTINATION BLOCK:"
1520 PRINT " ENTER ROW & COL NO'S OF THE..."
1530 INPUT " UPPER LEFT CORNER (R3,C3) ";R3,C3
1540 RL=R2-R1
1550 RW=C2-C1
1560 IF R3<1 OR R3>RX OR C3<1 OR C3>WX THEN 1580
1570 IF R3+RL<=RX AND C3+RW<=WX THEN 1600
1580 PRINT "COPY WOULD EXCEED LOOM BOUNDARRIES"
1590 GOTO 1450
1600 FOR J=0 TO RL
1610 ZA$=D$(R3+J)
1620 ZB$=MID$(D$(R1+J),(C1-1)*3+1,(RW+1)*3)
1630 ZP=(C3-1)*3+1
1640 GOSUB 1810
1650 D$(R3+J)=ZA$
1660 NEXT J
1670 RETURN
```

Lines 1460-1530 prompt you to specify the details of the block-copy information. First you define the source block by locating the upper-left corner and lower-right corner (lines 1470-1490). Then you define the upper-right corner of the destination block; the program assumes that the destination block is the same size as the source block.

Lines 1540-1590 ensure that the source and destination blocks are within the boundaries of the loom. The loop from 1600 to 1660 copies the specified block, one row at a time.

Auxiliary Subroutines

Here's the subroutine to print column numbers over the grid:

```
1680 PRINT RV$;
1690 PRINT SPC(2);
1700 FOR C=1 TO WX
1710 CC=C-INT(C/10)*10
1720 IF CC=5 OR CC=0 THEN 1760
1730 PRINT ".";
1740 GOTO 1770
1760 PRINT RIGHT$(STR$(CC),1);
1770 NEXT C
1790 PRINTNR$
1800 RETURN
```

The heading consists of a dot for every column except for column numbers ending in 5 or 0. In these cases, the program puts in a 5 or 0. Line 1710 calculates CC, the column number modulo 10 (the remainder after integer division of the column number by 10). When CC=5 or CC=0, the program prints a 5 or a 0.

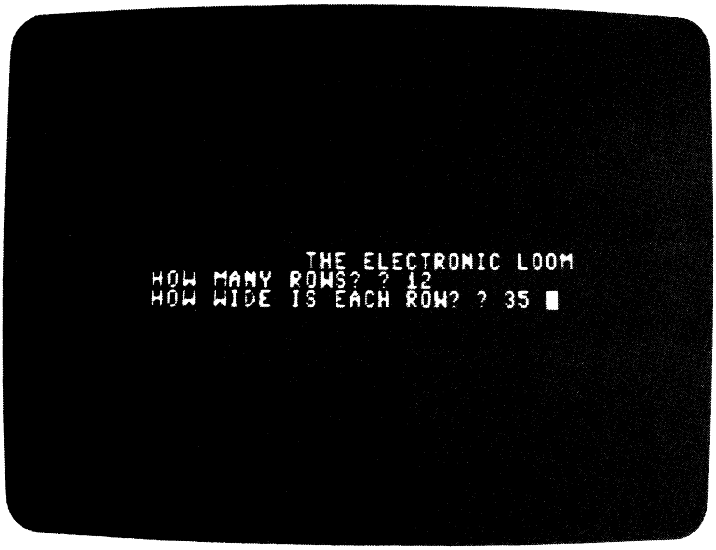
The final subroutine replaces a portion of a string with the contents of another:

```
1810 ZC$=NU$
1820 IF ZP=1 THEN 1840
1830 ZC$=LEFT$(ZA$,ZP-1)
1840 ZC$=ZC$+ZB$
1850 IF LEN(ZA$)-LEN(ZB$)-ZP+1=0 THEN 1870
1860 ZC$=ZC$+RIGHT$(ZA$,LEN(ZA$)-LEN(ZB$)-ZP+1)
1870 ZA$=ZC$
1880 RETURN
```

On entry to the subroutine, ZP is the position for the replacement, ZA$ is the string to be changed, and ZB$ is the string to be plugged into ZA$. On return from the subroutine, ZA$ contains ZB$ starting at position ZP.

—Using the Program

Figure 15-2 shows a few steps in a sample use of the program. The figure illustrates several tricks in using the program that might not be



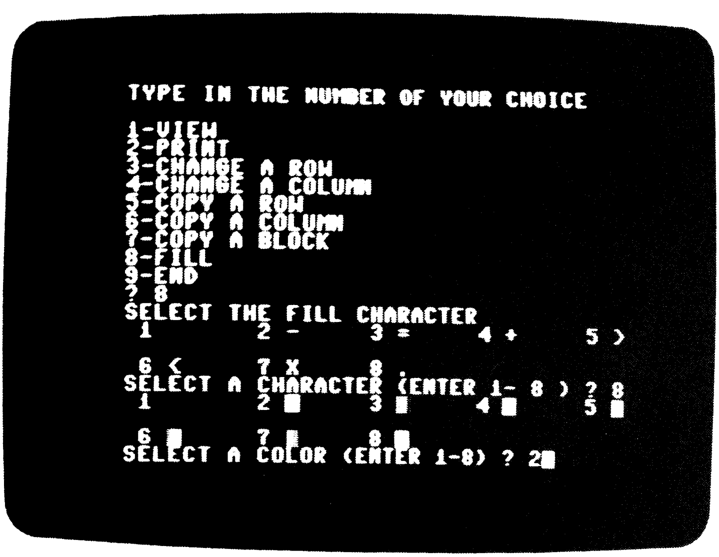
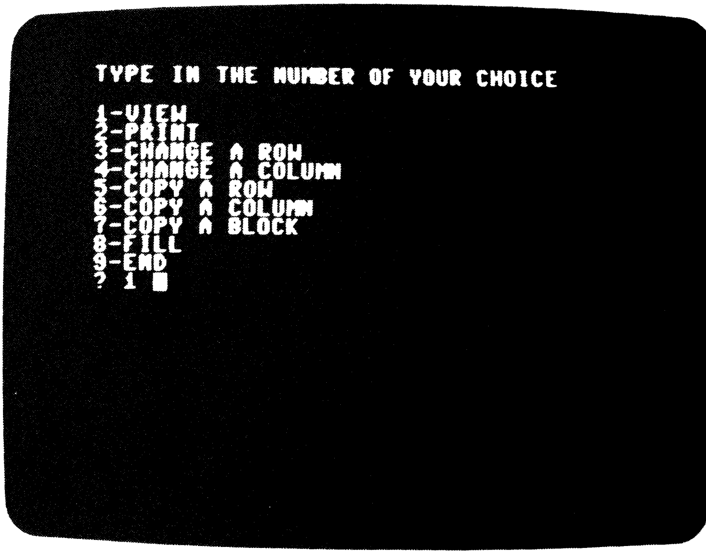


Figure 15-2. Sample use of the Electronic Loom



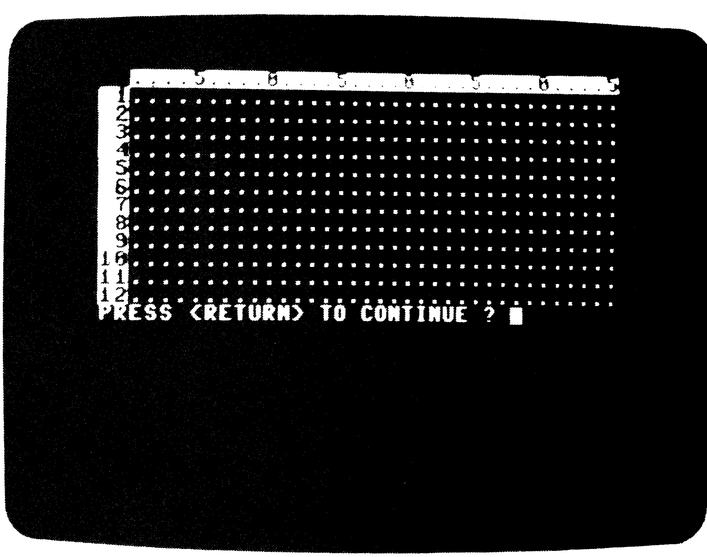
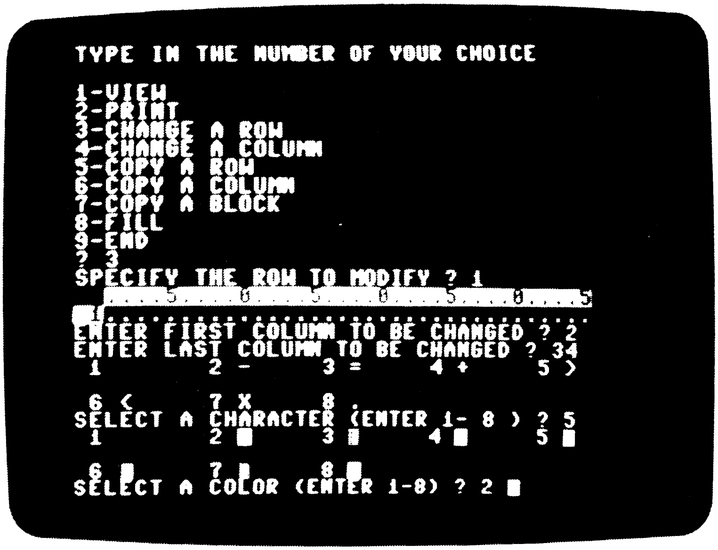


Figure 15-2. Sample use of the Electronic Loom (*continued*)



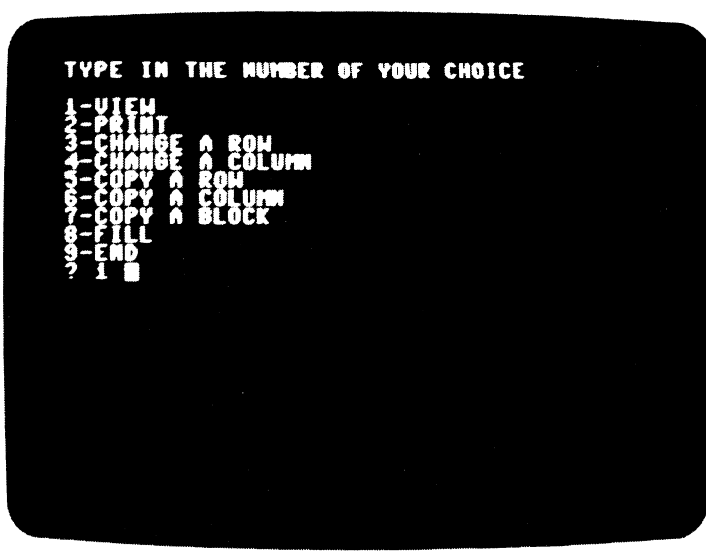
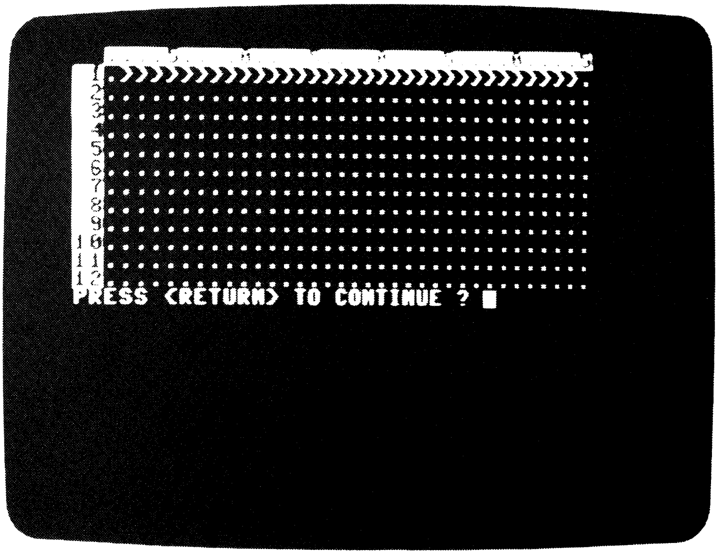


Figure 15-2. Sample use of the Electronic Loom (*continued*)



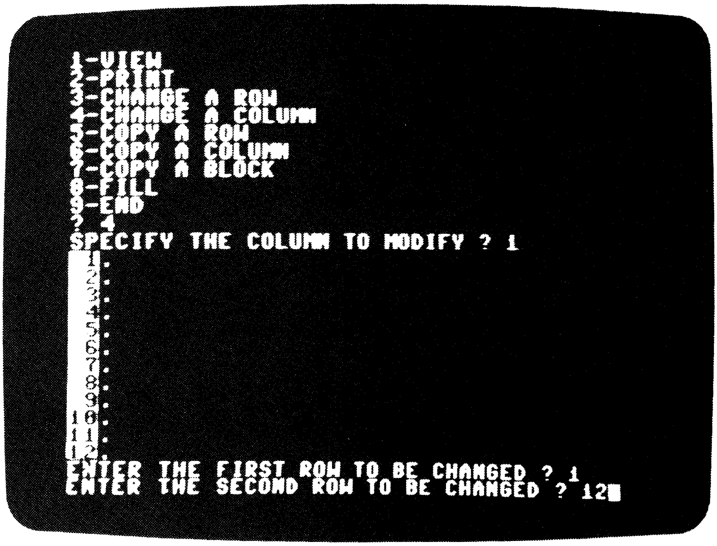
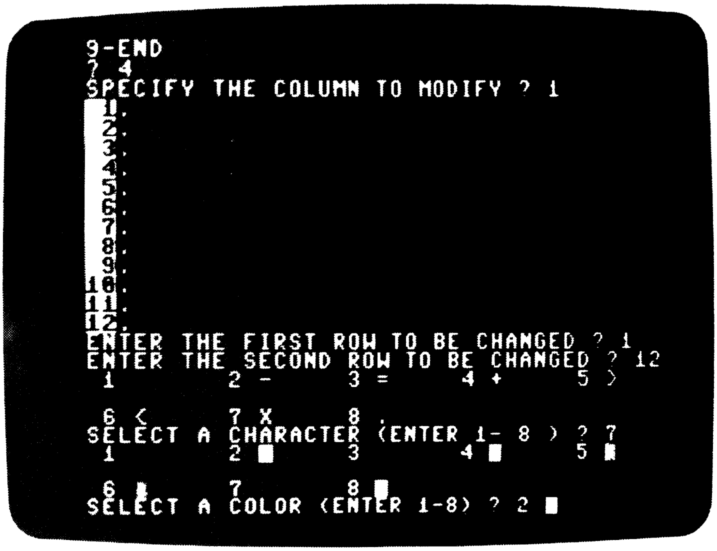


Figure 15-2. Sample use of the Electronic Loom (*continued*)



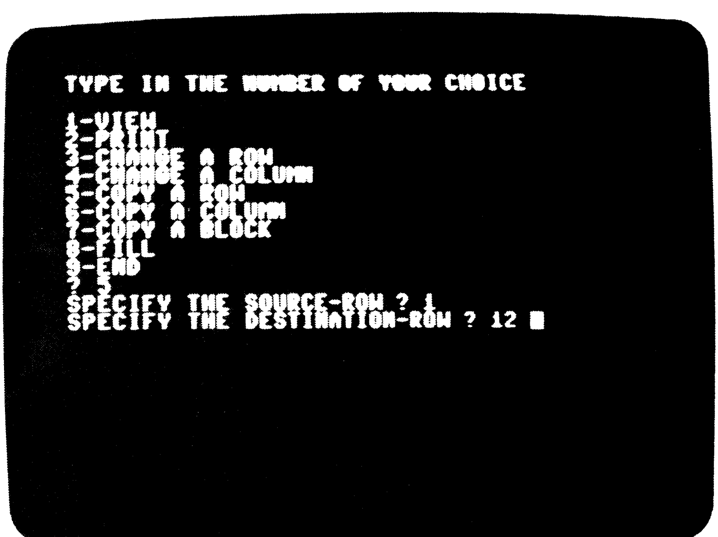
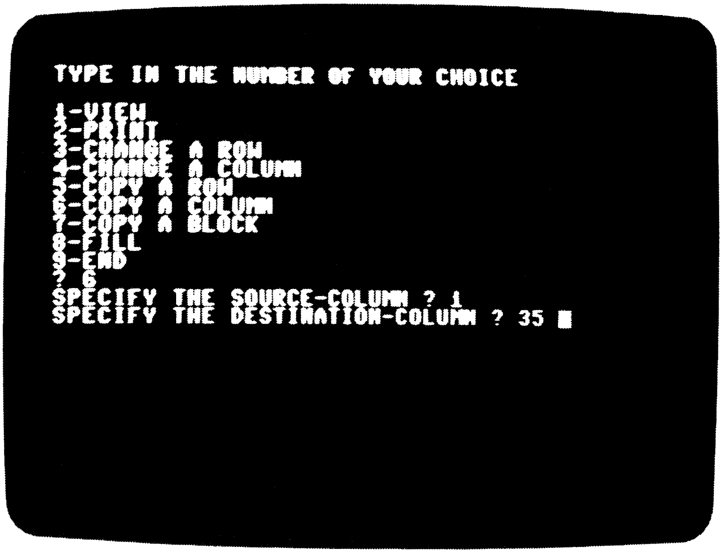


Figure 15-2. Sample use of the Electronic Loom (*continued*)



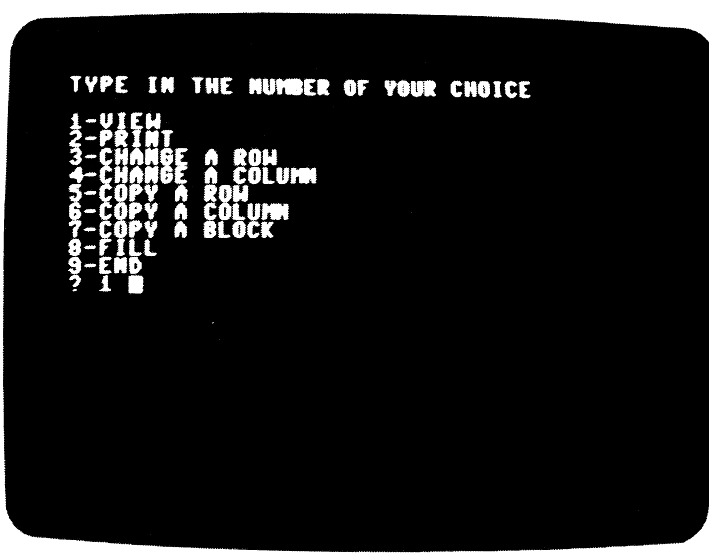


Figure 15-2. Sample use of the Electronic Loom (*continued*)

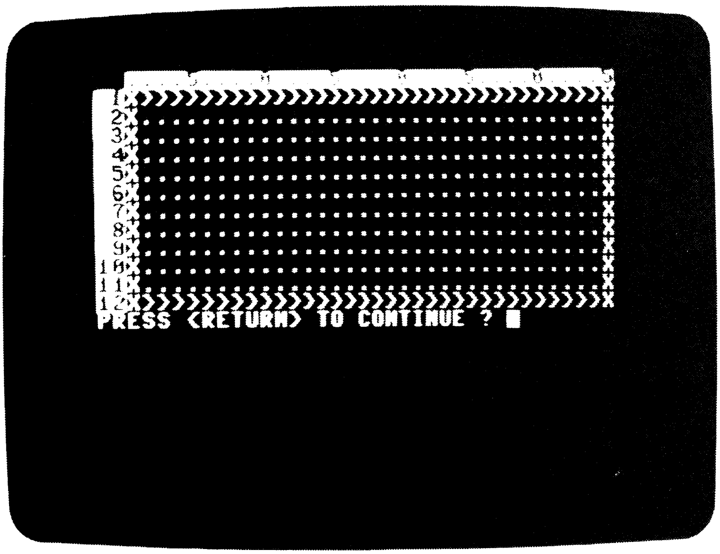


Figure 15-2. Sample use of the Electronic Loom (*continued*)

evident from the preceding discussion; it is more concise to simply show the program in use.

Figure 15-3 shows a sample design created with the program. The design was printed using condensed line spacing: instead of the usual 6

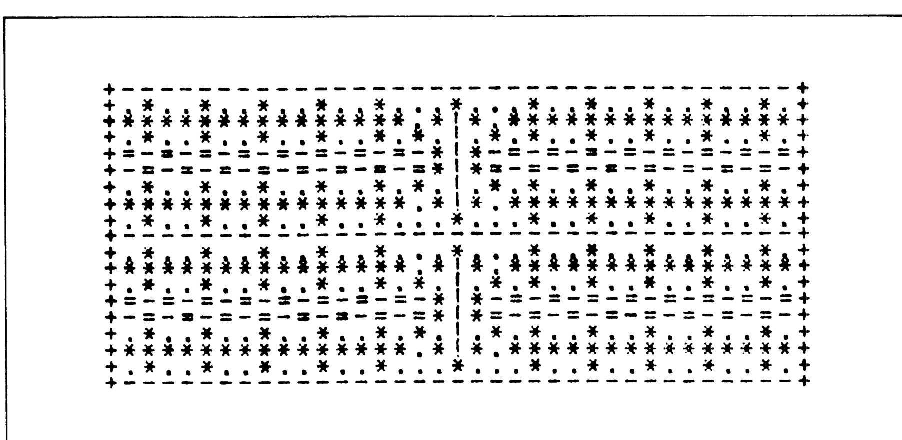


Figure 15-3. Design created with the Electronic Loom and printed at 9 lines per inch

lines per inch, the printer line spacing is set to 9 lines per inch, thus producing a denser, more interesting result. If your printer won't respond to the codes S9$ and S6$ set in line 9, look in your printer owner's manual for control codes to select this feature.

Suppose you find that the code sequence 27,65,6 selects 12 lines per inch (as does the Epson MX-80 printer). To activate this feature, set S9$=CHR$(27)+CHR$(65)+CHR$(6) and S6$=CHR$(27)+CHR$(65)+CHR$(12).

Chapter **16**

Designs in a Circle

Remember the Spirograph design toy? It consists of a large fixed circle and a selection of smaller circles, ellipses, and other shapes. The large circle has cogs on its inner surface, and all the smaller shapes have cogs on their outer surfaces.

To draw a design, you select one of the smaller "rotator" shapes and place it inside the larger "fixed" circle. Place them both on a sheet of paper, place a pen into a hole on the rotator, and using the pen as a handle, begin to turn the rotator inside the fixed circle. As it moves, the pen creates a design on the paper. You can get an astounding variety of designs by varying the smaller figure's size and shape.

In this chapter, you turn your C-64 computer into an electronic Spirograph. Unlike the real thing, you'll only work with a single type of rotator—the circle. Even so, you'll find plenty of variety among the possible designs. By modifying some of the formulas, you can depart from the circle-within-a-circle family and venture into some very unusual patterns. The program uses the C-64's high-resolution graphics and lets you print your designs on the Commodore MPS-801 printer. Figures 16-1 and 16-2 show sample designs created with this program.

Producing high-resolution graphics on the C-64 requires the memory access operations PEEK (examine a memory location) and POKE (store a value in memory).

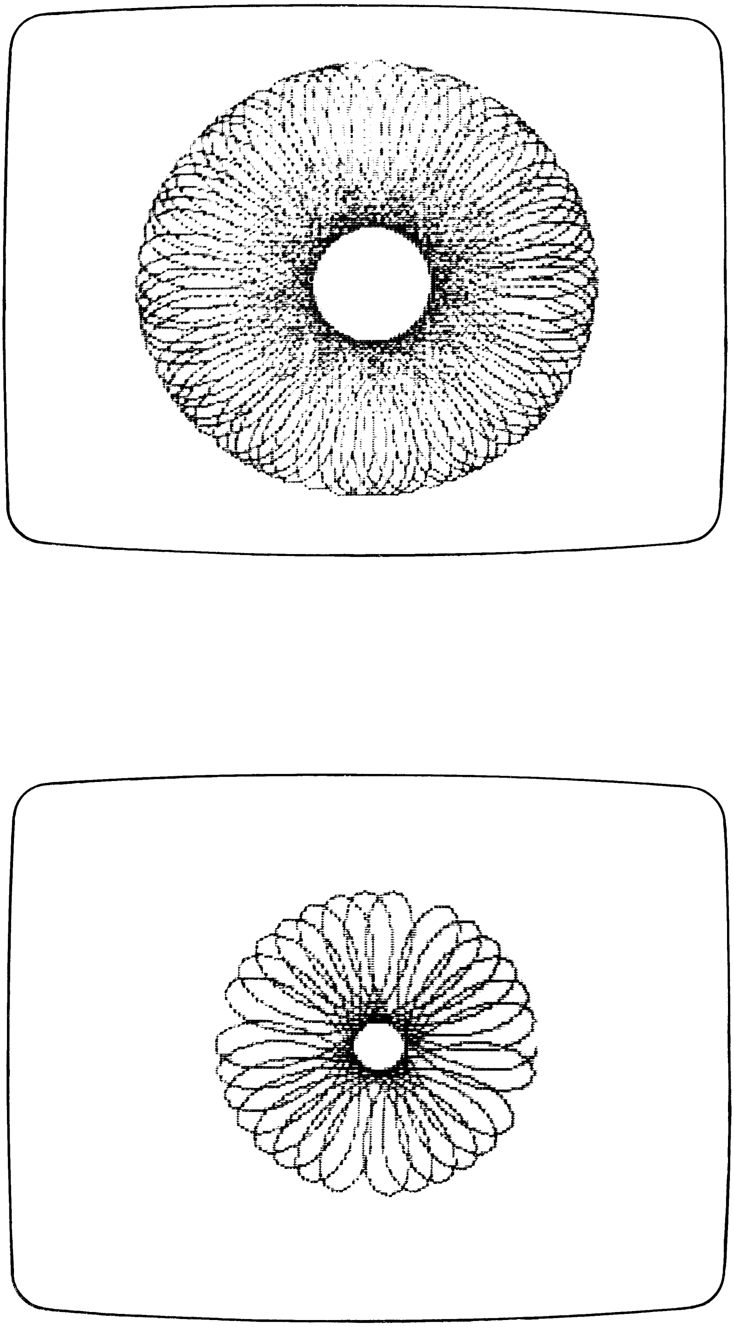


Figure 16-1. Sample designs created using the standard circle-within-a-circle formula

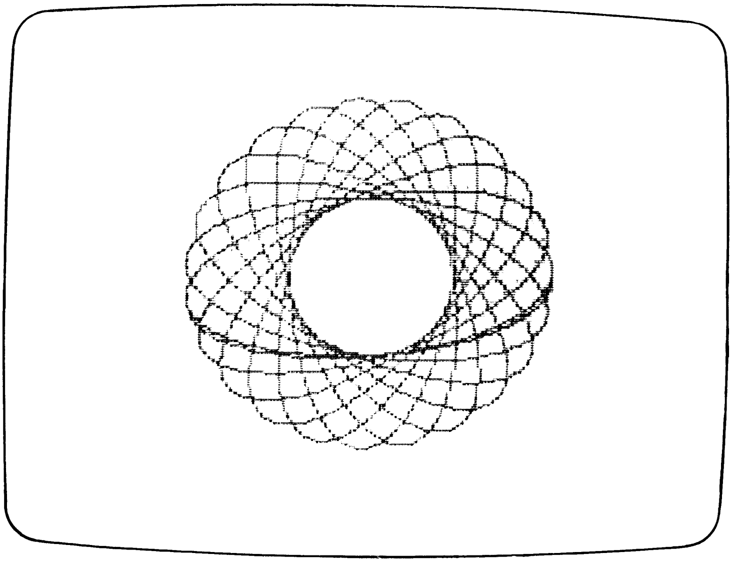


Figure 16-1. Sample designs created using the standard circle-within-a-circle formula (*continued*)

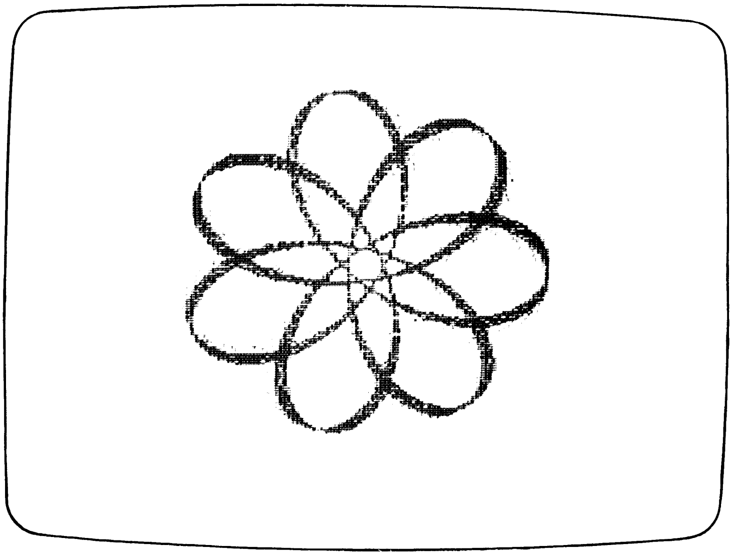
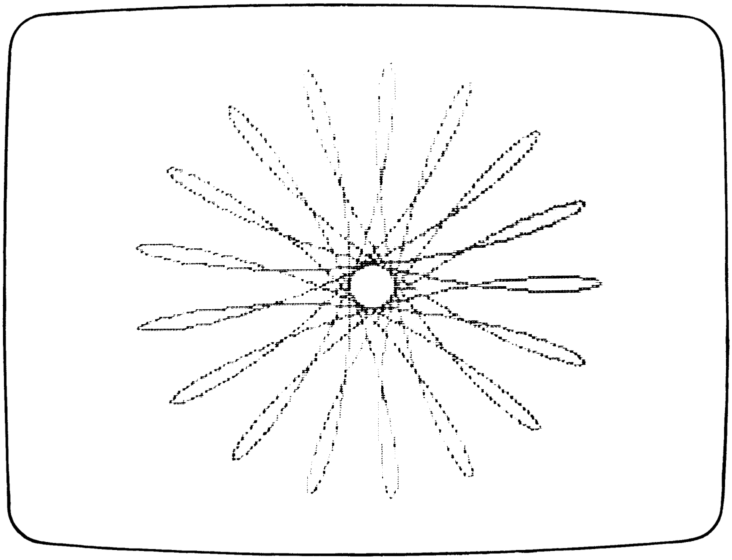


Figure 16-2. Sample designs created using the modified circle-within-a-circle formula



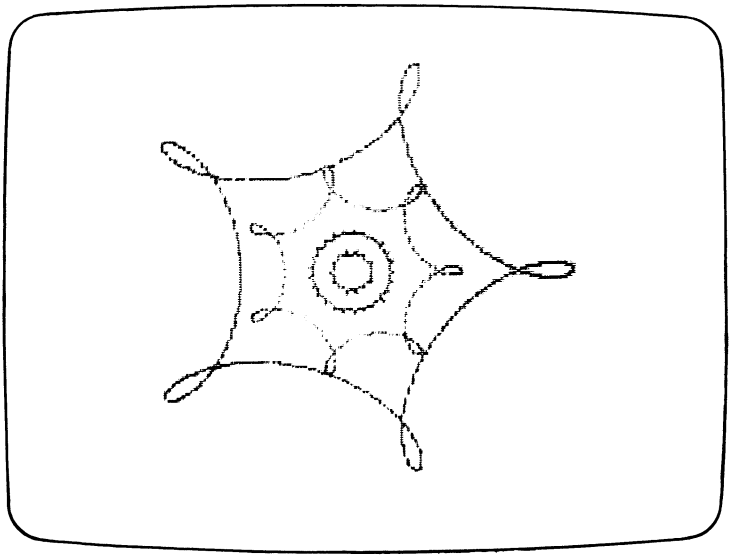


Figure 16-2. Sample designs created using the modified circle-within-a-circle formula (*continued*)

—The Program

The designs are drawn on a high-resolution display that is completely separate from the ordinary text display. The first block reserves memory for this high-resolution display and sets up certain program constants:

```
2 GOTO 9800
4 REM DON'T DELETE LINE 4
5 POKE 53280,1: REM WHITE BORDER
6 POKE 53281,1: REM WHITE SCREEN
7 PRINT CHR$(154);: REM LIGHT BLUE
8 DIM M%(7)
9 FOR B=0 TO 7
10 M%(B)=2↑(7-B)
11 NEXT B
12 SC=11/15: REM ADJUST HORZ/VERT SCALE
14 CX=160: CY=100
15 RR=INT(CY/SC): REM MAXIMUM CIRCLE RADIUS
20 GM=24576
21 CM=16384
30 CC=14*16+1: REM LT BLUE ON WHITE
40 CS$=CHR$(147): REM CLEAR SCREEN
44 NU$="": REM NO SPACES INSIDE QUOTES
50 PI=4*ATN(1)
55 PR=4: REM PRINTER DEVICE NUMBER
```

Line 2 jumps to a routine that protects an area that stores graphic images. Line 4 is the point of return from the routine; as the remark says, this line should remain in the program.

The array M%( ) is used to analyze the contents of the graphics memory one bit at a time. The eight elements of M%( ) correspond to the place values of the eight bits in a byte, as shown below:

Bit number:	0	1	2	3	4	5	6	7
Place value:	128	64	32	16	8	4	2	1

The use of M%( ) is explained in more detail later in this program commentary.

SC is a scaling factor that makes circles look like circles rather than ellipses when they are drawn on the screen. Before plotting a point X,Y, the program multiplies Y by SC.

CC determines the foreground and background color of the graphics screen. To calculate CC, multiply the foreground color code by 16 and add the background color code. For a foreground color of light blue (code 14) and a background color of white (code 1), CC=14×16+1=225.

The point CX,CY is the center of the fixed larger circle in which designs are drawn. GM and CM are the starting locations of high-resolution graphics and color memory.

PR is the printer device number. If your printer has a device number other than 4, change line 55 accordingly.

Printing the Title

The next block of lines prints a title and sets up the graphics memory color scheme.

```
90 PRINT CS$
95 PRINT SPC(7);"*** DESIGNS IN A CIRCLE ***"
96 REM
100 PRINT
105 PRINT "ERASING COLOR MEMORY. WAIT 10 SECONDS."
108 GOSUB 950
```

The subroutine called in line 108 fills graphics memory with the background color specified by line 30.

Setting the Circle Parameters

The next program block prompts you to specify the circle sizes and other details that determine the final appearance of the design:

```
110 PRINT
112 PRINT "RADIUS OF FIXED CIRCLE (10-";RR;")"
114 INPUT RA
115 IF RA<10 OR RA>RR THEN 112
117 PRINT
120 PRINT "RADIUS OF ROTATING CIRCLE (1-";RA;")"
125 INPUT RB
140 IF RB>RA OR RB<1 THEN 115
145 PRINT "DISTANCE OF PEN FROM CENTER (1-";RB;")"
150 INPUT D
155 IF D<1 OR D>RB THEN 145
157 MS=RB
160 PRINT "STEP SIZE (1-";MS;")"
165 INPUT SF
170 IF SF<1 OR SF>MS THEN 160
190 IN=SF/RA
200 AI=IN
260 INPUT "ERASE HI-RES MEMORY (Y/N)";YN$
270 IF YN$="N" THEN 350
280 PRINT "ERASING HI-RES MEMORY. WAIT 45 SECONDS."
290 GOSUB 910: REM ERASE
```

```
350 PRINT "WHILE DRAWING IS IN PROGRESS,
    PRESS ANY"
352 PRINT "KEY TO RETURN TO MENU."
353 PRINT
354 PRINT "NOW PRESS RETURN TO START DRAWING."
380 INPUT XK$
```

Refer to Figure 16-3 for a pictorial explanation of many of the quantities referred to in this block.

Four key parameters are set during this specification dialogue. RA is the radius of the fixed outer circle. The upper limit for RA is determined by the maximum Y coordinate that will fit on the screen, taking into account the use of the scaling factor SC.

RB is the radius of the rotating inner circle. D is the distance of the pen from the center point of the rotating circle MS is the maximum step size allowed; it is always set equal to the radius of the rotating circle.

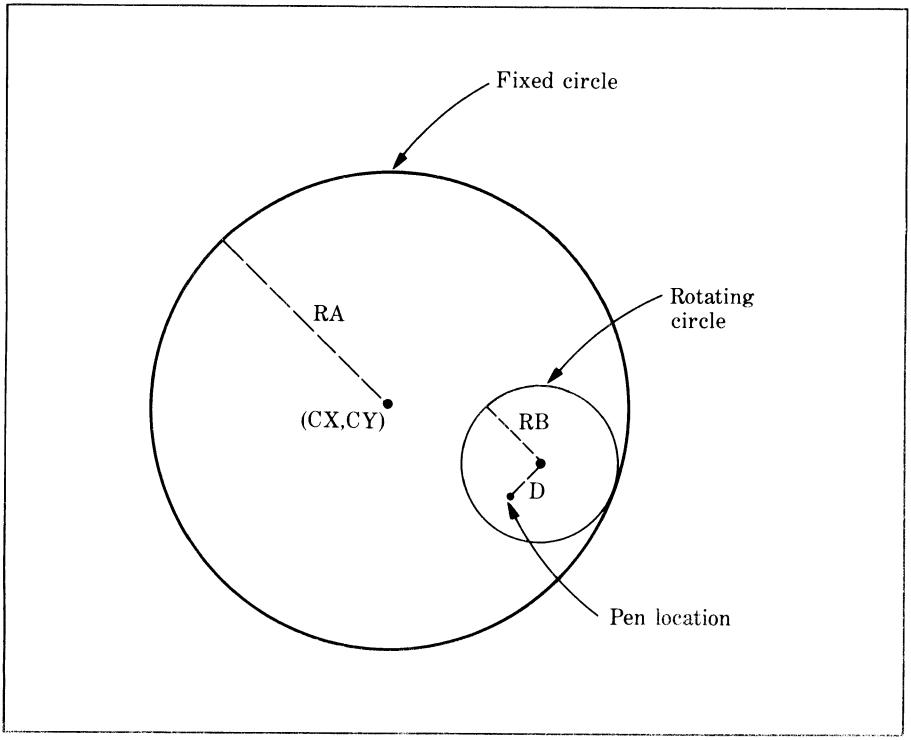


Figure 16-3. Parameters that determine the design

Lines 260-290 gives you an opportunity to erase the high-resolution graphics memory before beginning the drawing. When running the program, you should always select this option for the first drawing. For later drawings, you may decline to erase the graphics screen so that a new drawing can be superimposed upon a previously drawn image.

Drawing the Design

The next block of lines draws the design:

```
390 GOSUB 990: REM SWITCH TO GRAPHICS
430 A=0: REM INITIAL ANGLE
440 GOSUB 820: REM COMPUTE NEXT X AND Y VALUES
450 GOSUB 870: REM PLOT X,Y
460 A=A+AI: REM NEXT ANGLE
490 GET C$
500 IF C$=NU$ THEN 440
```

These lines comprise a loop (a repeating sequence): a plot a point, rotate the inner circle, and plot another point. The loop continues until a key is pressed.

The subroutine called in line 390 switches from the text display to the graphics display. Line 430 sets the initial angle. The subroutine called in line 440 calculates the correct X and Y coordinates for angle A, and the subroutine called in line 450 plots the point. Line 460 rotates the inner circle by adding an increment to angle A. Before continuing, line 490 checks to see whether a key has been pressed. If none has been pressed, line 500 jumps back to calculate the coordinates of the next point.

Continuation Menu

If a key has been pressed during the drawing loop, these lines print a continuation menu:

```
510 GOSUB 1030: REM RESTORE TEXT DISPLAY
520 PRINT CS$
522 PRINT "FIXED CIRCLE RADIUS      =";RA
523 PRINT "INNER CIRCLE RADIUS      =";RB
524 PRINT "PEN DISTANCE (INNER CIR.)=";D
525 PRINT "STEP SIZE   =";SF
530 PRINT
540 PRINT "1-CONTINUE DRAWING"
542 PRINT "2-CHANGE CIRCLE PARAMETERS"
544 PRINT "3-PRINT DESIGN"
```

```
545 PRINT "4-FREEZE DRAWING"
546 PRINT "5-END PROGRAM"
552 INPUT C
555 IF C<1 OR C>5 THEN 530
560 ON C GOTO 610,110,630,570,9910
```

The five continuation options are: 1 - CONTINUE DRAWING, 2 - CHANGE CIRCLE PARAMETERS, 3 - PRINT THE DESIGN, 4 - FREEZE DRAWING (view the design without changing or adding points to it), and 5 - END.

If you select option 2, change circle parameters, the program simply jumps back to the specification dialogue.

Continuing and Freezing the Design

The next block handles options 1 and 4 (continue the drawing and freeze the drawing):

```
570 AI=0: REM ANGLE INCREMENT IS 0
580 GOTO 612
610 AI=IN: REM SET NON-ZERO ANGLE INCREMENT
612 GOSUB 990
615 PRINTCS$
620 GOTO 440
```

For the freeze option, lines 570 and 580 set the angle increment to 0 before reentering the drawing routine. For the continuation option, line 610 sets the angle increment to a nonzero value and line 612 jumps back into the drawing loop.

Printing the Design

Because of variations in the way printers handle high-resolution graphics, the following block of lines works only with the Commodore MPS-801 printer or other compatible graphics printers:

```
630 OPEN 1,PR
640 FOR Y1=0 TO 28: REM 29 7-DOT ROWS
650 PRINT#1,CHR$(8);:REM GRAPHICS MODE
660 FOR PX=0 TO 319
670 OC=128: REM GRAPHICS CHARACTER CODE
680 FOR Y2=0 TO 6
690 PY=Y1*7+Y2
700 IF PY>199 THEN Y2=6: GOTO 740
710 MA=GM+40*(PY AND 248)+(PY AND 7)+(PX AND 504)
720 BV=SGN(PEEK(MA) AND M%(PX AND 7))
```

```
730 OC=OC OR M%(7-Y2)*BV
740 NEXT Y2
750 PRINT#1,CHR$(OC);
760 NEXT PX
770 PRINT#1,
780 NEXT Y1
790 CLOSE 1
800 GOTO 520
```

The MPS-801 prints graphic data in columns that are seven dots long, as shown in Figure 16-4. The program must do quite a few computations to translate a screen of video dots into a page of printed dots.

Line 630 sets up the printer as output device number PR (PR must be correctly set in line 55).

The graphics screen consists of 320 columns numbered 0-319 and 200 rows numbered 0-199. Copying a screen to the printer involves printing 29 rows in which each row contains 320 7-dot columns.

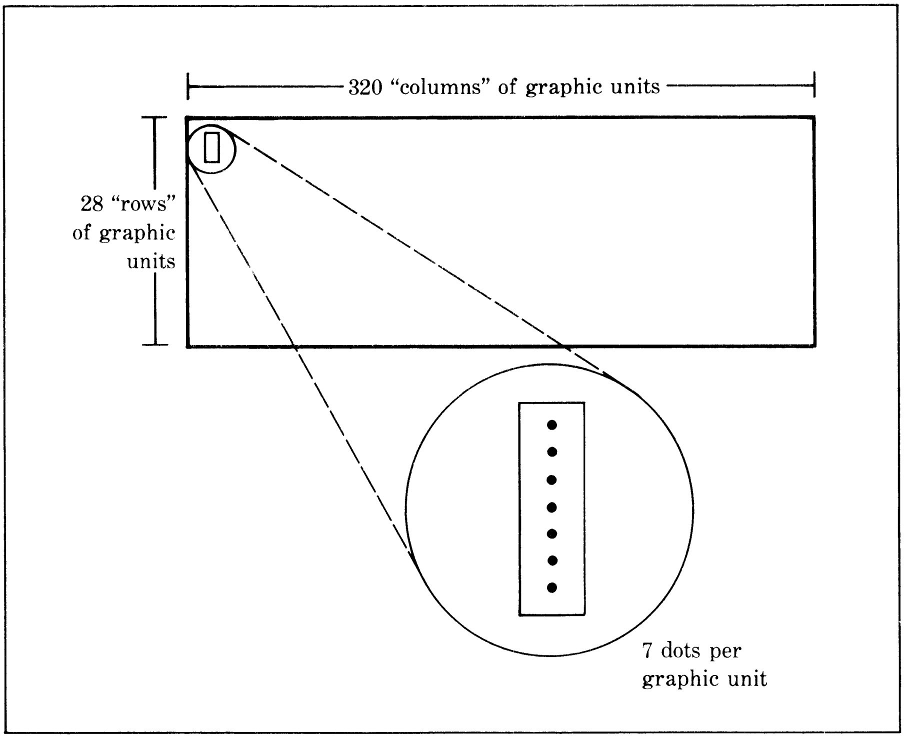


Figure 16-4. Printing graphics units on the Commodore MPS-801

Twenty-nine rows of seven dots gives 203 dots in the vertical dimension — three more than the graphics screen actually contains. Accordingly, the last three dots of the 29th row are always printed as blanks.

Variable Y1 in line 640 counts through all 29 rows. Variable PX in line 660 counts through all 320 columns. Variable Y2 in line 680 counts through the 7-dot columns that make up a graphics character on the MPS-801 printer. Line 650 sets the MPS-801 printer in the dot graphics mode, and line 670 initializes OC (the output character) to a graphics value (graphics characters are greater than or equal to 128).

Y1 and Y2 together produce PY, the vertical axis coordinate of the point being printed, according to this formula:

$$PY = Y1 \times 7 + Y2$$

Line 700 checks whether PY is one of the three nonexistent graphics rows mentioned previously. If it is, the program advances to the next coordinate. Otherwise, the program examines the on/off status of point PX,PY. Line 710 finds the memory location MA that contains the desired point. Line 720 uses the array M%( ) to determine whether the point is on or off. The expression PX AND 7 makes all but the three least significant bits of PX zero, producing a value from 0 to 6. The expression PEEK(MA) and M%(PX AND 7) returns a 0 if the indicated point is off and the place value of the point if it is on.

Line 730 incorporates the current point status into the output character code OC. After all seven dots of the graphics until have been accounted for, line 750 prints the graphics character OC. After a full 320 graphics units have been printed, comprising one output row, line 770 prints a carriage return to start a new line.

After all 28 graphics rows have been printed, line 790 closes the printer device and line 800 jumps back to the continuation menu.

Subroutines and Auxiliary Routines

Some of the program logic is put into subroutine form to facilitate program debugging and to make the main program logic easier to follow.

Calculating a Point on the Design These lines calculate the coordinate of a point on the design, based on the current parameter setting:

```
820 X=(RA-RB)*COS(A)+D*COS((RA-RB)*A/RB)
830 Y=(RA-RB)*SIN(A)-D*SIN((RA-RB)*A/RB)
```

```
840 PX=INT(ABS(X)+.5)*SGN(X)+CX
850 PY=INT(SC*(INT(ABS(Y)+.5)*SGN(Y)+.5))+CY
860 RETURN
```

Lines 820 and 830 are based on standard math formulas for "circles within circles," or epicycloids. Both X and Y are computed as functions of the angle A, which is measured in radians rather than degrees (1 radian=180/PI).

Initially X and Y are calculated with respect to the origin 0,0. Line 855 adjusts X and Y since the center point of the design is actually at CX,CY. Note the use of the scaling factor SC in line 850 to compensate for a vertical bias present in most television images.

Plotting a Point Given a coordinate pair PX,PY, the following subroutine plots the corresponding point in graphics memory:

```
870 IF PX<0 OR PX>319 OR PY<0 OR PY>199 THEN RETURN
880 MA=GM+(40*(PY AND 248))+(PY AND 7)+(PX AND 504)
890 POKE MA,PEEK(MA) OR M%(PX AND 7)
900 RETURN
```

MA is the memory address that contains the point. However, MA contains seven other points as well (one for each of the eight bits in a byte). The expression M%(PX AND 7) in line 890 indicates which bit is referenced by PX,PY. Refer to *Your Commodore 64* for a full explanation of graphics-plotting techniques.

Initializing Graphics and Color Memory Before doing any graphics, the program must load a uniform background and foreground color scheme into color memory. The following subroutine does this:

```
910 FOR J=GM TO GM+7999: REM FILL GRAPHICS MEMORY
920 POKE J,0
930 NEXT J
940 RETURN
```

Before the first design is drawn, the high-resolution graphics memory must be erased (otherwise, it will contain an undesirable dot pattern). When changing design parameters, you may also want to erase the graphics area before drawing the new design. These lines do the erasing:

```
950 FOR J=CM TO CM+999
960 POKE J,CC: REM CC = FG*16+BKG
```

```
970 NEXT J
980 RETURN
```

Switching Between Text and Graphics The program uses two separate memory areas to store text (such as the menus) and graphics. The next block of lines switches from text to graphics:

```
990 POKE 56576,(PEEK(56576) AND 252) OR 2
1000 POKE 53272,8
1010 POKE 53265,PEEK(53265) OR 32
1020 RETURN
```

The next block switches back to text from graphics:

```
1030 POKE 56576,(PEEK(56576) AND 252) OR 3
1040 POKE 53272,21
1050 POKE 53265,PEEK(53265) AND 223
1060 RETURN
```

Allocating Memory for Graphics The final two routines protect memory for the high-resolution graphics screen and release the memory at the end of the program.

```
9800 POKE 52,64
9810 POKE 56,64
9820 CLR
9830 GOTO 4
9910 POKE 52,128
9920 POKE 56,128
9930 CLR
9940 END
```

—Using the Program

You should be able to duplicate the designs shown in Figure 16-1. Experiment with different values for the larger circle, smaller circle, pen location, and step size.

One thing you'll notice is that the program draws designs very slowly. This is because of the large number of calculations that must be made before each point is plotted.

To reduce your waiting time, select a larger step size. This will cause the design to be drawn in dotted lines; however, you will quickly see the general outline of the design. If you like the design, you can go back to the continuation menu, select the change circle parameters option, and change only the step size (leave all other parameters unchanged).

For solid line designs, select a step size of 1. Designs using this step

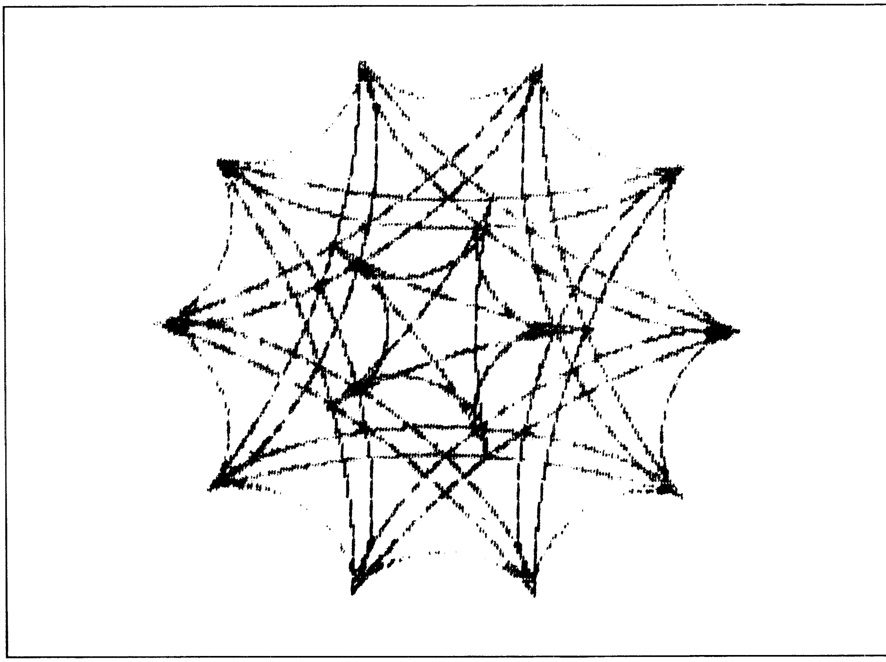


Figure 16-5. Sample printout of overlaid designs

size may take as long as 30 minutes to complete—so be sure you have previewed the design using a larger step size.

You'll notice that the program keeps drawing the design, even after it begins retracing lines it has already drawn. At this point, stop the drawing by pressing any key and you will see the continuation menu.

To superimpose one design upon another, select the change circle parameters options, enter new values, but *do not erase the graphics area* when the program gives you that as an option. Your previous drawing will remain on the screen while the new one is drawn.

Figure 16-5 shows a sample printout. If you plan to make such printouts, set SC=1 in line 12 *before* running the program.

Variations You can select another family of designs by changing the formulas. However, you must take care in making such changes or the design will exceed the limits of the C-64 graphics window.

The following changes produced the pictures shown in Figure 16-2:

```
820 X=(RA-RB)*COS(A)+D*COS((RA+RB)*A/RB)
830 Y=(RA-RB)*SIN(A)-D*SIN((RA+RB)*A/RB)
```

Chapter 17

Secret Messages

Cryptography, or secret writing, has been in use for almost 4000 years. Diplomats, military personnel, religious figures, and furtive lovers have all used it to send private messages through public channels. And a lot of people practice it just for fun.

The Secret Message Processor (SMP) program presented in this chapter turns your C-64 into a full-function code machine. The program converts English or any other language (plaintext) into apparent gibberish (ciphertext) and vice versa. The text is entered from the keyboard or read from a disk file, and the result is displayed on the monitor, output to a printer, or saved in a disk file.

In a typical use of the SMP, you and a friend both have access to a C-64 computer. The two of you agree on a key value prior to sending the secret message. You run the SMP, input the key value, and type in the plaintext. The program outputs ciphertext to a printer or disk file. You send your friend the printout or disk.

When your friend receives the ciphertext, the process is repeated: running the SMP, entering the key value, and typing in the ciphertext or loading it from disk. Presto! The program restores the original message.

Well, not quite presto: the program processes sample text at the rate of 3.478 characters per second, or 0.2875 seconds per character. At this rate, it takes 11.5 seconds to process (encipher or decipher) a line of 40 characters and over 21 minutes to process a 1000-word document.

The processing delay is acceptable for short messages, but is too slow for longer documents. Fortunately, the program offers a disk-to-disk option that allows you to ignore the program while it processes a long prepared document. You can read the processed text later without any delay for processing. This procedure is explained later.

—Secrets of the SMP

We'll start 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 of the plaintext according to a definite set of rules. The resultant letter-frequency distribution (the number of A's, B's, C's, and so forth) remains the same, but 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.

Figure 17-1 shows examples of each type of cipher.

The cryptographic method employed by the SMP 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 the apostrophe, the hyphen, the digits 0 through 9, and all uppercase and lowercase letters.

The SMP 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, as shown in the following example:

Message:	m	e	e	t	m	e	a	t	7	p	m
Key stream:	47	17	19	34	56	3	4	57	58	34	36

Given a character-number pair, the program derives the ciphertext character.

The SMP can generate a very large number of different key streams; to decipher a message, you use the same key stream that was used to encipher it. The "key value" determines which key stream is used.

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.

Figure 17-1. Examples of simple transposition and substitution ciphers

After pairing the plaintext characters with numbers from the key stream, the program follows these steps:

1. Find the location of character *c* within the 64-character cipher list. By convention, the first position in the list is position 0, and the last is 63. Therefore, the position of character *c* is a number from 0 to 63. Refer to this number as *p(c)*, short for *position of c*.
2. Take the number *n* that is paired with character *c*, and calculate *n* XOR *p(c)*. (The XOR operator is explained next.) The result of this calculation is a number ranging from 0 to 63. Call it *p(d)*.
3. Locate the character within the cipher list at position *p(d)*. Call

that character *d*. It is the ciphertext character corresponding to plaintext character *c*.

—The XOR Operator

The 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 on/off status of the corresponding bit in the result C.

The following table summarizes the rules for comparing bits from A and B.

A XOR	B	=	C
0	0		0
0	1		1
1	0		1
1	1		0

For example:

	Binary	Decimal
(1) A	10101110	174
B	01110111	119
C	11011001	217
(2) A	11011001	217
B	01110111	119
C	10101110	174
(3) A	11011001	217
B	10101110	174
C	01110111	119

As illustrated in these examples, XOR has a special property: if C=A XOR B, then A=C XOR B and B=C XOR A. In other words, 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 SMP is able to encipher or decipher a message using the same program logic.

For a specific example refer to the message shown in Table 17-1. The position of M, the first letter of the message, in the cipher list is 24; in short, p("M")=24. The key stream number assigned to M is 47. Calculating 24 XOR 47 produces the number 55. The character in the cipher list at position 55 is "r." By doing the same for each letter and number, you encode the entire message.

The deciphering process is exactly the same. The ciphertext characters are paired with numbers from the original key stream, and the preceding steps 1 through 3 are repeated.

Table 17-2 illustrates the calculations for deciphering the sample message. Note that it is identical to Table 17-1, except that the data from columns 1 and 2 are exchanged with the data from columns 4 and 5.

You can sum up the enciphering/deciphering process with two equations. Remember that *p(character)* refers to the position of *character*

Table 17-1. Steps for Enciphering the Message *meet me at 7 pm*

Input Character c	Cipher-list* Position p(c)	Key Stream Value n	Cipher-list Position p(d) = p(c) XOR n	Output Character d
m	24	47	55	R
e	16	17	1	—
e	16	19	3	L
t	31	34	61	X
m	24	56	32	u
e	16	3	19	h
a	12	4	8	6
t	31	57	38	A
7	9	58	51	N
p	27	34	57	T
m	24	36	60	W

*Cipher list: '-0123456789abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Table 17-2. Steps for Deciphering the Message R—LX uh 6A NTW

Input Character c	Cipher-list* Position p(c)	Key Stream Value n	Cipher-list Position p(d) = p(c) XOR n	Output Character d
R	55	47	24	m
—	1	17	16	e
L	3	19	16	e
X	61	34	31	t
u	32	56	24	m
h	19	3	16	e
6	8	4	12	a
A	38	57	31	t
N	51	58	9	7
T	57	34	27	p
W	60	36	24	m

*Cipher list: '-0123456789abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ

within the cipher list. Knowing *p(character),* you can find *character,* and knowing *character,* you can find *p(character).*

To encipher *c:*

$$p(d) = p(c) \text{ XOR } n$$

To decipher *d:*

$$p(c) = p(d) \text{ XOR } n$$

—Source of the Key Stream

The sequence of numbers that comprises the key stream is the key to enciphering or deciphering a message. Once a message has been enciphered, only the identical keystream can restore it to plaintext.

When this cryptographic method is used manually, both parties (sender and receiver) keep a printed copy of the key stream. They may even have a book of different key streams and a prior agreement about which key stream to use on each given day.

The key stream we'll use is built right into the C-64. It's more commonly known as the random number generator, or RND in BASIC.

The RND function returns an apparently 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 C-64's memory. Each time the C-64 executes the RND function, the seed value changes, so that the next time RND(1) is used, it generates a different value. After a very large number of uses, RND(1) completes its sequence and starts over.

Our key stream must consist of numbers between 0 and 63. To scale the result of RND(1) into the range 0-63, we multiply by 64 and take the integer portion of the result.

We must also be able to generate a repeatable sequence of numbers. To do this, we use the RND function with a negative argument greater than −32768 and less than 0. This "primes" the RND function with a particular seed value. For instance, using RND(−1) establishes 1 as the seed. Subsequent uses of RND(1) will return the sequence 0.328780872, 0.978964086, 0.895758909, 0.161031701, ... Scaling to the present range produces 21, 62, 57, 10, ...

In summary, the C-64 has a built-in "book" of key streams. To select a given key stream for enciphering, specify a negative number from −32767 to 0. The same number must be used as the key to decipher a given message.

—The Program

The program is presented in logical blocks. Type them in as you read along. Before you begin typing, put your C-64 into lowercase mode by pressing the SHIFT and COMMODORE keys together. This will allow you to type in certain string constants correctly.

The first block sets up certain useful display and string constants.

```
1 poke 657,128: rem disable uc/lc switch
2 print chr$(14);: rem lowercase display
10 s1$=" ": rem 1 space inside quotes
20 nu$="": rem no spaces inside quotes
30 cs$=chr$(147): rem clear screen
40 et$=chr$(26): rem end of text signal
50 el$=chr$(13): rem end of line signal
60 pr=4: rem printer device number
65 tv=3: rem tv device number
66 di=2: rem data input channel no.
68 do=3: rem data output channel no.
```

Lines 1 and 2 lock the C-64 into lowercase mode.

ET$ is the end-of-text character that may be used to terminate keyboard or disk entries. It corresponds to the keyboard character CONTROL Z. If your printer has a device number different than 4, change line 60 accordingly.

Storing the Cipher List

The next lines set up the cipher list:

```
70 tb$="'-0123456789"
80 tb$=tb$+"abcdefghijklmnopqrstuvwxyz"
90 tb$=tb$+"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
100 tl=len(tb$)
110 if tl=64 then 150
120 print "character table does not contain"
130 print "64 characters. can't continue."
140 stop
150 for t=1 to tl-1
160 if mid$(tb$,t,1)<mid$(tb$,t+1,1) then 185
170 print "invalid cipher list -- check sequence."
180 stop
185 next t
```

TB$ contains the cipher list (the list of cipherable characters). It is very important to type the table exactly as shown: 64 characters listed in ascending order according to their C-64 keyboard codes. To type in lines 70-90 correctly, you must have your computer in lowercase mode (press the SHIFT and COMMODORE keys together until your display shows lowercase letters).

Lines 110-130 ensure that the list does contain 64 characters; however, it is up to you to ensure that the correct characters are used and that the sequence is correct.

Lines 110-185 check the cipher list for valid length and sequence.

Displaying the Menu

The next block prints a menu:

```
190 print cs$
200 print "secret message processor"
210 print
220 input "read text from: 1-keyboard  2-disk ";s
230 if s<>1 and s<>2 then 220
235 lc$=nu$: rem deactivate lowercase option
```

```
236 uc$=nu$: rem deactivate uppercase option
240 input "output to: 1-tv  2-disk  3-printer ";d
250 if d<>1 and d<>2 and d<>3 then 240
```

The variable S indicates the input device (1=keyboard, 2=disk file); D indicates the output device (1=CRT, 2=disk file, 3=printer).

Based on your specifications for S and D, the following block sets up the necessary input/output channels:

```
260 if s=1 then 290
270 input "view disk directory (y/n)? ";yn$
272 if yn$<>"y" then 280
274 gosub 2900
280 fi$=nu$
282 input "name the input file: ";fi$
284 if fi$=nu$ then 190
285 open di,8,2,fi$+",seq,read"
290 on d goto 400,310,380
310 input "view disk directory (y/n)? ";yn$
312 if yn$<>"y" then 320
314 gosub 2900
320 fo$=nu$
322 input "name the output file: ";fo$
324 if fo$=nu$ then 190
330 open do,8,3,"@0:"+fo$+",seq,write"
360 goto 410
380 open do,pr: rem open printer channel
385 lc$=chr$(17): rem activate lowercase
387 uc$=chr$(145): rem activate uppercase
390 goto 410
400 open do,tv: rem open tv channel
```

The variables LC$ and UC$ are used to force the printer into lowercase mode.

Inputting the Key Stream

The following block asks you to input the key and then selects the corresponding key stream:

```
410 ky=0
412 input "enter the key (0=no processing): ";ky
415 print "secret message processor at work. wait"
420 r=rnd(-abs(ky))
```

Line 412 prompts you to enter the key. When using the program, enter any whole number or fraction from 0 to 32767. The program converts your entry into a negative value that can be used to reset the random number seed.

To turn off the code processor, enter 0 as the key; the input text will be output to the specified device (CRT, disk file, or printer) without any changes. The no-processing option comes in handy when entering a lengthy text, as explained at the end of this chapter.

Line 420 sets the random number seed according to your specification.

Initializing Counters and Buffers

The next block of lines initializes certain counters and buffers before the text processing begins:

```
430 el=1
440 cr=0
442 ot$=nu$
```

The variable EL indicates the end-of-line status. EL=1 indicates that a carriage return has just been read. Two consecutive carriage returns are equivalent to an end-of-text character. CR counts the characters remaining in the program's input buffer; when CR=0, the program gets another line of input from the keyboard or disk file. OT$ is the output line; as each character is processed, the program adds it to OT$. When a carriage return is read, the program outputs OT$ to the CRT, disk file, or printer.

Inputting a Character

The logic for inputting from disk and keyboard is broken into two blocks. Here's the routine to input a character from a disk file:

```
444 if s=1 then 460
446 get#di,c$
448 if st=0 then 660
450 if st<>64 then print "file error. canceling
    the operation"
452 c$=et$
454 goto 660
```

Line 444 causes the computer to skip to the next block in case input is from the keyboard. Line 446 attempts to get a character, and line 448 determines whether the attempt was successful. If not, lines 450-454 terminate the processing of text.

The following lines get a character from the keyboard:

```
460 c$=et$
470 if cr>0 then 630
480 if el=1 then 550
490 c$=el$
500 el=1
510 goto 670
550 print
560 print "type a quote, then a line of text"
570 print "enter an empty line to quit"
575 b$=nu$
580 input b$
600 bl=len(b$)
610 cr=bl
620 if cr=0 then 740
630 el=0
640 c$=mid$(b$,bl-cr+1,1)
650 cr=cr-1
```

The routine draws characters one at a time from a buffer B$. When the buffer is empty (CR=0), the program prompts you to enter another line. The program assumes an end-of-text condition upon reading two consecutive carriage returns or a single end-of-text character (ET$).

Processing the Character

Upon completion of lines 444-650, the variable C$ contains the character just read. The following lines process the character:

```
660 if ky=0 then 740
670 gosub 950
680 a=ix
690 if a=0 then 740
700 a=a-1
710 b=int(rnd(1)*tl)
720 c=(a and not b) or (b and not a)
730 c$=mid$(tb$,c+1,1)
```

In line 660, the program checks to see if the code processor is turned off (KY=0). If it is, the program skips the rest of the processing section and goes to the output routine. Otherwise, the program continues with the subroutine called in line 670, which searches for the character C$ inside the cipher list TB$.

If A=0 in line 690, the C$ is not in TB$ so the program skips to the output section. Otherwise, the variable A contains a number from 1 to 64. Subtracting 1 from A (line 700) brings it into the range 0-63. Now A corresponds to p(c) in the preceding examples.

Line 710 gets the next number from the key stream (that is, the random number generator) and stores it in B. The subroutine called in line 720 calculates A XOR B and stores the result in the variable C. The variable C corresponds to d(c) in the preceding examples. Finally, line 730 replaces C$ with the corresponding character from the cipher list.

Adding to the Output Buffer

The following block of lines adds C$ to the output buffer and prints the buffer in case C$ is a carriage return or an end-of-text character.

```
740 ot$=ot$+c$
750 if c$<>et$ and c$<>el$ then 444
780 print#do,lc$;ot$;
790 if c$<>et$ then 442
800 print#do,
810 close di
820 close do
```

In line 740, C$ is added to the current contents of the output buffer OT$. Line 750 causes the program to loop back for another character unless it is a terminating character (carriage return or end-of-text).

In case of a terminating character, line 780 prints the current buffer contents on the specified output device. If the character is a carriage return, line 820 jumps back for another character from the input device. If the character was an end-of-text marker, lines 800-820 close the input and output devices.

Displaying the Continuation Menu

The following lines print a continuation menu:

```
830 print "processing complete"
860 input "<c>ontinue or <q>uit? ";cq$
870 if cq$="c" then 190
880 if cq$<>"q" then 860
890 poke 657,0: rem enable uc/lc switch
895 end
```

If you select the continue option, the program resumes at the main menu, allowing you to specify new input and output devices and a new key.

Searching the Cipher List

Here's the subroutine to search for a character within the cipher list:

```
950 ll=0
960 ul=tl+1
970 ix=int((ul-ll)/2)+ll
980 tc$=mid$(tb$,ix,1)
990 if tc$=c$ then 1040
1000 if tc$<c$ then ll=ix
1010 if tc$>c$ then ul=ix
1020 if ll<ul-1 then 970
1030 ix=0
1040 return
```

Simple sequential search logic has been used in other programs in this book (see the Guess My Word program). However, because of the length of the search list, the sequential search technique is too slow. Instead, a "binary search" technique is used. A binary search divides the list into successively smaller intervals until the desired data is found or the interval is null (no data between the interval's lower and upper limits).

The lower limit of the interval is set to 0, and the upper limit set to 1 more than the length of the cipher list TB$ (lines 950 and 960). IX is an index pointing to the current search location. It is always set equal to a midpoint between the lower and upper limits (line 970). Line 980 examines the TC$, the character at position IX in TB$. If it matches C$, the search ends, and the subroutine returns to the main program with IX containing the location of character C$ inside the cipher list.

If TC$ does not match C$, the program resets either the lower or the upper limit, depending on whether TC$ precedes or follows C$. The midpoint IX is recalculated for this new interval, and the checking process is repeated.

The cycle continues until the program finds a matching character or until the interval defined by LL,UL contains no character positions (UL−LL=1). In the latter case, the search fails, so IX is set to 0, indicating that C$ is not found in TB$.

This searching method is three to four times faster than a sequential search for a list of this size. However, it will only work if the characters in TB$ are given in ascending order of ASCII codes. That is why lines 70-90 must be entered exactly as shown.

Reading the Disk Directory

These lines read the disk directory without erasing the resident program (unlike the ordinary LOAD "$",8 command):

```
2900 print "loading directory..."
2910 open 1,8,4,"$,seq,read"
2920 iw=0
2930 if st<>0 then 3040
2940 get#1,a$
2950 if len(a$)=0 then 2920
2960 if a$>chr$(31) and a$<chr$(122) then 3010
2970 if iw=0 then 2930
2980 iw=0
2990 print
3000 goto 2930
3010 if iw=0 then wl=1
3015 iw=1
3020 print a$;
3023 wl=wl+1
3024 if wl<17 then 3030
3026 print
3028 wl=0
3030 goto 2930
3040 close 1
3050 print
3060 return
```

When you run the program, expect a delay while the computer searches for file names among all the other directory information.

—Using the Program

Figure 17-2 shows a sample run of the program, illustrating the keyboard-to-TV option for enciphering and the keyboard-to-TV option for deciphering. The sample run shows what happens when an incorrect key is used to decipher a message.

—Tips for Processing Lengthy Texts

As mentioned previously, 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 SMP, specifying the keyboard as the input device and a disk file PLAINTEXT as the output device. Enter a key of 0 (no processing). 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

```
SECRET MESSAGE PROCESSOR

READ TEXT FROM: 1-KEYBOARD  2-DISK  1
OUTPUT TO: 1-TV  2-DISK  3-PRINTER  1
ENTER THE KEY (0=NO PROCESSING):  32050
SECRET MESSAGE PROCESSOR AT WORK. WAIT

TYPE A QUOTE, THEN A LINE OF TEXT
ENTER AN EMPTY LINE TO QUIT
The new password is
r30 SXz 'ywCUnSm ew

type a quote, then a line of text
enter an empty line to quit
grapefruit
Lz23f5JoAm

type a quote, then a line of text
enter an empty line to quit


processing complete
<c>ontinue or <q>uit? c
```

Figure 17-2. Sample run of the Secret Message Processor

```
secret message processor

read text from: 1-keyboard  2-disk  1
output to: 1-tv  2-disk  3-printer  1
enter the key (0=no processing):  32050
secret message processor at work. wait

type a quote, then a line of text


enter an empty line to quit
r30 SXz 'ywCUnSm ew
The new password is

type a quote, then a line of text
enter an empty line to quit
Lz23f5JoAm
grapefruit

type a quote, then a line of text
enter an empty line to quit


processing complete
<c>ontinue or <q>uit? c

SECRET MESSAGE PROCESSOR

READ TEXT FROM: 1-KEYBOARD  2-DISK  1
OUTPUT TO: 1-TV  2-DISK  3-PRINTER  1
ENTER THE KEY (0=NO PROCESSING):  114
SECRET MESSAGE PROCESSOR AT WORK. WAIT

TYPE A QUOTE, THEN A LINE OF TEXT
ENTER AN EMPTY LINE TO QUIT
r30 SXz 'ywCUnSm ew
4yt 1LL X2M7DV-C 99

type a quote, then a line of text
enter an empty line to quit
Lz23f5JoAm
U-ac-qKh99
```

Figure 17-2. Sample run of the Secret Message Processor (*continued*)

```
type a quote, then a line of text
enter an empty line to quit

Processing complete
<c>ontinue or <q>uit? q

ready.
```

Figure 17-2. Sample run of the Message Processor (*continued*)

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 SMP because the distribution of letters in its ciphertext is almost uniform. (See Table 17-3.)

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.

Table 17-3. Frequency Distribution of Characters in the Ciphertext

Plaintext	Key	Ciphertext
AAAAAAAAA	32050	0K059vXco
111111111	12345	BluvcbC2k
Joe Joe Joe	41200	B−f gw0 1KG

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 might be inferred, 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 SMP program. This time-consuming method requires the would-be code-breaker systematically to try different keys and see the results on the ciphertext.

In summary, the SMP 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.

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.

This 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.

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

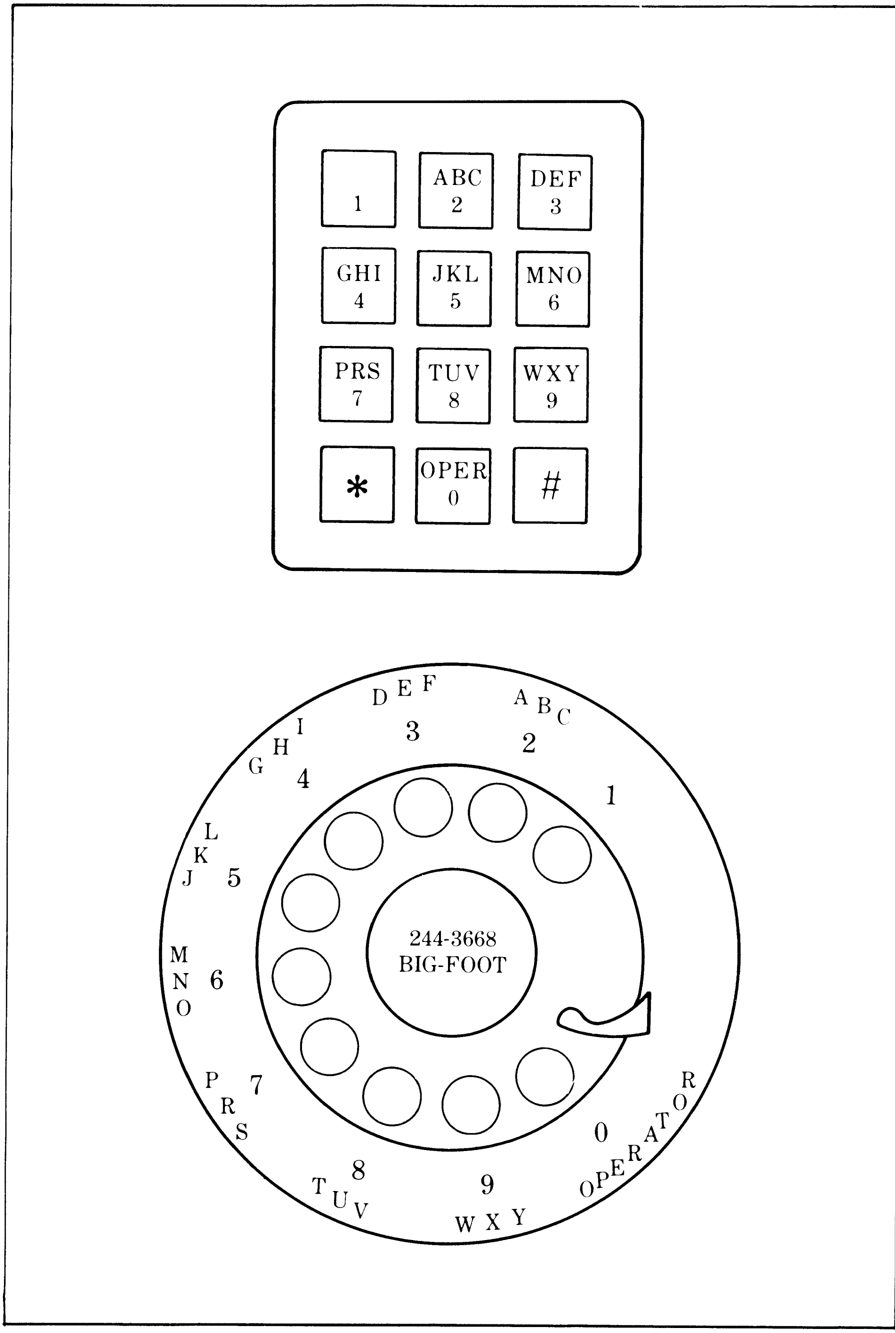


Figure 18-1. Pushbutton and rotary dial telephone faces

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.

—The Program

The first block sets up the program's constants:

```
1 CS$=CHR$(147): REM CLEAR SCREEN
2 RV$=CHR$(18): REM REVERSE PRINTING
3 NR$=CHR$(146): REM NORMAL PRINTING
4 S1$=" ": REM 1 SPACE INSIDE QUOTES
5 NU$="": REM NO SPACES INSIDE QUOTES
6 PR=4: REM PRINTER DEVICE NUMBER
10 MD=15
20 DIM K(MD)
30 P$="000111ABCDEFGHIJKLMNOPRSTUVWXY"
50 MG=2
```

PR is the printer device number. If your printer has a different number, change line 6 accordingly.

MD in line 10 is the maximum number of digits allowed in a phone number, not including 1's and 0's, which are not changed by the program. Line 20 creates an array to store the current base 3 number. Each element in the array K( ) corresponds to a base 3 digit. K(1) contains the least significant digit, and K(MD) stores the most significant digit.

The variable P$ in line 30 stores the letter triplets. Since there are no letter replacements for the numbers 0 and 1, the triplets 000 and 111 are used for these numbers respectively. When entering this line, be sure you leave out the letters Q and Z, which do not appear on the telephone dial. Line 50 determines how many spaces are used between each column when the names are printed.

Displaying the Menu

The next block prints a menu of options and gets your selection:

```
60 PRINT CS$;
70 PRINT
80 PRINT RV$;: REM REVERSE
90 PRINT SPC((40-17)/2);"BLAZING TELEPHONES"
100 PRINT NR$;: REM REVERSE OFF
110 PRINT
120 PRINT "1-CONVERT NAME TO NUMBER"
130 PRINT "2-CONVERT NUMBER TO NAME"
140 PRINT "3-QUIT"
150 PRINT
160 INPUT "SELECT 1, 2, OR 3? ";CH
170 IF CH<>1 AND CH<>2 AND CH<>3 THEN 110
180 ON CH GOTO 190,360,1090
```

The menu offers two options: (1) convert a “phone name” or alphabetic sequence into a telephone number, or (2) generate all possible alphabetic sequences for a given telephone number. The first option is useful if you are a businessperson looking for desirable phone numbers to request from the telephone company. The second option is for those who already have a number.

Converting a Name to a Phone Number

The following lines perform the name-to-number conversion:

```
190 PRINT
200 PN$=NU$
205 INPUT "ENTER NAME: ";PN$
210 IF PN$=NU$ THEN 190
220 FOR CN=1 TO LEN(PN$)
230 C$=MID$(PN$,CN,1)
240 Q0=1
250 Q1$=P$
260 Q2$=C$
270 GOSUB 1100
280 PS=QF
290 IF PS=0 THEN 320
300 PD=INT((PS-1)/3)
310 C$=CHR$(PD+48)
320 PRINT C$;
330 NEXT CN
340 PRINT
350 GOTO 70
```

PN$ stores the alphabetic sequence. The program examines each character C$ of the sequence. The subroutine called in line 270 searches for C$ inside the translation list P$. If the C$ is contained in P$, line 300 derives the corresponding telephone digit PD, and line 310 converts that number to its corresponding ASCII character C$. Line 320 prints the result.

After every character in the sequence has been examined, line 350 returns to the main menu.

Converting a Phone Number to a Name

The second option is more complicated. Here's the first block:

```
360 PRINT
370 PN$=NU$
375 INPUT "ENTER PHONE NUMBER: "; PN$
380 IF PN$=NU$ THEN 360
390 PL=LEN(PN$)
400 ND=0
410 FOR CN=1 TO PL
420 C$=MID$(PN$,CN,1)
430 IF C$>="2" AND C$<="9" THEN ND=ND+1
440 NEXT CN
450 IF ND>0 THEN 480
460 PRINT "NO TRANSLATABLE DIGITS FOUND."
470 GOTO 360
480 IF ND<=MD THEN 510
490 PRINT "TOO MANY DIGITS. MAX IS ";MD
500 GOTO 360
```

PN$ stores the phone number. Lines 400-440 count the number of translatable digits ND in PN$. (Translatable digits are numbers 2 through 9.) The program rejects PN$ if it contains fewer than 1 or more than MD translatable digits.

After confirming that phone number PN$ is acceptable for translation, the next block prompts you to specify the form for its voluminous output. For a seven-digit number, the program is going to generate as many as 2187 names. It is important to set up the output in a condensed yet readable format.

```
510 PRINT "OUTPUT TO: 1-TV  2-PRINTER"
515 OD=1
520 INPUT "SELECT 1 OR 2? ";OD
530 IF OD<>1 AND OD<>2 THEN 510
```

```
560 IW=PL+MG
570 PRINT "MAXIMUM LINE WIDTH (";IW;"-80)? "
575 LW=40
580 INPUT "RETURN=40: ";LW
610 IF LW<IW OR LW>80 THEN 570
620 IL=INT(LW/IW)
630 PRINT "PRINT HOW MANY LINES BEFORE PAUSING?"
635 NP=0
640 INPUT "RETURN=NO PAUSE): ";NP
660 IF NP<0 THEN 630
```

Lines 510-550 select the TV or printer. Lines 560-610 determine the number of names printed on each line. IW equals PL (the length of each name) plus MG (the number of spaces between names). Lines 570 and 580 prompt you to enter LW (the line width), which must be wide enough for a single name and at most 80 characters. When running the program with a printer for output, specify the widest line your printer can handle.

Lines 630-660 give you the option of having a pause after a specified number of lines are printed. If you are outputting to the C-64 display, specify a pause after each 24 lines.

Printing a Title

The following lines print a title on the display or printer:

```
670 IT=1
680 LN=1
690 IF OD=2 THEN OPEN 1,PR: CMD 1
700 PRINT INT(3↑ND+.5);" DISTINCT NAMES FOR ";PN$
710 PRINT
720 FOR TD=1 TO ND
730 K(TD)=0
740 NEXT TD
```

Lines 670 and 680 initialize the items-per-line counter and lines-per-page counter.

Line 690 begins routing output to the selected device, and line 700 prints the title. The expression 3^ND calculates the number of distinct names; ND is not the total number of digits, but the total number of *translatable* digits.

Lines 720-740 set all the base three digits to 0, the first key value used in converting the number to a name.

Generating a Name

The next lines produce a single name by applying the key value in K( ) to the number in PN$:

```
750 D=1
760 FOR CN=1 TO PL
770 C$=MID$(PN$,CN,1)
780 IF C$<"2" OR C$>"9" THEN 820
790 PD=VAL(C$)
800 C$=MID$(P$,PD*3+1+K(D),1)
810 D=D+1
820 PRINT C$;
830 NEXT CN
```

D is a pointer indicating which base 3 digit to use for the next digit in PN$. The loop from 760-830 examines each character of PN$, loading it into the variable C$. Line 780 determines whether C$ is a translatable digit. If C$ is translatable, lines 790 and 800 perform the translation on C$. Line 790 increments D—in effect pointing to the next digit of the base 3 key. If C$ is not translatable, it is printed "as is," and the pointer D is left unchanged.

The program continues this process until all the characters of PN$ have been processed.

Making Line and Page Breaks

Upon completion of the preceding block, the computer has printed a single name. The next block checks to see whether it's time to start a new line or to pause between "pages."

```
840 IF IT>=IL THEN 880
850 IT=IT+1
860 PRINT SPC(MG);
870 GOTO 970
880 IT=1
890 PRINT
900 IF NP>0 AND LN>=NP THEN 930
910 LN=LN+1
920 GOTO 970
930 LN=1
940 IF OD=2 THEN PRINT#1,
950 INPUT "PRESS RETURN TO CONTINUE ";EN$
960 IF OD=2 THEN CMD 1
```

Lines 840-870 insert a carriage return after IL names have been printed. Lines 880-960 insert a pause in the output after the specified number of lines NP.

At this point, the program has completed the process of converting a name, printing it, and adjusting the format.

Generating the Next Key

Now the program is ready to generate the next base 3 key:

```
970 DP=1
990 K(DP)=K(DP)+1
1000 IF K(DP)<=2 THEN 750
1020 K(DP)=0
1030 IF DP=ND THEN 1060
1040 DP=DP+1
1050 GOTO 990
```

First a general description of what's going on: each successive base 3 key is generated by adding 1 to the current value. To do this, the program mimics the manual method of adding 1. As you read the following steps, keep in mind that the program is using base 3 arithmetic, which allows only the digits 0, 1, and 2.

1. Set the current digit pointer to the least significant digit. In this program, that's defined as the *leftmost* digit (ordinarily the rightmost digit is the least significant).
2. Add 1 to the digit indicated by the digit pointer.
3. If the result is less than 3, the process is complete. Otherwise, set the digit to 0 and carry a 1 to the next step.
4. If the digit pointer is already at the most significant (that is, the *rightmost*) digit, there is no place to put the carry: the largest number possible for the number of digits available has already been generated, so the process is complete.
5. Otherwise, move the digit pointer to the next digit on the right, and go back to Step 2.

Figure 18-2 gives a few examples of the process.

Now back to the details of the program. In line 970, the digit pointer DP is set to 1, the least significant digit. Line 990 adds 1 to the corresponding base 3 digit. Line 1000 determines whether the result exceeds 2, necessitating a carry to the next digit position. If no carry results, the newest key is ready, so the program jumps back to line 750 to generate another name.

```
                          Carry   1
N       0000000           N       2110010
      + 1                       + 1
N + 1   1000000           N + 1   0210010

Carry   11                Carry   1111111    No more digits
N       2202222           N       2222222    available = Done
      + 1                       + 1
N + 1   0012222           N + 1   0000000
```

Figure 18-2. Samples of base 3 counting as performed by the program

If there is a carry, line 1020 sets the current digit to 0. Line 1030 determines whether any more digits are available to store the carry. If DP is less than ND, the program continues at line 1040, which increments the digit pointer and then continues with the addition process.

If DP equals ND, no more digits are available: that is, the last key in the series has been generated, so the number-to-name generation is complete. In that case, the following lines reroute the output to the display and jump back to the main menu:

```
1060 PRINT
1070 IF OD=2 THEN PRINT#1,: CLOSE 1
1080 GOTO 70
```

Ending the Program

There's one more line to the main program. It corresponds to option 3 (quit):

```
1090 END
```

String Search Subroutine

The following subroutine probably looks familiar; it is used in numerous programs throughout this book.

```
            BLAZING TELEPHONES

1-CONVERT NAME TO NUMBER
2-CONVERT NUMBER TO NAME
3-QUIT

SELECT 1, 2, OR 3?  1

ENTER NAME: BUSY BEE
2879 233

            BLAZING TELEPHONES

1-CONVERT NAME TO NUMBER
2-CONVERT NUMBER TO NAME
3-QUIT

SELECT 1, 2, OR 3?  1

ENTER NAME: NUMBER 1
686237 1

            BLAZING TELEPHONES

1-CONVERT NAME TO NUMBER
2-CONVERT NUMBER TO NAME
3-QUIT

SELECT 1, 2, OR 3?  2

ENTER PHONE NUMBER: 424-6245
OUTPUT TO: 1-TV  2-PRINTER
SELECT 1 OR 2?  1
MAXIMUM LINE WIDTH ( 10 -80)?
RETURN=40:  32
PRINT HOW MANY LINES BEFORE PAUSING?
RETURN=NO PAUSE):  4
 2187  DISTINCT NAMES FOR 424-6245

GAG-MAGJ  HAG-MAGJ  IAG-MAGJ
GBG-MAGJ  HBG-MAGJ  IBG-MAGJ
GCG-MAGJ  HCG-MAGJ  ICG-MAGJ
GAH-MAGJ  HAH-MAGJ  IAH-MAGJ
PRESS RETURN TO CONTINUE
GBH-MAGJ  HBH-MAGJ  IBH-MAGJ
GCBREAK IN 780
```

Figure 18-3. Sample run of Blazing Telephones

```
1100 QF=0
1110 IF Q0+LEN(Q2$)-1>LEN(Q1$) THEN RETURN
1120 IF MID$(Q1$,Q0,LEN(Q2$))=Q2$ THEN 1150
1130 Q0=Q0+1
1140 GOTO 1110
1150 QF=Q0
1160 RETURN
```

On entry to the subroutine, Q0 is the starting position for the search, Q1$ is the string to be searched, and Q2$ is the string to search for. On return from the subroutine, QF points to the starting position of Q2$ in Q1$. QF=0 indicates the string is not found.

—Running the Program

Figure 18-3 shows a sample run of the program. To be sure you have entered the program correctly, try to duplicate the results shown.

When using the number-to-name option, it is not necessary to process the entire number at once. You may find it helpful to enter only a part of the number at a time (for example, the initial three-digit extension 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, concentrate on the other portion.

If your number contains any 1's or 0's, it's a good idea to enter only the segments on either side of these digits. For example, given the number 665-8415, you should enter the number as 66584, which produces only 243 distinct names. Among them you'll find NOJUG, NOLUI ("no Louie"), and OOLUH. Now combine the names with the last two digits to get NOJUG-15, NOLUI-15, and OOLUH-15. All of these are more memorable than the original number sequence.

Who knows what bright new name may be hiding inside your telephone number?

Chapter 19

Nutritional Advisor

A one-ounce bag of potato chips provides 150 calories, 2 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.

—Program Operation

The program includes data about 48 foods commonly used as cooking ingredients. You can easily expand the list to include unusual ingredients that you use. For each food, the following need to be included:

1. Food name
2. Measurement unit
3. Calories
4. Protein (in grams)
5. Carbohydrates (in grams)
6. Fat (in grams)

Items 3 through 6 are based on one measurement unit of the ingredient. For instance, the sample entry

MILK, CUP, 165, 8, 12, 10

indicates that one cup of milk contains 165 calories, 8 g of protein, 12 g of carbohydrates, and 10 g of fat.

The program prompts you to list the ingredients of the recipe one at a time. If the ingredient you give is contained in the program's list, the program will name the appropriate measurement unit and ask you to specify the quantity used. For example, after you type "milk," the program will ask, "How many cups are used?"

If the ingredient you specify is not in the list, the program will tell you so and give you three options:

1. See food list
2. Enter data for *ingredient*
3. Enter a new ingredient name

Option 1 lets you check the list to see exactly how many foods are known. For example, if you specify flour as an ingredient, the program will print, "No data available on flour." Examine the food list and you'll see entries for whole wheat flour and white flour. Select option 3 and enter the appropriate ingredient—*exactly* as it is listed in the food list.

Option 2 lets you enter the correct information for an unlisted ingredient. For instance, if your recipe includes anchovies, you can type in the appropriate nutritional information taken from the package. However, information entered this way is not permanently stored in the list for use the next time you run the program. To do that, you must add the information for each data record in the program's DATA lines, as explained later.

After typing in all the ingredients, enter an empty line (press RETURN in response to the prompt, "Ingredient?"). The program will ask how many servings the recipe makes. Ordinarily, you should enter the number of people the recipe is intended to serve; however, for some recipes like those for breads or pies, you may want to know the nutritional makeup of the full recipe. In that case, enter 1.

Finally, the program gives you a nutritional analysis of a typical serving. Figure 19-1 shows a sample run of Nutritional Advisor.

—Program Listing

The first block prints a title and initializes the totals for calories (CA), protein (PR), carbohydrates (CB), and fats (FA).

```
1 CS$=CHR$(147): REM CLEAR SCREEN
2 RV$=CHR$(18): REM REVERSE PRINTING
3 NR$=CHR$(146): REM NORMAL PRINTING
4 S1$=" ": REM 1 SPACE INSIDE QUOTES
5 NU$="": REM NO SPACES INSIDE QUOTES
6 QT$=CHR$(34)
10 PRINT CS$
20 PRINT "THE NUTRITIONAL ADVISOR"
30 PRINT
40 PRINT "TYPE IN THE RECIPE"
50 PRINT "ONE INGREDIENT AT A TIME"
60 CA=0
70 PR=0
80 CB=0
90 FA=0
```

QT$ is a double quote character. CA, PR, CB, and FA are totals for various food components.

Entering an Ingredient

The next block prompts you to enter an ingredient name and then searches for that name in the food list:

```
100 PRINT
110 PRINT "TYPE AN EMPTY LINE FOR TOTALS"
120 PRINT "TYPE A SLASH (/) TO SEE FOOD LIST"
130 IG$=NU$
135 INPUT "NEXT INGREDIENT ";IG$
140 IF IG$=NU$ THEN 580
145 IF IG$="/" THEN 330
```

```
THE NUTRITIONAL ADVISOR

TYPE IN THE RECIPE
ONE INGREDIENT AT A TIME

TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT FLOUR
NO DATA AVAILABLE ON "FLOUR"
TYPE 1 TO SEE FOOD-LIST
     2 TO ENTER DATA FOR "FLOUR"
     3 TO ENTER A NEW INGREDIENT NAME.
 3

TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT WHITE FLOUR
HOW MANY CUP(S) ARE USED 1.75

TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT BUTTERMILK
NO DATA AVAILABLE ON "BUTTERMILK"
TYPE 1 TO SEE FOOD-LIST
     2 TO ENTER DATA FOR "BUTTERMILK"
     3 TO ENTER A NEW INGREDIENT NAME.
 1
MILK
WHIPPING CREAM
COTTAGE CHEESE
CHEDDAR CHEESE
CREAM CHEESE
EGGS
BUTTER
MARGARINE
VEGETABLE OIL
GROUND BEEF
CHICKEN
LAMB
HAM
COD
FLOUNDER
CRABMEAT
TUNA
GREEN SNAP BEANS
```

Figure 19-1. Sample run of Nutritional Advisor

```
GREEN LIMA BEANS
RED KIDNEY BEANS (CANNED)
BROCCOLI
CABBAGE
CARROTS
CAULIFLOWER
CELERY
CORN
MUCHROOMS
ONIONS
GREEN PEAS (CANNED)
POTATOES
TOMATOES (CANNED)
SPINACH
APPLES
BANANA
BLUEBERRIES (CANNED)
PEACHES (CANNED)
PINEAPPLE (CANNED)
RAISINS
CORN MEAL
WHITE FLOUR
WHOLE WHEAT FLOUR
BROWN RICE
WHITE RICE
NOODLES
OATMEAL
SUGAR
ALMONDS
WALNUTS
END

TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT MILK
HOW MANY CUP(S) ARE USED 1.5

TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT BUTTER
HOW MANY 1/4-LB STICK(S) ARE USED .33
```

Figure 19-1. Sample run of Nutritional Advisor (*continued*)

```
TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT EGGS
HOW MANY EGG(S) ARE USED 1

TYPE AN EMPTY LINE FOR TOTALS
TYPE A SLASH (/) TO SEE FOOD LIST
NEXT INGREDIENT
HOW MANY SERVINGS DOES THE RECIPE MAKE 6
EACH SERVING CONTAINS
 214.4  CALORIES
 6.5  GRAMS PROTEIN
 27.5  GRAMS CARBOHYDRATE
 8.5  GRAMS FAT

TYPE 1 TO ANALYZE ANOTHER RECIPE
     2 TO END.
 2
```

Figure 19-1. Sample run of Nutritional Advisor (*continued*)

```
150 RESTORE
160 READ N$,U$,N1,N2,N3,N4
170 IF N$="END" THEN 260
180 IF N$<>IG$ THEN 160
```

Line 135 gets the ingredient name you type in and stores it in IG$. Line 150 resets the DATA pointer so that the program starts searching for the ingredient at the beginning of the food list.

Line 160 reads a complete food "record" consisting of a name N$, measurement unit U$, calories N1, protein N2, carbohydrate N3, and fat N4.

Line 170 checks whether the last record has been encountered. The last data record *must* include END as the food name, and include dummy values for all the other items (see line 1210).

Line 180 compares the food name just read with the value stored in IG$. If they don't match, the program reads the next record.

Finding a Matching Food

The following lines are executed after the program finds a matching record in the food list:

```
190 PRINT "HOW MANY ";U$;"(S) ARE USED";
200 U=0
205 INPUT U
210 CA=CA+N1*U
220 PR=PR+N2*U
230 CB=CB+N3*U
240 FA=FA+N4*U
250 GOTO 100
```

Line 190 requests the quantity needed. When you provide that information (line 200), the program can compute the nutritional contribution of the given ingredient (lines 210-240).

Finding an Unknown Food

The lines that follow are executed when the program cannot find your ingredient in its food list.

```
260 PRINT "NO DATA AVAILABLE ON ";QT$;IG$;QT$
270 PRINT "TYPE 1 TO SEE FOOD-LIST"
280 PRINT "     2 TO ENTER DATA FOR ";QT$;IG$;QT$
290 PRINT "     3 TO ENTER A NEW INGREDIENT NAME."
300 S=1
305 INPUT S
310 IF S<>1 AND S<>2 AND S<>3 THEN 270
320 ON S GOTO 330,470,100
```

Lines 260-300 print the option list referred to previously and input your selection. Line 320 jumps to the program block corresponding to your selection.

Displaying the Food List

Here's the block that displays the food list:

```
330 LC=0
340 RESTORE
350 READ N$,U$,N1,N2,N3,N4
360 PRINT N$
370 IF N$="END" THEN 100
380 LC=LC+1
390 IF LC<24 THEN 350
400 LC=0
410 PRINT RV$;
420 INPUT "PRESS RETURN TO CONTINUE:";RT$
430 PRINT NR$;
460 GOTO 350
```

The variable LC counts the number of lines printed; after the twenty-fourth line, the program inserts a pause so you can read a full display before continuing.

Line 340 resets the data pointer to the start of the food list; line 350 reads a data record. Line 370 checks if it is the end-of-data record. If not, line 380 increments the lines-printed counter, and lines 390-430 insert a pause after every twenty-fourth line.

Adding a New Food

The following lines provide option 2 (enter unlisted data):

```
470 PRINT "ENTER MEASUREMENT UNIT FOR ";IG$
480 U$=""
485 INPUT U$
490 PRINT "CALORIES PER ";U$
500 N1=0
505 INPUT N1
510 PRINT "PROTEIN (G.) PER ";U$
520 N2=0
525 INPUT N2
530 PRINT "CARBOHYDRATE (G.) PER ";U$
540 N3=0
545 INPUT N3
550 PRINT "FAT (G.) PER ";U$
560 N4=0
565 INPUT N4
570 GOTO 190
```

The variables used in lines 480, 500, 520, 540, and 560 correspond to the variables used in the READ statements in lines 160 and 350. The program simply fills each variable using your keyboard inputs rather than reading them from the food list.

Line 570 causes the program to continue just as if the data had been read in from the food list.

Displaying the Results

The final block of the main program requests the number of servings, performs the final calculations, and prints the results.

```
580 PRINT "HOW MANY SERVINGS DOES THE RECIPE MAKE";
590 INPUT NS
600 IF NS<1 THEN 580
610 PRINT "EACH SERVING CONTAINS"
620 PRINT INT(CA/NS*10+.5)/10;" CALORIES"
```

```
630 PRINT INT(PR/NS*10+.5)/10;" GRAMS PROTEIN"
640 PRINT INT(CB/NS*10+.5)/10;" GRAMS
    CARBOHYDRATE
650 PRINT INT(FA/NS*10+.5)/10;" GRAMS FAT"
660 PRINT
670 PRINT "TYPE 1 TO ANALYZE ANOTHER RECIPE"
680 PRINT "      2 TO END."
690 INPUT S
700 IF S<>1 AND S<>2 THEN 670
710 ON S GOTO 10,720
720 END
```

The Food List

That's the end of the program logic, but the food list is still missing. The food list occupies 48 DATA lines from 730 through 1200. You can add or subtract items anywhere in this range. However, be sure that line 1210 remains the last record in the list. It is a special end-of-data record.

Here's the food list. Type it in very carefully, and be sure *not* to include any spaces that aren't shown here. It is particularly important not to include spaces after a food name. Otherwise, when you request a food type, you will have to include that trailing space or the program won't find it in the list.

```
730 DATA MILK,CUP,165,8,12,10
740 DATA WHIPPING CREAM,CUP,860,4,6,94
750 DATA COTTAGE CHEESE,CUP,240,30,6,11
760 DATA CHEDDAR CHEESE,1-INCH CUBE,70,4,0,6
770 DATA CREAM CHEESE,OZ,105,2,1,11
780 DATA EGGS,EGG,75,6,0,6
790 DATA BUTTER,1/4-LB STICK,800,0,0,90
800 DATA MARGARINE,1/4-LB STICK,806,0,0,91
810 DATA VEGETABLE OIL,TABLESPOON,125,0,0,14
820 DATA GROUND BEEF,LB,1307,112,0,91
830 DATA CHICKEN,LB,1326,114,0,91
840 DATA LAMB,LB,1675,107,0,75
850 DATA HAM,LB,1547,85,0,117
860 DATA COD,LB,777,128,0,23
870 DATA FLOUNDER,LB,914,137,0,37
880 DATA CRABMEAT,LB,480,75,5,11
890 DATA TUNA,LB,907,133,0,37
900 DATA GREEN SNAP BEANS,CUP,25,1,6,0
910 DATA GREEN LIMA BEANS,CUP,140,8,24,0
920 DATA RED KIDNEY BEANS (CANNED),CUP,230,15,42,0
930 DATA BROCCOLI,CUP,45,5,8,0
940 DATA CABBAGE,CUP,40,2,9,0
```

```
950 DATA CARROTS,CUP,45,1,10,0
960 DATA CAULIFLOWER,CUP,30,3,6,0
970 DATA CELERY,CUP,20,1,4,0
980 DATA CORN,CUP,170,5,41,0
990 DATA MUCHROOMS,1/2-CUP,12,2,4,0
1000 DATA ONIONS,CUP,80,2,18,0
1010 DATA GREEN PEAS (CANNED),CUP,68,3,13,0
1020 DATA POTATOES,MED.SIZE POTATO,100,2,22,0
1030 DATA TOMATOES (CANNED),CUP,50,2,9,0
1040 DATA SPINACH,CUP,26,3,3,0
1050 DATA APPLES,CUP,100,0,26,0
1060 DATA BANANA,MED.SIZE BANANA,85,0,23,0
1070 DATA BLUEBERRIES (CANNED),CUP,245,1,2,0
1080 DATA PEACHES (CANNED),CUP,200,0,52,0
1090 DATA PINEAPPLE (CANNED),SLICE,95,0,26,0
1100 DATA RAISINS,CUP,230,2,62,0
1110 DATA CORN MEAL,CUP,360,9,74,4
1120 DATA WHITE FLOUR,CUP,400,12,84,0
1130 DATA WHOLE WHEAT FLOUR,CUP,390,13,79,2
1140 DATA BROWN RICE,CUP,748,15,154,3
1150 DATA WHITE RICE,CUP,692,14,150,0
1160 DATA NOODLES,CUP,200,7,37,2
1170 DATA OATMEAL,CUP,150,5,26,3
1180 DATA SUGAR,CUP,770,0,199,0
1190 DATA ALMONDS,1/2-CUP,425,13,13,38
1200 DATA WALNUTS,1/2-CUP,325,7,8,32
1210 DATA END,"",0,0,0,0
```

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 a bit 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 from March 1920 through November 2009.

In addition to performing mental time traveling for fun, 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 information to and from disk, so you won't have to retype it every time you want another printout.

—Anatomy of a Calendar

The Time Machine program arranges the calendar in the traditional table of four to six rows by seven columns. The rows correspond to weeks, and the columns to days of the week. The cell where each row and column intersect may represent an actual date in the month, or it may be empty. Figure 20-1 shows a sample personalized calendar from the Time Machine.

How much information is needed to produce an accurate monthly calendar? Just two factors are involved: the number of days in the month and the weekday on which the month begins.

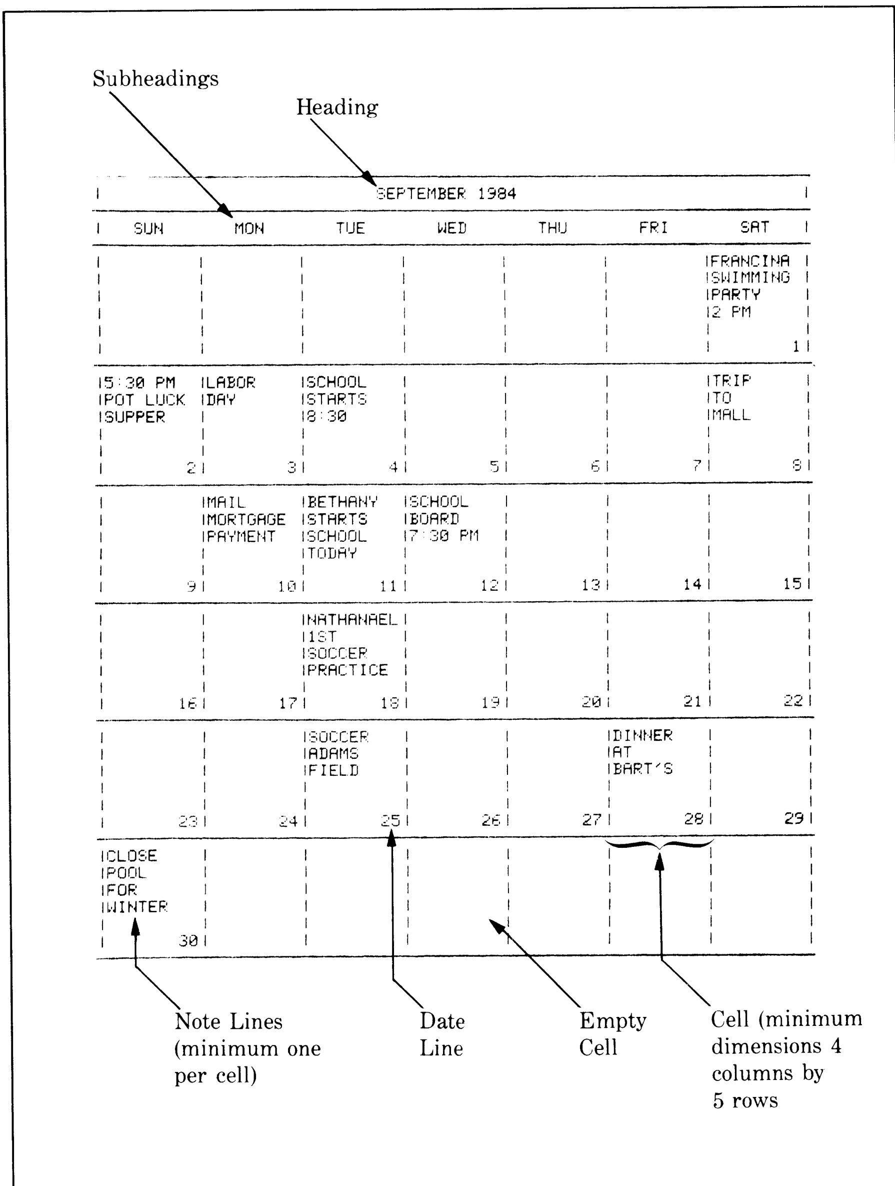


Figure 20-1. Sample personalized calendar

Initial Calculations

Finding the number of days in a month is a trivial exercise, even for a computer. February, of course, is a special case because its length depends on whether the year is a leap year. Leap years are those that are evenly divisible by 4, unless the year happens to be the first year of a new century, such as 1900 or 2000. A year that begins a new century is a leap year only if it is evenly divisible by 400. According to these rules, 1984 is a leap year because it can be divided evenly by 4, and 2000 is a leap year because it can be divided by 400.

Finding the weekday on which a month begins is more difficult. One method involves referring to three tables, each consisting of hundreds of numbers and letter codes. A simpler method uses a known *base date* and extrapolates forward from that date. For example, if we know that March 1, 1920, occurred on a Monday, we can calculate the day of the week for any subsequent date.

The Time Machine uses the latter method. Because of practical limitations in the precision of numbers in Commodore BASIC, the calculations are limited to a span of approximately 89 years.

Data Structures

Two arrays store the key information about each monthly calendar. The calendar array C(*cell number*) maps each day of the month onto its corresponding position among the 42 possible cells (6 weeks multiplied by 7 days per week equals 42). For instance, if the first of the month falls on a Saturday, C(7)=1, since that Saturday is in column 7.

After the number 1 is assigned to one of the first seven cells in C( ), all of the other numbers from 2 through the last day of the month are assigned in sequential order. For example, if C(7)=1, then C(8)=2, C(9)=3, and so forth. What about all the unused cells before the first and after the last day of the month? They are set to zero.

The array MS$(*date,note-line*) stores the reminder notes that you have assigned to certain days of the month. For instance, MS$(5,1) stores line 1 of the notes for the fifth day of the month; MS$(5,2) stores line 2 of the notes for the fifth of the month; and so forth. The notes are printed in the corresponding cell of the monthly calendar.

Since the notes are stored separately from the calendar mapping, it is possible to change months while retaining the notes. For example, you may have certain monthly obligations that remain the same from month to month (rent due on the first, for example). The program lets you recalculate C( ) for a different date while retaining the contents of MS$(,).

The Program

The first block defines four useful numeric functions:

```
10 DEF FN M1(X)=X-INT(X/4)*4
20 DEF FN M2(X)=X-INT(X/100)*100
30 DEF FN M3(X)=X-INT(X/400)*400
40 DEF FN M7(X)=X-INT(X/7)*7
```

All of the functions calculate X modulo N, which is the remainder of the quotient:

$$\frac{X}{N}$$

Function M1 calculates X modulo 4; M2 calculates X modulo 100; M3 calculates X modulo 400; and M7 calculates X modulo 7.

Storing Information About the Calendar

The next block of lines stores fundamental information about the 12 months and weeks of the year:

```
50 DIM MO$(12),MD(12),WY$(7),C(42)
60 FOR M=1 TO 12
70 READ MO$(M),MD(M)
80 NEXT M
90 FOR D=1 TO 7
100 READ WY$(D)
110 NEXT D
120 DATA JANUARY,31,FEBRUARY,28,MARCH,31,APRIL,30
130 DATA MAY,31,JUNE,30,JULY,31,AUGUST,31
140 DATA SEPTEMBER,30,OCTOBER,31,NOVEMBER,30,
    DECEMBER,31
150 DATA SUN,MON,TUE,WED,THU,FRI,SAT
```

MD$( ) stores the names of the months; MD( ), the number of days in each month; WY$( ), the abbreviated names of the days; and C( ), the number assigned to each of 42 possible calendar cells.

When typing in the DATA lines, take care not to add spaces before or after the commas since this will upset the calendar format.

Storing Miscellaneous Constants

The program needs a few other constants as well.

```
152 S1$=" ": REM 1 SPACE INSIDE QUOTES
153 NU$="": REM NO SPACES INSIDE QUOTES
154 QC$=S1$: QL=40: GOSUB 2370: SS$=QS$
160 DW=5
170 DL=5
180 PR=4: REM PRINTER DEVICE NUMBER
190 QT$=CHR$(34): REM QUOTE
200 CT=0
210 VW=80
220 MW=INT((VW-1)/7)
230 VL=66
240 ML=INT((VL-5)/6)
250 VL$=CHR$(125): REM VERTICAL LINE
260 HL$=CHR$(192): REM HORIZONTAL LINE
270 DIM MS$(31,ML-1)
```

DW and DL are preset values for cell width and cell length, respectively. The program gives you an opportunity to change these (reformat the calendar). PR is your printer's device number. Change it in line 180 if necessary. QT$ is the character for a double quote. This character is needed when storing data in a disk file.

VW is the maximum calendar width measured in actual characters; from this, the program derives MW, the maximum width of a calendar cell. VL is a maximum calendar length in lines, and ML is the derived maximum length of a cell. Even though your C-64 display allows only 40 characters by 25 lines, you can design a larger calendar format for output to a printer; VW=80 and VL=66 correspond to standard 80-column, 66-line printer paper. CT is the display slot number.

VL$ and HL$ are the characters used for vertical and horizontal lines, respectively. Finally, MS$(31,ML−1) is the array that stores the daily reminder notes.

—Inputting the Month and Year

The following short block gets initial values for the calendar month and calendar format:

```
280 GOSUB 450
290 GOSUB 1170
```

The subroutine called in line 280 asks you to enter the month and year of the desired calendar and then fills in the array C( ) according to certain date calculations. The subroutine called in line 290 prompts you to specify the output format of the calendar.

The Main Menu

The following lines print the main menu:

```
300 PRINT
310 PRINT "1-PRINT MINIATURE CALENDAR"
320 PRINT "2-PRINT FULL-SIZE CALENDAR"
330 PRINT "3-EDIT CALENDAR NOTES"
340 PRINT "4-REFORMAT FROM DISK OR KEYBOARD"
350 PRINT "5-CHANGE CALENDAR MONTH"
360 PRINT "6-SAVE NOTES AND FORMAT ON DISK"
370 PRINT "7-QUIT"
380 PRINT
390 INPUT "SELECT 1-7: ";F
400 IF F<1 OR F>7 THEN 300
410 IF F=7 THEN END
420 ON F GOSUB 820,1690,2070,1170,450,990
430 GOTO 300
```

The seven options available are: 1 - PRINT MINIATURE CALENDAR, 2 - PRINT FULL-SIZE CALENDAR, 3 - EDIT CALENDAR NOTES, 4 - REFORMAT FROM DISK OR KEYBOARD, 5 - CHANGE CALENDAR MONTH, 6 - SAVE NOTES AND FORMAT ON DISK, and 7 - QUIT.

Line 420 calls the subroutine corresponding to your selection. Upon completion of the subroutine, line 430 jumps back to the start of the main menu.

Change Calendar Month

Here's the first part of the subroutine to change the calendar month:

```
450 PRINT
460 PRINT "INPUT THE CALENDAR MONTH AS MONTH,YEAR."
470 PRINT "FOR EXAMPLE: 1,1984 FOR JANUARY 1984."
480 PRINT "VALID MONTHS ARE 3,1920 TO 11,2009"
490 PRINT
500 PRINT "NOW TYPE IN THE MONTH AND YEAR"
510 INPUT M,Y
520 IF Y<1920 OR Y>2009 THEN 450
530 IF Y=1920 AND M<3 THEN 450
540 IF Y=2009 AND M>11 THEN 450
550 IF M<1 OR F>12 THEN 450
```

Lines 520-550 ensure that the month and year you enter are within the acceptable ranges.

The next part of the subroutine initializes the calendar framework C( ):

```
560 H$=MO$(M)+STR$(Y)
570 LH=LEN(H$)
580 PRINT
590 PRINT "CONSTRUCTING A CALENDAR FOR"
600 PRINT H$;" . . .
610 FOR CN=1 TO 42
620 C(CN)=0
630 NEXT CN
```

H$ is the calendar heading (for instance, "January 1984"). Lines 610-630 set every cell in the calendar to 0, indicating that no dates are assigned yet.

At this point, the program needs to know how many days are between the base date (March 1, 1920) and the first day of the month you have selected. The next lines make that calculation:

```
640 Y1=Y-1920
650 IF M<=2 THEN 680
660 M1=M-3
670 GOTO 700
680 M1=M+9
690 Y1=Y1-1
700 JD=INT(1461*Y1/4)+INT((153*M1+2)/5)
```

Lines 640-690 convert the actual month M and year Y into relative values M1 and Y1 based on the starting point of March 1, 1920. Y1 is years elapsed, and M1 is months elapsed within the last year. For instance, for calendar month May 1970, we get Y1=Y−1920=50 and M1=M−3=2, indicating a span of 50 years and two months.

Line 700 calculates the number of days represented by that span. The expression INT(1461*Y1/4) gives the total number of days in Y1 years. The expression INT(153*M1+2)/5) gives the number of days in M1 months. Adding these two expressions gives the total number of days JD in the span from March 1, 1920, up to the first day of the month for the specified calendar month.

(The calendar logic summarized here is presented in greater detail in *Dr. Dobbs Journal #80,* June 1983, page 66, "Julian Dates for Microcomputers," by Gordon King.)

Given the number of days elapsed, the program can complete the calendar calculations.

```
710 WD=FN M7(JD+1)
720 IF (FN M1(Y)=0 AND FN M2(Y)<>0) OR
    FN M3(Y)=0 THEN 750
730 LP=0
740 GOTO 760
750 LP=1
760 LD=MD(M)
770 IF LP=1 AND M=2 THEN LD=29
780 FOR D=1 TO LD
790 C(D+WD)=D
800 NEXT D
810 RETURN
```

Line 710 computes WD=JD+1 modulo 7, the weekday on which the first of the month falls. The value returned ranges from 0 to 6. Zero represents Sunday, 1 Monday, and so forth.

Lines 720-770 determine whether it is a leap year and then set the last day of the month LD. Line 770 adjusts LD for February in a leap year.

Lines 780-800 number the calendar cells with the appropriate day numbers. Line 810 returns to the main program.

Print Miniature Calendar

The next block of lines prints a miniature calendar, as shown in the sample run later in this chapter (Figure 20-2).

Here are the line that print the miniature calendar:

```
820 GOSUB 2240
830 PRINT TAB((20-LH)/2+1);H$
840 FOR D=1 TO 7
850 PRINT LEFT$(WY$(D),2); S1$;
860 NEXT D
870 PRINT
880 FOR D=1 TO LD+WD
890 IF C(D)<>0 THEN 920
900 PRINT S1$;S1$;
910 GOTO 930
920 PRINT RIGHT$(S1$+STR$(C(D)),2);
930 PRINT S1$;
940 IF FN M7(D)=0 THEN PRINT
950 NEXT D
960 PRINT
970 GOSUB 2330
980 RETURN
```

The subroutine called in line 820 lets you select the output device (display or printer). The loop from 840 to 860 prints the abbreviated day-of-week headings.

The loop from 880 to 950 prints the dates in the appropriate column locations for each week of the calendar. Line 940 starts a new row of dates after every week is completed.

The subroutine called in line 970 resets the display as the output device, and line 980 returns to the main program.

Save Calendar to Disk

The next block saves the calendar notes and format in a disk file:

```
990 PRINT
1000 YN$="N"
1010 INPUT "VIEW DISK DIRECTORY (Y/N)? ";YN$
1020 IF YN$="Y" THEN GOSUB 2900
1030 FO$=NU$
1040 INPUT "NAME OF OUTPUT FILE ";FO$
1050 IF FO$<>NU$ THEN 1070
1060 PRINT "DATA NOT STORED"
1065 RETURN
1070 OPEN 2,8,3,"@0:"+FO$+",SEQ,WRITE"
1072 IF ST=0 THEN 1080
1074 PRINT "DISK ERROR. DATA NOT STORED"
1076 RETURN
1080 PRINT#2, CW
1090 PRINT#2, CL
1100 FOR CN=1 TO 31
1110 FOR L=1 TO CL-1
1120 PRINT#2, QT$;MS$(CN,L);QT$
1130 NEXT L,CN
1140 CLOSE 2
1150 PRINT "DATA STORED IN ";FO$
1160 RETURN
```

Line 1040 prompts you to enter a file name. Line 1070 creates the output file, erasing any existing file by that name. If a disk-related error occurs, the program will inform you and return to the menu. After evaluating the source of the error, rerun the program and try again.

Lines 1080 and 1090 print the cell width and cell length, and the loop from 1100 to 1130 prints the entire contents of the notation array MS$(,). Every line of text is printed inside quotes so that the program will be able to retrieve the lines correctly, even if they contain commas or colons.

Reformat Calendar

Here is the subroutine that lets you reformat a calendar by loading one from disk or by typing in new specifications. The subroutine starts by printing a short menu:

```
1170 PRINT
1180 PRINT "CALENDAR FORMAT:"
1190 PRINT "1-ENTER FROM KEYBOARD"
1200 PRINT "2-LOAD FROM DISK FILE"
1210 INPUT "SELECT 1 OR 2 ";CD
1220 IF CD<>1 AND CD<>2 THEN 1170
1230 IF CD=2 THEN 1470
```

The menu gives you two options: enter specifications from keyboard, or load them from disk. These lines handle keyboard input:

```
1240 PRINT
1250 PRINT "ERASING OLD NOTES..."
1260 FOR CN=0 TO 31
1270 FOR L=1 TO ML-1
1280 MS$(CN,L)=NU$
1290 NEXT L,CN
1300 PRINT
1310 PRINT "ENTER CELL WIDTH (4-";MW;")"
1320 PRINT "(RETURN=";DW;")"
1330 CW=DW
1340 INPUT CW
1370 IF CW<4 OR CW>MW THEN 1300
1380 PRINT
1390 PRINT "ENTER CELL LENGTH (2-";ML;")"
1400 PRINT "(RETURN=";DL;")"
1410 CL=DL
1420 INPUT CL
1450 IF CL<2 OR CL>ML THEN 1380
1460 GOTO 1610
```

The loop from 1260 to 1290 erases the previous contents of the notations array MS$(,). Lines 1300-1450 prompt you to enter the cell width and cell length. DW and DL are default values given if you press RETURN without typing in any specifications.

The next lines handle disk input:

```
1470 YN$="N"
1480 INPUT "VIEW DISK DIRECTORY (Y/N)? ";YN$
1490 IF YN$="Y" THEN GOSUB 2900
1500 FI$=NU$
1505 INPUT "NAME OF INPUT FILE ";FI$
1510 IF FI$<>NU$ THEN 1525
1515 PRINT "DATA NOT LOADED"
1520 GOTO 1170
1525 OPEN 2,8,3,FI$+",SEQ,READ"
1530 INPUT#2,CW
1540 INPUT#2,CL
1550 FOR CN=1 TO 31
1560 FOR L=1 TO CL-1
1570 INPUT#2,MS$(CN,L)
1580 NEXT L,CN
1590 CLOSE 2
1592 IF ST=0 OR ST=64 THEN 1600
1594 PRINT "DISK ERROR. DATA NOT LOADED."
1596 GOTO 1170
1600 PRINT "DATA LOADED FROM ";FI$
```

Line 1505 prompts you to enter a file name, and line 1525 attempts to open the specified file for inputting.

Lines 1530 and 1540 input the cell width and cell length, and the loop from 1550 to 1580 inputs every element of the note array MS$(,). If a disk error occurs, lines 1594-1596 inform you and restart the calendar reformatting routine.

After keyboard or disk specification is complete, the next lines perform a few other calculations related to calendar format:

```
1610 PW=CW*7+1
1620 NB=CL-1
1630 OF=INT((CW-3)/2)
1640 QL=PW
1650 QC$=HL$
1660 GOSUB 2370
1670 R$=QS$
1680 RETURN
```

PW is the rightmost column position of the calendar in its new format. NB is the number of message lines available within each cell. OF is the offset required to center each day-of-week name within its column. Lines 1640-1670 store a string of horizontal lines in R$, forming a horizontal line of length PW.

Line 1680 returns to the main program.

Full-Size Calendar Printout

The calendar printout subroutine is the longest part of the program, so it will be broken into smaller segments. The first part prints the title (month and year) and day-of-week headings:

```
1690 GOSUB 2240
1700 PRINT R$
1710 TB=INT((PW-2-LH)/2)
1720 PRINT VL$;SPC(TB);H$;SPC(PW-LH-TB-2);VL$
1730 PRINT R$
1740 FOR DA=1 TO 7
1750 IF DA=1 THEN PRINT VL$;
1760 IF DA<>1 THEN PRINT S1$;
1770 PRINT SPC(OF);WY$(DA);SPC(CW-4-OF);
1780 NEXT DA
1790 PRINT VL$
```

Line 1690 lets you select the output device (display or printer). Throughout this printing section, the SPC function is used instead of TAB to advance the print position. SPC(*n*) outputs a string of *n* spaces.

Line 1720 prints the heading centered over the calendar, and line 1770 prints each day-of-week name centered over the corresponding column in the calendar.

The next part of the subroutine prints the note lines (there are NB note lines in each cell):

```
1800 PRINT R$
1810 LW=-INT(-(LD+WD)/7)
1820 FOR W=1 TO LW
1830 FOR L=1 TO NB
1840 FOR DA=1 TO 7
1850 PRINT VL$;
1860 DN=DA+(W-1)*7
1870 M$=MS$(C(DN),L)
1880 PRINT M$;SPC(CW-LEN(M$)-1);
1890 NEXT DA
1900 PRINT VL$
1910 NEXT L
```

Line 1810 calculates the number of rows (Sunday through Saturday cycles) that must be printed to cover the first of the month through the last of the month. Depending on the length of the month and on where the first day of the month falls in the week, from four to six rows may be required.

Line 1820 starts a loop that counts through each of the LW rows.

Each calendar row consists of NB note lines followed by a date line (see Figure 20-1). Line 1830 starts a loop that counts through the NB note lines. Line 1840 starts a loop that counts through the seven days. Line 1870 gets the appropriate note for each numbered cell, and line 1880 prints it in the cell.

Line 1890 selects the next day, and line 1910 selects the next line. After all of the note lines have been printed for a given calendar week, the program moves on to the next block, which prints the date lines:

```
1920 FOR DA=1 TO 7
1930 PRINT VL$;
1940 DN=DA+(W-1)*7
1950 IF C(DN)<>0 THEN 1980
1960 PRINT SPC(CW-1);
1970 GOTO 2000
1980 DT$=STR$(C(DN))
1985 DT$=RIGHT$(DT$,LEN(DT$)-1)
1990 PRINT SPC(CW-LEN(DT$)-1);DT$;
2000 NEXT DA
2010 PRINT VL$
```

The loop from 1920 to 2000 counts through seven days of the week. DN is the cell number, which ranges from 1 to 42. Line 1950 determines whether that cell is numbered. If it is not numbered, line 1960 fills the cell with spaces. Otherwise, lines 1980-1990 put the number into the cell in right-justified form (the number is always printed at the extreme right side of the cell).

Here is the final part of the calendar-printing subroutine:

```
2020 PRINT R$
2030 NEXT W
2040 PRINT
2050 GOSUB 2330
2060 RETURN
```

Line 2020 prints a horizontal rule, and line 2030 advances to the next week number. After all LW weeks are printed, the calendar is complete. The subroutine called in line 2050 resets the display as the output device, and line 2060 returns to the main program.

Edit Calendar Notes

The next group of lines lets you add or change the contents of any numbered calendar cell.

```
2070 PRINT
2080 PRINT "ADD NOTES TO WHICH DATE? (1-";LD;")"
2090 SD=0
2100 INPUT "ENTER 0 TO QUIT: ";SD
2110 IF SD<1 OR SD>LD THEN RETURN
2120 PRINT
2130 PRINT "ENTER ";NB;" NOTE LINES.
     MAX LENGTH=";CW-1
2140 FOR L=1 TO NB
2150 PRINT "LINE ";L
2160 TX$=NU$
2170 INPUT TX$
2180 IF LEN(TX$)>CW-1 THEN TX$=LEFT$(TX$,CW-1)
2190 MS$(SD,L)=TX$
2200 NEXT L
2210 PRINT
2220 PRINT "TEXT STORED."
2230 GOTO 2070
```

Lines 2080-2110 prompt you to select a date. (Typing a 0 ends the editing session and returns you to the main menu.) The loop from 2140 to 2200 inputs the NB lines required to fill that cell. Line 2180 ensures that the lines you enter will fit into the cell by chopping off extra characters on the right side of the line. Line 2190 stores each line in the appropriate location in MS$(,).

Auxiliary Subroutines

The next subroutine lets you select an output device:

```
2240 PRINT
2250 PRINT "OUTPUT TO: 1-TV  2-PRINTER"
2260 OD=1
2270 INPUT "SELECT 1 OR 2: ";OD
2280 IF OD<>1 AND OD<>2 THEN 2240
2290 IF OD=1 THEN RETURN
2300 OPEN 1,PR
2310 CMD 1
2320 RETURN
```

If you select printer output, lines 2300-2310 route the output to that device. (PR, set in line 180, must be the device number of your printer.)

Here are the lines that restore output back to the video display:

```
2330 IF OD=1 THEN RETURN
2340 PRINT#1,
```

```
2350 CLOSE 1
2355 OD=1
2360 RETURN
```

The next subroutine generates a repeating string of characters:

```
2370 QS$=NU$
2380 FOR QQ=1 TO QL
2390 QS$=QS$+QC$
2400 NEXT QQ
2410 RETURN
```

On entry to the subroutine, QL is the length of the string and QC$ is the repeating character. On return from the subroutine, QS$ consists of QL of character QC$.

Disk Directory

The final subroutine reads the disk directory and prints all file names on the screen. There is a slight delay while the computer sorts through extraneous information.

```
2900 PRINT "LOADING DIRECTORY..."
2910 OPEN 1,8,4,"$,SEQ,READ"
2920 IW=0
2930 IF ST<>0 THEN 3040
2940 GET#1,A$
2950 IF LEN(A$)=0 THEN 2920
2960 IF A$>CHR$(31) AND A$<CHR$(122) THEN 3010
2970 IF IW=0 THEN 2930
2980 IW=0
2990 PRINT
3000 GOTO 2930
3010 IF IW=0 THEN WL=1
3015 IW=1
3020 PRINT A$;
3023 WL=WL+1
3024 IF WL<17 THEN 3030
3026 PRINT
3028 WL=0
3030 GOTO 2930
3040 CLOSE 1
3050 PRINT
3060 RETURN
```

—Using the Program

Figure 20-2 shows a sample run of the program using the display for output. You should be able to get the same results.

Once you have become familiar with using the program with output to the Commodore display, try printing some calendars. Experiment with different line-space values and character sets that may be available with your printer. Check in your printer manual for ways of selecting alternate print modes.

You may also find it convenient to change certain default settings of the program, in particular

- Cell width DW$ (line 160)
- Cell length DI $ (line 170)
- Maximum total width of calendar VW (line 210)
- Maximum total length of calendar VL (line 230)
- Character used for vertical lines VL$ (line 250)
- Character used for horizontal lines HL$ (line 260).

```
INPUT THE CALENDAR MONTH AS MONTH,YEAR.
FOR EXAMPLE: 1,1984 FOR JANUARY 1984.
VALID MONTHS ARE 3,1920 TO 11,2009

NOW TYPE IN THE MONTH AND YEAR
 12 , 1984

CONSTRUCTING A CALENDAR FOR
DECEMBER 1984 . . .

CALENDAR FORMAT:
1-ENTER FROM KEYBOARD
2-LOAD FROM DISK FILE
SELECT 1 OR 2  1

ERASING OLD NOTES...

ENTER CELL WIDTH (4- 11 )
(RETURN= 5 )
 5
```

Figure 20-2. Sample run of the Time Machine

```
ENTER CELL LENGTH (2- 10 )
(RETURN= 5 )
 4

 1-PRINT MINIATURE CALENDAR
 2-PRINT FULL-SIZE CALENDAR
 3-EDIT CALENDAR NOTES
 4-REFORMAT FROM DISK OR KEYBOARD
 5-CHANGE CALENDAR MONTH
 6-SAVE NOTES AND FORMAT ON DISK
 7-QUIT

 SELECT 1-7:  1

 OUTPUT TO: 1-TV  2-PRINTER
 SELECT 1 OR 2:  1
     DECEMBER 1984
 SU MO TU WE TH FR SA
                    1
  2  3  4  5  6  7  8
  9 10 11 12 13 14 15
 16 17 18 19 20 21 22
 23 24 25 26 27 28 29
 30 31

 1-PRINT MINIATURE CALENDAR
 2-PRINT FULL-SIZE CALENDAR
 3-EDIT CALENDAR NOTES
 4-REFORMAT FROM DISK OR KEYBOARD
 5-CHANGE CALENDAR MONTH
 6-SAVE NOTES AND FORMAT ON DISK
 7-QUIT

 SELECT 1-7:  3

 ADD NOTES TO WHICH DATE? (1- 31 )
 ENTER 0 TO QUIT:  17

 ENTER  3  NOTE LINES. MAX LENGTH= 4
 LINE  1
 LM'S
 LINE  2
 AT
```

Figure 20-2. Sample run of the Time Machine (*continued*)

```
LINE  3
7:00

TEXT STORED.

ADD NOTES TO WHICH DATE? (1- 31 )
ENTER 0 TO QUIT:  5

ENTER  3  NOTE LINES. MAX LENGTH= 4
LINE  1
PAY
LINE  2
RENT
LINE  3
$370

TEXT STORED.

ADD NOTES TO WHICH DATE? (1- 31 )
ENTER 0 TO QUIT:  21

ENTER  3  NOTE LINES. MAX LENGTH= 4
LINE  1
XMAS
LINE  2
VAC.
LINE  3
!!!

TEXT STORED.

ADD NOTES TO WHICH DATE? (1- 31 )
ENTER 0 TO QUIT:  0

1-PRINT MINIATURE CALENDAR
2-PRINT FULL-SIZE CALENDAR
3-EDIT CALENDAR NOTES
4-REFORMAT FROM DISK OR KEYBOARD
5-CHANGE CALENDAR MONTH
6-SAVE NOTES AND FORMAT ON DISK
7-QUIT

SELECT 1-7:  2

OUTPUT TO: 1-TV  2-PRINTER
SELECT 1 OR 2:  1
```

Figure 20-2. Sample run of the Time Machine (*continued*)

```
 ----------------------------------------------------------
|                     DECEMBER 1984                        |
 ----------------------------------------------------------
| SUN     MON     TUE     WED     THU     FRI     SAT |
 ----------------------------------------------------------
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |     1 |
 ----------------------------------------------------------
|       |       |       |PAY    |       |       |       |
|       |       |       |RENT   |       |       |       |
|       |       |       |$370   |       |       |       |
|     2 |     3 |     4 |     5 |     6 |     7 |     8 |
 ----------------------------------------------------------
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|     9 |    10 |    11 |    12 |    13 |    14 |    15 |
 ----------------------------------------------------------
|       |LM'S   |       |       |       |XMAS   |       |
|       |AT     |       |       |       |VAC.   |       |
|       |7:00   |       |       |       |!!!    |       |
|    16 |    17 |    18 |    19 |    20 |    21 |    22 |
 ----------------------------------------------------------
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|    23 |    24 |    25 |    26 |    27 |    28 |    29 |
 ----------------------------------------------------------
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|       |       |       |       |       |       |       |
|    30 |    31 |       |       |       |       |       |
 ----------------------------------------------------------

1-PRINT MINIATURE CALENDAR
2-PRINT FULL-SIZE CALENDAR
3-EDIT CALENDAR NOTES
4-REFORMAT FROM DISK OR KEYBOARD
5-CHANGE CALENDAR MONTH
6-SAVE NOTES AND FORMAT ON DISK
7-QUIT
```

Figure 20-2. Sample run of the Time Machine (*continued*)

```
SELECT 1-7:  6

VIEW DISK DIRECTORY (Y/N)? N
NAME OF OUTPUT FILE DECEMBER

1-PRINT MINIATURE CALENDAR
2-PRINT FULL-SIZE CALENDAR
3-EDIT CALENDAR NOTES
4-REFORMAT FROM DISK OR KEYBOARD
5-CHANGE CALENDAR MONTH
6-SAVE NOTES AND FORMAT ON DISK
7-QUIT

SELECT 1-7:  7


READY.
```

Figure 20-2. Sample run of the Time Machine (*continued*)

—Index

THE C-64 PROGRAM FACTORY

Here's a book that will put your C-64 to work right now as an entertainer, puzzle generator, teacher, and creative assistant.

Written by George Stewart, author of POPULAR COMPUTING's column "The Program Factory," these programs can easily be keyed into your computer.

This collection includes many entertaining programs—
- The Matchmaker
- The Codebreaker
- Blackjack '84
- Poetry Generator
- The Time Machine

Beginners will enjoy quick access to these programs, and experienced users who would like to understand how the programs work can learn from the explanations that accompany each program.

The C-64 Program Factory is ideal for any C-64 user looking for programs that provide hours of fun and learning!

McGraw-Hill

ISBN 0-88134-150-9