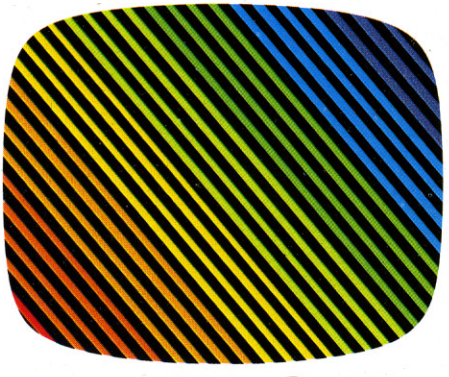
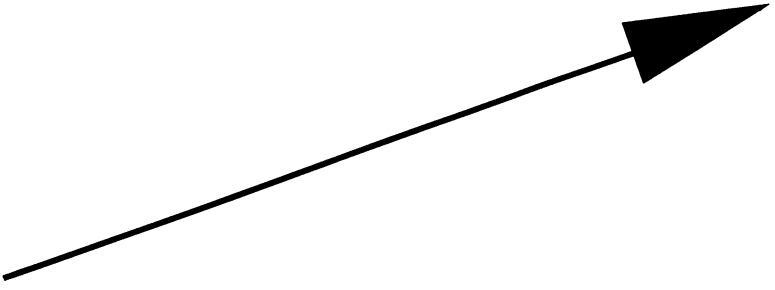
ZAPPERS

HAVING FUN PROGRAMMING AND PLAYING 23 GAMES FOR THE TI-99/4A



Henry Mullish and Dov Kruger



Zappers

Having Fun Programming and Playing 23 Games for TI-99/4A

HENRY MULLISH
and
DOV KRUGER

COMPUTER BOOK DIVISION
SIMON & SCHUSTER, INC.
NEW YORK

Published by the Computer Book Division/
Simon & Schuster, Inc.
Simon & Schuster Building
Rockefeller Center
1230 Avenue of the Americas
New York, New York 10020

Designed by Irving Perkins Associates
Manufactured in the United States of America

1 3 5 7 9 10 8 6 4 2

Library of Congress Cataloging in Publication Data
Mullish, Henry.
Zappers: having fun programming and playing 23 games
for TI-99/4A.

1. Computer games. 2. TI 99/4A (Computer)—Programming. 3. Basic (Computer program language) I. Kruger, Dov. II. Title.
GV1469.2.M85 1984 794.8 84-1219
ISBN 0-671-49862-1

CONTENTS

INTRODUCTION

According to various surveys aimed at owners of home or personal computers, the major use to which these incredible machines are being put is that of game playing. This is not surprising, since game playing on a home computer is both convenient and inexpensive. In contrast to arcade video games, playing games on a home computer has various advantages. For one thing, the whole family can participate in the action. For another, the home computer user can modify his games to suit his whims, while the arcade player is a hostage to the machine—although he or she can battle with the preset program of the machine, there is no way that the game can be changed in even the slightest degree.

Each of the games illustrated in this book has been written in the most popular of all home computer languages—BASIC, a language which is increasingly being taught in elementary schools as well as most institutions of higher learning across the nation. Readers of this book may very well already have some knowledge of the BASIC language. Such readers might be tempted, and indeed are encouraged, to amend the programs wherever they deem it desirable. However, for those who do not possess the necessary programming skills, a list of proposed modifications is presented at the end of each program description. In this way, the games can be continually updated to retain their freshness. In order to cater to as wide an audience as possible, various levels of difficulty are suggested, thereby enabling

each member of the family to participate in one way or another.

Among the numerous benefits that accrue to the game player are a heightening of intellectual skills and a strengthening of the powers of coordination. Not only are games of space-age excitement included, such as "Robot Attack," "Lander," and "Snazzle," but also games of a pragmatic nature, such as "Calendar," which permits the user to have the computer display a full-month calendar for any month of any year of his or her choice. The "Morse Code" program will assist a beginner in learning the Morse code quickly and in an unusually pleasant environment. Many of the other programs are TI-99/4A versions of popular classical games, such as "Hangman," "Blackjack," "Roulette," "Tic Tac Toe," and "Flip-a-Disk," among others.

THE HISTORY AND DEVELOPMENT OF THE TI-99/4A

In July of 1978, Texas Instruments released a brand-new product—indeed, a whole line of products, including a computer and a monitor (a specially designed, high-resolution TV-type display device used specifically with computers) together with expansion interfaces that allowed the machine to grow substantially with the user's needs. Despite all its considerable capabilities, the machine (called in those days the TI-99/4) was not particularly successful in the marketplace, not only because few programs were available for it but also because its price was too high. In 1980, TI responded to the situation by upgrading the keyboard (which was difficult for many people to use effectively), renamed the computer the TI-99/4A, and then hired Bill Cosby, the noted American comedian, to star in its commercials. At the same time, Texas Instruments drastically cut the price of the computer to make it competitive with comparable computers in the marketplace.

As a result of these steps, sales of the 99/4A exploded to the extent that, by mid-1983, over a million models were shipped. In fact, by the end of that year, 150,000 a month were being sold. This is largely due to the fact that the price

of the TI-99/4A today is under $100.00—making it one of the most powerful and flexible computers available in this price range. It comes complete with a full-stroke keyboard (far easier to type on than the cheaper membrane keyboard) and a wide variety of "ports" (the computer's windows to the outside world) allowing for powerful expansion capabilities. Even without these added features the TI-99/4A represents the state of the art in microcomputers, providing the user with computational punch, which was previously available only to large corporations, academic institutions, and governmental agencies.

The standard 99/4A can display output in 24 rows of 32 columns each. Visual effects are enhanced by the 16-color capability which is an integral part of the version of BASIC that comes with the machine, allowing for easy use of this most desirable feature with only simple programming. It has two cassette ports to allow it more flexibility when permanently storing programs and data. In addition to all these features, a port is dedicated for communication with two joysticks to allow for the creation of interactive, fast-action game playing. A connection to an expansion box is built in to allow for communication with peripherals such as printers, modems, and many others. The version of BASIC the 99/4A uses has been augmented to make control of these devices simple.

Aside from controlling the various accessory equipment, the version of BASIC supported by the TI-99/4A is superior to that found on many other machines. Writing and editing programs is fast and simple. With the advanced BASIC module, even more capabilities are gained. In this book, however, we shall assume that the reader has only the TI-99/4A with 16K (16,000 characters or bytes) of memory and a cassette recorder together with a connecting cable.

SETTING UP

In order to be able to properly enjoy your computer, it has to be set up correctly to insure maximum ease of operation with a minimum of problems. If treated with care, the

TI-99/4A should last a very long time and will provide an unlimited source of enjoyment and learning.

The first consideration is the location of the machine. Since the circuitry of the computer generates an appreciable amount of heat, blocking the cooling vents may permanently damage it, requiring expensive repairs. Therefore, the computer should always be placed on a flat, nonmetallic surface with unimpeded ventilation. It should be placed in an open location—it shouldn't be enclosed in any way. Because the machine gets warm enough on its own, it is a good idea to locate it away from direct sunlight. Since the computer is itself an electronic device, placing it too close to another electronic device such as a television set runs the risk of mutual interference between the two machines. Therefore, it is suggested that the computer not be placed on top of a television set.

Human comfort is another factor to consider. If you are going to look at a screen for an extended period of time, the screen should be free of any reflected glare and away from direct bright lights. But it is suggested that there still be some light in the room beyond that provided by the television set.

GETTING READY FOR ACTION

Once a convenient site for the computer has been selected, there are a few simple cable connections that have to be made. The first is the connection to the television set or the video monitor to display the output produced by the computer. For most people the TV set is the only option available because they don't have a monitor. If a TV set is used, it must be connected to the computer with the TI Video Modulator, which is included with the purchase of the computer. Check the *User's Reference Guide* for the proper way to make the connection. The round end of the cable should be plugged in the socket on the back left of the computer. The end of the cable that carries the video modulator should be attached to the VHF terminals of the television set. On the flat top surface of the modulator is a

switch labeled "TV and Antenna." When you wish to use the computer, this switch should be set to "TV." On the side of the video modulator from which the cable exits is a smaller switch that switches between channels 3 and 4. It should be set to the channel that does not reach your area; if it is set incorrectly, the computer is forced to compete with the local TV station, making the picture less clear.

Of course, in order to function, the computer must be supplied with a source of electricity. The power cable should be plugged into the back of the computer. Before plugging in the cable to the wall, be sure that the computer is switched off. There is a sliding ON/OFF switch on the front right of the computer. By making sure that this switch is in the OFF position, possible damage is prevented from the surge of current that occurs when the computer is first plugged in.

Now, with the other end plugged into the nearest convenient wall socket, switch the computer and television set on. So long as the television is set to channel 3 or 4 (whichever one is appropriate for your locality) you should see a most attractive and colorful picture on your screen. This is what Texas Instruments calls the "master computer title screen." If, for some reason, this does not appear on the screen, first look to see whether the red light that is located immediately to the left of the ON/OFF switch is on. If it is not, the probable explanation is that the wall socket from which you are attempting to draw power is dead. It is possible that it is under the control of a wall switch. Such wall sockets should be avoided (if at all possible) because the computer might be unintentionally switched off in the middle of a heavy session at the machine. This could result in the infuriating situation of wasting all your hard work and having to key in your program again from scratch. In the unlikely event that the red light is on but the proper display does not appear, switch off the computer immediately and review the steps that have been outlined so far. If you cannot detect anything wrong in your setup, you are advised to seek professional assistance.

While we now explain how to connect a tape cassette

recorder, you should temporarily switch off the TV and computer. In general, it is a good habit to switch off every peripheral and the computer itself when plugging in an additional device.

Now that the computer and TV are switched off, the time has come to try and connect the tape cassette recorder to the computer. A cassette recorder is essential for saving programs once they have been typed into the computer. Sometimes a program will be quite long and it will take a considerable amount of time to enter it into the computer. Once the power is switched off, all the contents of memory are irretrievably lost. This means that, if the same program is to be run again, it must be retyped from the beginning. Since it would be very tedious and time consuming to have to retype large programs over and over again, computers such as the 99/4A are designed so that you can store programs in certain peripherals made for this purpose. Cassette tapes are the least expensive and most popular way of permanently storing programs and data. In this book we assume that the reader has a cassette recorder. A disk drive is also available for the 99/4A but although it is far more efficient and faster than a cassette recorder, it is considerably more expensive, and most people do not own one.

When connecting a cassette recorder to the TI-99/4A, you must bear in mind that not all recorders are identical and many are incompatible with the computer. In order to communicate with the computer, the recorder must have at least two jacks—MICrophone and EXTernal speaker or EARphone jack. These are the connections through which the computer communicates directly with the tape recorder, and without these nothing can be saved. Additionally, it is beneficial to have a REMote jack, because it gives the computer more direct control over the recording and playback processes.

First it should be said that the cable that is required to connect a cassette recorder to the computer is not included in the purchase of the computer and so must be purchased separately, from a computer dealer who carries the TI line

of equipment. The double cassette cable (which permits not just one but two tape recorders to be connected to the computer) currently sells for approximately $15. It is not necessary to get the double cable, but the additional cost over the single cable unit is only about $5, and the extra capability may well be useful sometime in the future.

Now take the cable (single or double) and plug it into the back right of the computer, adjacent to the power cable socket. On the other end of the cable there are three plugs leading from red, white, and black wires. (If you have the double cable, use the end labeled "1.") The red wire should be connected to the MICrophone socket and the white wire is connected to the EARphone or EXTernal speaker socket. If the recorder has a REMote socket, plug the black wire into it. It is not necessary to use this wire, however, and many cassette recorders will not work properly with it.

When all the above steps have been taken, the TI-99/4A is ready for action. Without further ado, you can turn it on, get into BASIC by pressing any key and then the 1 key, and type in one of the programs included in this book.

HOW TO TYPE IN PROGRAMS

The computer programs included in this book have been designed specifically for the TI-99/4A. As a result, they take the best possible advantage of its considerable music and graphics capabilities, as well as its particular version of BASIC. Because the programs were made with the TI display in mind, the output produced by the programs fits perfectly into the available screen, twenty-eight characters on a line. In order to avoid a line break in the middle of a word, extra spaces are sometimes added before a word to "push" it onto the next line. Multiple or awkward spacing is indicated by a triangle (▲), which corresponds to one stroke on the **SPACE BAR.** Likewise, words that are run together with no space between them should be typed as is —the line break will fall exactly between them. There is a single space before the closing quotation mark in most

INPUT statements and in many PRINT statements; again, type the line exactly as it appears. The spacing within quotes is for formating only; the program will run without it.

Keep in mind that BASIC (like any other computer language) is totally unforgiving as far as the details of its instructions are concerned. One mistyped character and the program will not work properly. Because of this, it is a good idea to remember that any program you type in will probably not work perfectly the first time. Don't make the mistake of blaming the computer. If, after a long, fruitless search for an error you are still unable to find it, don't stalk off in utter disgust. Take a rest, but not before you have saved the program. The odds are that the error will be easily recognized later on, when you return to it with a fresh mind.

The actual mechanics of typing in programs on the TI are quite simple. All the special characters, such as ?, _, [,], and ", are obtained by holding down the special key labeled **FCTN** and pressing the appropriate key. The two arrows (on the keys labeled **S** and **D**) allow the programmer to go back and forth within a line to correct mistyped characters, should this be necessary—and there is no question that it will become necessary. After each line is correctly typed, it is sent to the computer by pressing the **ENTER** key. The flashing box called the "cursor" disappears from the screen for a short time, during which you cannot type. When it returns, the line will have been placed in memory and you may then type the next line.

EDITING A PROGRAM

It is somewhat comforting to note that it is impossible to do any physical damage to the computer by making errors in typing in a program. Indeed, the system provides an easy way to correct any such errors. The act of correcting mistyped lines within a program is called "editing."

The simplest method of editing is by replacing the entire

line in question. This may be done by simply typing the number of the line you want to correct and then retyping the line. Since each program line is stored in the order of its line number, the old line is automatically replaced.

This method is not well-suited to the most common mistake—mistyping just one character. In such a case, it is not necessary to retype the whole line; it is much faster and far less tedious to merely correct the one character involved. This may be done by means of the EDIT command. If, for example, line 155 was typed

```
155 PRIMT "HI THERE"
```

(where the BASIC command PRINT is misspelled), all you need do is to type

```
EDIT 155
```

and, magically, line 155 appears on the screen ready to be corrected. Using the left and right arrow keys, move the blinking cursor to the site of the error and type the correction. Once this is accomplished, hit the **ENTER** key and the amended line replaces the old (incorrect) one.

There is yet another way of editing a line. It involves typing the line number followed by the up or down arrow (obtained by holding down the **FCTN** key and pressing the key marked **E** or **X**). This will cause the line to be displayed as with the EDIT command, and you may proceed to edit the line as before.

Oftentimes you will want to insert (add) a character in the middle of a line. This is done by moving the cursor to the appropriate position in the line and pressing the **INS**ert key. This involves holding down the special key **FCTN** and pressing the key marked **2.** After this, any characters typed are inserted. (They do not write over the rest of the line but "push" it over to the right.) In a similar way, characters may be deleted (erased) by moving the cursor to the desired character with the arrow keys, holding down **FCTN,** and pressing the key marked **1** (this is the **DEL**ete function). TI

provides a plastic strip that, when placed above the top row of the keyboard, indicates clearly the action of each of the editing keys.

Occasionally, you might want to abandon typing a line if there are too many typos. There are two ways of doing this —both are equally acceptable. One way is to use the **ERASE** key (hold down the **FCTN** key and press the key marked **3**). The other way is to use the **CLEAR** key (**FCTN-4**) which does not erase the line from the screen, but nevertheless ignores the line.

Once you have typed in a program and would like to check it for accuracy, it may be displayed on the screen by typing the LIST command. This "lists" each instruction of the program in ascending order of line number—regardless of the order in which they were actually typed in. That is to say, if you type in the lines

```
100 PRINT "HELLO"
120 PRINT "GOODBYE"
110 PRINT "HOW ARE YOU?"
```

the program will list in the order of their line number:

```
100 PRINT "HELLO"
110 PRINT "HOW ARE YOU?"
120 PRINT "GOODBYE"
```

It is still a good idea to type in the lines in number order, not only because it is so easy to forget skipped lines, but also because TI BASIC has a special facility to help you when typing in programs. As you may already know, each BASIC instruction must be preceded by a line number. Typing in these line numbers, of course, takes time just like everything else. However, by typing the command NUM (for automatic line NUMbering) the computer will automatically supply a line number beginning with 100 and going up in steps of 10. For this reason, each of the programs listed in this book begins with line 100 and increases in steps of 10. The way to stop the computer from printing out the line

numbers is to press **ENTER** without typing anything else, or you can hit **FCTN-4.** If you wish to resume automatic line numbering in the middle of a program, say with line 280, then type

```
NUM 280
```

and the line numbers will continue with line 280 and on in steps of 10.

Once a program has been typed into the computer it must be executed in order for it to produce any results. This is done by means of the RUN command. Assuming (somewhat optimistically) that the program worked the way it was designed to, you might wish to save it permanently on cassette tape so that you could recall it any time later. This is done by typing the command

```
SAVE CS1
```

and following the detailed instructions that automatically appear on the screen. Later on, you might want to load in the program to run it again. This is done by typing in the command

```
LOAD CS1
```

and again following the instructions on the screen.

Using the above as a guide, why don't you try your hand at typing in the first program? The game that the program simulates is probably familiar to you. It is a TI-99/4A version of a number-guessing game. Since this is one of the smaller of the programs, it is recommended that you begin with this one and wait until you have gained a little experience before you attempt some of the more exotic programs that appear later in the book. Good luck!

PROGRAM

"GUESS MY NUMBER"

THE GAME

The game "Guess My Number" permits the user to specify a range of numbers from which the computer picks a number out of its hat, so to speak. The wider the range selected, the greater the number of guesses that will be necessary before the "hidden" number is arrived at. With each guess, the computer responds with one of three answers: TOO SMALL, TOO LARGE, or CORRECT, YOU WIN! The number of guesses allotted is determined mathematically by the computer and is based on the range that the user selects. If the correct answer has not been selected in the allotted number of guesses, the computer prints out SORRY, YOU LOSE. THE ANSWER WAS . . ., followed by the random number that the computer selected.

```
100  CALL CLEAR
110  PRINT "THIS GAME IS GUESS MY NUMBER"
120  PRINT "YOU SPECIFY THE RANGE OF"
130  PRINT "NUMBERS AND I'LL PICK
     A▲▲▲▲▲RANDOM ONE AND ASK YOU TO▲▲▲GUESS
     WHAT IT IS."
140  PRINT
```

```
150 PRINT
160 PRINT "ENTER THE RANGE"
170 INPUT "FROM -->":LOW
180 INPUT "TO   -->":HI
190 RANGE=HI-LOW+1
200 RANDOMIZE
210 X=INT(RND*RANGE)+LOW
220 P2=INT(LOG(RANGE)/LOG(2))+1
230 PRINT ::"YOU HAVE";P2;"GUESSES"::
240 FOR I=1 TO P2
250 PRINT ::
260 INPUT "PLEASE ENTER YOUR GUESS: ":GUESS
270 IF GUESS=X THEN 360
280 IF GUESS<X THEN 390 ELSE 410
290 NEXT I
300 PRINT ::"SORRY, YOU LOSE."
310 PRINT "THE ANSWER WAS ";X
320 PRINT ::"DO YOU WANT TO PLAY AGAIN?"
330 INPUT "(Y-YES, N-NO): ":MORE$
340 IF SEG$(MORE$,1,1)="Y" THEN 100
350 END
360 PRINT ::"CORRECT! YOU WIN."
370 PRINT "YOU GUESSED IT IN ";I;" TRIES"
380 GOTO 320
390 PRINT ::"TOO SMALL"
400 GOTO 290
410 PRINT ::"TOO LARGE"
420 GOTO 290
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	clears screen
110–150	displays introduction to the game
160	prompts the user
170–180	requests the user to enter the lower and upper limits of the range

▲This symbol represents a stroke on the **SPACE BAR.**

190 computes the range
200 reseeds the random generator
210 sets the variable X to a random integer in the specified range
220–230 sets the variable P2 to the number of guesses allowed and prints it out
240–290 within this loop each guess is accepted and is tested against the computer's selection, providing branching to the different parts of the program depending on the outcome
300–310 provides for a failing response
320–330 requests another round
340 tests the user's response that was stored in MORE$
350 stops execution of the program if the response is negative
360–380 provides the response for a successful round
390–400 user's guess is too small
410–420 user's response is too large

PROGRAM DESCRIPTION

After the screen has been cleared (this is the role of the CALL CLEAR instruction) the user is asked to enter the low and high ends of the range. These values are stored in the variables LOW and HI. The range (stored in RANGE) is computed by subtracting LOW from HI. The amount "1" is added to this result, since the range is inclusive. For example, if the range is from 2 to 5, there are four possible responses—2, 3, 4, and 5. Therefore, we subtract the low from the high ($5-2=3$) and add 1 to get the number of integers (positive whole numbers) within the range.

The RANDOMIZE instruction in line 200 ensures that each time the program is run, the random number generated is unpredictable. If the statement is removed from the program (you might want to try this and see for yourself) the same random numbers are generated for the same values of the range each time the program is started. In other words, if you run the program many times and ask for a

range of from 1 to 100, you will get the same random number generated each time—not a very good practice in a guessing game. This happens only when the program is run from scratch, however—not when you go around again within the program.

In line 210, the variable X is set to a random number within the range that is currently stored in RANGE. This is accomplished by means of the RND function which, when invoked, returns a value between 0 and 1 (not including 1). When that number is multiplied by RANGE its integer value is taken by means of the function called INT. The value of LOW is then added and the result is stored in X. Here is a brief explanation of the logic involved. Suppose the range selected is 1 to 100 and the random number selected by the computer (using the RND function) happens to be 0.23984563. Multiplying this number by the range 100 and taking the integer portion gives us the value 23. To this result is added the value of LOW (which is 1). Thus, we arrive at the random number 24, which is subsequently stored in X. This is the number that the user has to guess.

Because the wider the range, the greater the number of guesses that should be allowed, a little mathematics is used to arrive at a fair estimate. The variable P2 simply contains the number of times that the range can be split into two parts. If you are mathematically inclined, you might recognize that this is nothing more than the logarithm of RANGE to the base 2, with 1 added to the result. The user is then advised of what this calculated value is.

Within the FOR . . . NEXT loop, the user's guess is compared with X and an appropriate response is displayed. Once the game has ended the user is asked if he or she wants to play again. The response of Y (or any word beginning with Y) sends control back to line 100, which again clears the screen and restarts the game. Whatever the response is, it is stored in the string variable MORE$. The instruction

```
340  IF SEG$(MORE$,1,1) = "Y" THEN 100
```

compares the SEGment of the string MORE$, starting with

its 1st character and continuing for a length of 1 character. If this segment is equal to the letter Y, control is transferred immediately to line 100. If, on the other hand, the segment is anything other than Y, control falls down to the next statement and the program ends.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. You might want to change the number of guesses allowed to make the game either easier or more difficult. This may be done by changing the value of P2 as it is assigned in line 220.
2. Provide a check to insure that no guess is allowed which is outside of the range, since it would obviously be a mistake.
3. Provide a check to insure that once the range is narrowed down, any guess outside of this limited range is not accepted.

PROGRAM

2

"TYPING TEST"

PURPOSE

The following program has been devised to familiarize you with the keyboard of the TI-99/4A and to improve both your reflexes and typing skills at the same time. This is a timed-response quiz, which means that if you don't type the correct key within a certain period of time (a time interval which gets smaller with every level) you are scored as having missed the key. The game starts off so slowly that you may easily be led to believe that the quiz is a breeze. But as the levels progress, you'll realize that nothing could be further from the truth, and even the most skilled typists will have a very hard time keeping up, especially at level 10 (the most difficult).

```
100  REM TYPING TEST
110  CALL CLEAR
120  FOR I=1 TO 10
130  J=4000
140  CALL SOUND(50,J,0)
150  J=J*.75
160  IF J>1000 THEN 140
170  CALL SCREEN(I+4)
```

```
180 PRINT "ALL HANDS TO TYPING STATIONS":::
190 NEXT I
200 FOR L=1 TO 10
210 CALL CLEAR
220 PRINT ::"LEVEL";L::
230 CALL SOUND(250,262,2,330,2,392,2)
240 CALL SOUND(500,262,0,330,0,392,0)
250 CALL SOUND(500,262,2,349,2,440,2)
260 CALL SOUND(500,262,2,330,2,392,2)
270 CALL SOUND(500,247,2,349,2,392,2)
280 CALL SOUND(800,262,2,330,2,392,2)
290 TL=-(400/L)*(L<6)-(1000/(L*L))*(L>5)
300 RIGHT=0
310 FOR Q=1 TO 10
320 FOR T=1 TO 200
330 NEXT T
340 RANDOMIZE
350 CHAR$=CHR$(INT(RND*26)+65)
360 PRINT :::
370 PRINT "TRY #";Q
380 PRINT CHAR$
390 FOR T=1 TO TL
400 CALL KEY(3,A,X)
410 IF X THEN 490
420 NEXT T
430 PRINT "SORRY, YOU EXCEEDED THE
    TIMELIMIT"
440 NEXT Q
450 PRINT ::"YOUR SCORE IS:";RIGHT*10;"%"::
460 INPUT "DO YOU WANT ANOTHER QUIZ▲▲▲▲(Y/
    N) ":QUERY$
470 IF SEG$(QUERY$,1,1)="N" THEN 590
480 IF SEG$(QUERY$,1,1)="Y" THEN 550 ELSE
    460
490 IF CHR$(A)<>CHAR$ THEN 600
```

Type words that are run together just as they appear here and in other listings; don't add a space.

```
500  RIGHT=RIGHT+1
510  PRINT "CORRECT!"
520  CALL SOUND(200,440,0,220,0)
530  CALL SOUND(200,880,0,110,0)
540  GOTO 440
550  IF RIGHT<7 THEN 210
560  NEXT L
570  CALL CLEAR
580  PRINT "CONGRATULATIONS. YOU ARE NOWA
     GRANDMASTER OF TYPING."::::::::::
590  END
600  PRINT "WRONG"
610  CALL SOUND(700,-6,0)
620  FOR S=1 TO 5
630  CALL SCREEN(2)
640  CALL SCREEN(1)
650  CALL SCREEN(4)
660  NEXT S
670  GOTO 440
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark statement
110–190	prints the heading by using the sound and color features
200–560	the main loop, in which typing quizzes are generated and the level of difficulty is increased from 1 to 10.
210–280	prepares the screen for each level and plays the introductory music
290	sets the time limit for the particular level
300	sets to 0 the number of questions answered correctly
310–440	this loop generates a ten-question quiz
320–330	this loop creates a small time delay for audio-visual effect

340 reseeds the random number generator
350 sets the variable CHAR$ to a random letter A-Z
360–380 prints three blank lines and the question number, followed by the character to be typed
390–420 this loop is executed until the time limit expires
400–410 scans the keyboard for a typed character and, if found, transfers control to the routine which tests whether it is correct or not
430 displays the "time-limit exceeded" message, if required
450 prints final score for each level
460–480 tests if the player wants another quiz
490 routine to check if the typed character is equal to the computer's random one
500–540 if typed character is correct, adds 1 to score, prints CORRECT!, generates winning sound, and returns to the main routine
550 if score < 70% then the same level is taken over again
560 continues in the main loop and goes on to the next level
570–590 winning message is displayed and the program ends
600–670 prints WRONG and generates losing sound and some video effects before returning to the main routine

PROGRAM DESCRIPTION

After the screen has been cleared, the variable J is initialized to 4000 within a loop extending from line 120 to 190. This value of J is used to generate a tone whose frequency starts out at 4000 Hz. The sound is emitted by means of the statement in 140

```
CALL SOUND(50,J,0)
```

which specifies that the sound should last for fifty units of time (50/1000ths of a second) at a frequency of J Hz, at the

loudest volume (0; the softest volume is 30). J is then reduced to three fourths of its value, and as long as it is greater than 1000, control is sent back to line 140, which causes a note of that frequency to be emitted. As soon as the value of J is no longer greater than 1000, the color of the screen is set by means of the CALL SCREEN statement. A message is then displayed in line 180 and the whole process is repeated ten times.

In lines 230 to 280 each CALL SOUND statement is used to produce not one, but three voices. The first number in each statement is common to each tone and is the length of the tone in milliseconds. However, when written in this fashion, the length applies to each of the separate tones generated in the statement, so the remaining numbers are tones followed by their respective volumes. TL, which stands for time limit, stores the value calculated by the expression. This is arranged so that, as the level increases, the time limit is correspondingly shortened, with the net effect that the game becomes increasingly difficult.

The variable RIGHT, which keeps a count of the correctly answered questions, is initialized to 0 before the loop in which the index Q (for question) is entered. To allow the user some time to prepare before each question, an empty loop is repeatedly executed (200 times) thus providing a convenient time delay. After the RANDOMIZE statement, the variable CHAR$ is set to a random letter from A to Z. This is accomplished by generating a random number from 65 to 90 (which is the computer's internal representation of the letters of the alphabet) and converting these numeric equivalents (called ASCII codes) to their respective characters by means of the CHR$ function.

After printing out the number of the try, the computer-selected character is displayed. Then the timed response loop (the FOR . . . NEXT loop extending from 390 to 420) is entered. The keyboard is then scanned by means of the CALL KEY statement, which checks to see if a character has been typed, but continues execution of the program whether or not this is true. The first number in the CALL

KEY statement represents the keyboard mode. Depending on the application, the keyboard may be considered as a single unit or two adjacent "mini-keyboards." Mode 3 is the former, the standard keyboard arrangement. As soon as any character is typed, its ASCII code is stored into the variable A and the variable X—the status variable—is set to 1, indicating that a key was pressed. If not, the variable A contains the value −1 and the variable X the value 0. A test is then made to see whether X is 0 or not. If it is not, that is a sure sign that something—we have no idea what—has been typed in, and control is sent to line 490. If nothing has been typed and the time limit is exceeded, a message to this effect is printed and the quiz proceeds to the next question. If some character was typed in, it is tested against CHAR$ in line 490. If they do not agree, control is sent to line 600, which prints out the message WRONG and rocks the screen with a dazzling display of color and a mini explosion before going to the next question. If they agree, 1 is added to the variable RIGHT, the message CORRECT! is printed out, and a victory roll is sounded before continuing with the next question.

At the end of ten questions the percentage score of the player is displayed in line 450. The player is then asked if another round is desired. The response is stored in QUERY$ and the first character is tested in line 470 against the letter N. If a match is found, the program is immediately terminated. If the first letter of the response stored in QUERY$ is equal to Y (meaning that another round is requested) control is sent to line 550. If this is not the case, then it is clear that the response did not start with either an N or Y. Thus an error was committed and so the ELSE clause in line 480 sends control to line 460, the INPUT statement. The user is again asked to respond and the same checking procedure occurs. In line 550, the number of correct questions scored in the quiz is compared to 7 (the lowest passing score). If it is less than 7, the player must retake the quiz at the same level, having failed in his first attempt. Otherwise, control drops to the NEXT L state-

ment in line 560 and the level is advanced by 1. At the end of ten levels, a winning message is printed out and the game ends.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The time limit may be modified to allow for easier or harder levels of play, although the game is already very difficult indeed at the top levels.
2. The program could be redesigned to allow for a string of random characters which would have to be typed within a certain period of time. This would make for a considerably more challenging test. Of course, in view of the fact that more time is required to type in a greater number of letters, the time limit would have to be increased.
3. The number of questions in a quiz can be increased or decreased at will by changing the final value of Q in the loop at line 310. Should you decide to do this, the calculation of the percent of correctly answered questions as scored in line 450 must be amended.
4. The passing grade may be changed by using a different constant in line 550. For example, in order to make the passing grade 50 percent line 550 may be changed to

```
IF RIGHT < 5 THEN 210
```

PROGRAM

3

"ARITHMETIC QUIZ"

PURPOSE

You can relax a little on this quiz because the questions are not timed at all. It is a test of your arithmetic ability, one in which your skills of addition, subtraction, multiplication, and division are challenged. The quiz is devised so that all the correct answers are integers. Don't be fooled into thinking that for this reason there is no real challenge, however. It still isn't that easy to score well—unless, of course, you cheat by using pencil and paper or a calculator.

```
100  CALL CLEAR
110  PRINT "QUIZ ALERT! REV UP
     YOUR▲▲▲▲▲BRAIN!":::::::::::
120  FOR I=1 TO 6
130  FOR J=2 TO 10 STEP 2
140  CALL SOUND(50,-4,0,(3340-J*110),0,
     (1620-J*110),2,J*110,0)
150  CALL SCREEN(J)
160  NEXT J
170  NEXT I
180  CALL SCREEN(13)
190  CALL CLEAR
```

```
200 RANDOMIZE
210 PRINT "THIS IS A TEST"
220 PRINT "OF YOUR ARITHMETIC ABILITY"
230 PRINT
240 PRINT "1 ... ADDITION"
250 PRINT "2 ... SUBTRACTION"
260 PRINT "3 ... MULTIPLICATION"
270 PRINT "4 ... DIVISION"
280 PRINT
290 RIGHT=0
300 INPUT "WHICH ONE? ":CHOICE
310 IF (CHOICE<1)+(CHOICE>4)+(CHOICE<>
    INT(CHOICE))THEN 300
320 CALL CLEAR
330 FOR Q=1 TO 10
340 PRINT :"QUESTION #";STR$(Q)::
350 ON CHOICE GOSUB 430,490,550,610
360 NEXT Q
370 PRINT ::"YOUR SCORE IS";RIGHT*10;"%"::
380 INPUT "ANOTHER QUIZ? ":QUERY$
390 IF SEG$(QUERY$,1,1)="Y" THEN 180
400 IF SEG$(QUERY$,1,1)<>"N" THEN 380
410 CALL CLEAR
420 END
430 REM ADDITION ROUTINE
440 A=INT(RND*100)+1
450 B=INT(RND*100)+1
460 ANSWER=A+B
470 PRINT A;"+";B;"=";
480 GOTO 670
490 REM SUBTRACTION ROUTINE
500 A=INT(RND*100)+1
510 B=INT(RND*100)+1
520 ANSWER=A-B
530 PRINT A;"-";B;"=";
540 GOTO 670
550 REM MULTIPLICATION ROUTINE
560 A=INT(RND*25)+1
```

```
570  B=INT(RND*25)+1
580  ANSWER=A*B
590  PRINT A;"*";B;"=";
600  GOTO 670
610  REM DIVISION ROUTINE
620  A=INT(RND*25)+1
630  B=INT(RND*25)+1
640  ANSWER=A
650  A=A*B
660  PRINT A;"/";B;"=";
670  REM GET THE RESPONSE
680  INPUT RESPONSE
690  IF RESPONSE=ANSWER THEN 730
700  CALL SOUND(400,110,0,120,0,130,0,-4,0)
710  PRINT :"SORRY, THE CORRECT
     ANSWER▲▲▲WAS";ANSWER::
720  RETURN
730  FOR I=1 TO 5
740  CALL SOUND(50,110*I,I,220*I,I,440*I,I)
750  NEXT I
760  PRINT :"THAT'S CORRECT!"::
770  RIGHT=RIGHT+1
780  RETURN
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100–190	clears screen and sets up display with audio-visual effects
200	reseeds the random number generator
210–280	prints the menu of options
290	sets the number of correct answers to 0
300–310	inputs a choice and rejects invalid responses
320	clears screen
330–360	main loop of the ten-question quiz
340	prints the current question number
350	depending on which subject was chosen, control is sent to the appropriate subroutine

370 prints out percent score of correct answers

380–400 asks the user if another round is desired. If so, control is sent to line 180. If the response is neither Y nor N, control is returned to the INPUT statement

410–420 screen is cleared and the program ends

430–480 addition subroutine; generates two random numbers A and B each between 1 and 100 and places A + B in ANSWER

490–540 subtraction subroutine; generates two random numbers A and B each between 1 and 100 and places A − B in ANSWER

550–600 multiplication subroutine; generates two random numbers between 1 and 25 and places their product in ANSWER

610–660 division subroutine; generates two random numbers between 1 and 25, and manipulates them to provide for integer quotients

670–780 main subroutine that requests the user's response, which is checked against the computer's calculated result

700–720 negative response—the answer given was wrong; prints appropriate message and emits losing sound

730–780 positive response—the answer given was right; prints appropriate message, emits winning sound and adds 1 to RIGHT

PROGRAM DESCRIPTION

After the screen is cleared and a welcoming display in sound and sight is created, the screen is first set to light green and is then cleared. A menu is then displayed, indicating the four choices of arithmetic operations that are available. According to the menu, 1 represents addition, 2 subtraction, 3 multiplication, and 4 division. The player is then prompted to select a number from 1 to 4, which is then validated; that is to say, it is tested to ensure that it is

neither less than 1, greater than 4, nor a noninteger. If any of these conditions are violated, control is sent back to line 300, where the user is asked to make another response. If all is well, the screen is cleared and the ten-question quiz begins.

Line 340 prints the literal QUESTION # followed by the value of Q, which ranges from 1 to 10. Instead of simply printing out the value of Q, we have adopted the strategy of using its string equivalent, which is achieved by using the STR$ function. The reason for resorting to this technique is that normally when numbers are printed, they are surrounded on each side by a space. Changing the representation of the number to its string equivalent eliminates the extra space and thus improves the appearance of the output. Maybe you would like to check this out for yourself on the computer. If so, just change line 340 to read

```
340  PRINT :"QUESTION #";Q
```

and you will find that the number in Q is printed alongside the literal with an unwanted space separating it from the number sign.

Depending on the selected arithmetic operation, the value of CHOICE will be 1, 2, 3, or 4. Control must be sent to one of the four different subroutines depending on this value. There is a particularly convenient instruction to direct such a flow of control. The command that accomplishes this task is called ON GOSUB. Line 350 reads as follows:

```
350  ON CHOICE GOSUB 430, 490, 550, 610
```

If the value of CHOICE is equal to 1 a branch is made to the subroutine located at line 430. If it is equal to 2, control is sent to the subroutine at 490, and so on. Assuming that the value of CHOICE is equal to 1, control is sent to the subroutine beginning at 430, the addition subroutine. Two random numbers, A and B, are generated and the variable ANSWER is set to their sum. The numbers A and B are first printed out and then control is sent to 670, where the

user types in an answer. If the user's response agrees with the computer's calculated result, control is sent to line 730 where a complimentary sound is emitted and a cheerful message is displayed. At the same time the value of RIGHT is incremented by 1 and the subroutine returns control to the main loop for the next question.

The other options behave in a similar manner, except for division. Since division can result in fractional answers even where only integers are involved, it is important to make sure that the numbers divide evenly. This is done by "reverse engineering" the problem. In other words, we take A and B (which are whole numbers) and multiply them, getting a product that must, of course, be divisible by both A and B. The problem may then be presented as that product divided by either one of the two numbers. This operation will always yield the other number—thus ensuring an integer result.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. It is easy to change the difficulty of the problems by restricting the random numbers to smaller or larger values in each of the four subroutines. For younger children, single-digit numbers are more advisable, while for those at the stage of learning the multiplication tables, setting the maximum value for a number at 12 is probably a good idea.
2. Subtraction can produce negative results. If this is not desired, it is recommended that a test be added to the subtraction routine to ensure that the value of A is always greater than B.
3. Should you wish to change the number of questions in the quiz, this may be done quite easily by changing the value of the upper limit of Q in line 330. Should you do this, however, you will have to give some attention to line 370, where the percentage of correct scores is calculated and printed out.
4. You might wish to add other operations, such as exponentiation, to add variety to the quiz.

PROGRAM

4

"SCRAMBLER"

PURPOSE

Scrambler is a game for testing your pattern-perception and vocabulary skills. The game consists of a timed quiz in which you are asked to recognize a series of random words, presented one at a time, which have been randomly jumbled. It is not enough to figure out each original word, however, You must also type it, correctly spelled, within the time interval allotted. At the higher levels, this may not be so easy—particularly if you increase the word list stored in the DATA statements.

```
100   REM SCRAMBLER
110   CALL CLEAR
120   PRINT "WELCOME TO ...":::::::::
130   FOR J=1 TO 3
140   FOR I=1 TO 10
150   CALL KEY(3,KEY,STATUS)
160   IF STATUS THEN 230
170   CALL SOUND(50,110*I,0)
180   CALL SCREEN(INT(RND*16)+1)
190   CALL HCHAR(23,13,ASC(SEG$
      ("SCRAMBLER▲",I,1)),5)
```

```
200   PRINT TAB(13);SEG$("SCRAMBLER▲",I,1)
210   NEXT I
220   NEXT J
230   PRINT ::::::::::
240   CALL SCREEN(13)
250   DIM WORD$(100)
260   N=10
270   FOR I=1 TO N
280   READ WORD$(I)
290   NEXT I
300   DIM SCRAMBLER(15),VICTORY(5)
310   FOR I=1 TO 5
320   READ VICTORY(I)
330   NEXT I
340   INPUT "ENTER THE STARTING
      LEVEL▲▲▲▲(1-9) ":LEVEL
350   IF (LEVEL<1)+(LEVEL>9)+
      (LEVEL<>INT(LEVEL))THEN 340
360   FOR L=LEVEL TO 9
370   LF=(20-2*LEVEL)*25
380   CALL CLEAR
390   PRINT "LEVEL:";L::
400   RIGHT=0
410   FOR Q=1 TO 10
420   RANDOMIZE
430   SCRAMBLE$=""
440   ANSWER$=""
450   R=INT(RND*N)+1
460   TEMP$=WORD$(R)
470   LNG=LEN(TEMP$)
480   FOR I=1 TO LNG
490   CALL SOUND(100,110*1.3^I,0)
500   SCRAMBLER(I)=ASC(SEG$(TEMP$,I,1))
510   NEXT I
520   FOR I=1 TO LNG
530   CALL SOUND(100,(LNG-I+1)*110,0)
540   R=INT(RND*LNG)+1
550   TEMP=SCRAMBLER(I)
```

```
560  SCRAMBLER(I)=SCRAMBLER(R)
570  SCRAMBLER(R)=TEMP
580  NEXT I
590  FOR I=1 TO LNG
600  SCRAMBLE$=SCRAMBLE$&CHR$(SCRAMBLER(I))
610  NEXT I
620  TL=LF*2*LOG(LNG)
630  PRINT SCRAMBLE$:
640  FOR T=1 TO TL
650  CALL KEY(3,A,X)
660  IF X=1 THEN 700
670  NEXT T
680  PRINT :"SORRY, YOU EXCEEDED
     THE▲▲▲▲▲TIME LIMIT. THE ANSWER
     WAS: ";TEMP$::
690  GOTO 760
700  T$=CHR$(A)
710  PRINT T$;
720  ANSWER$=ANSWER$&T$
730  IF LEN(ANSWER$)<LNG THEN 670
740  IF ANSWER$=TEMP$ THEN 830
750  PRINT ::"SORRY, THAT IS
     INCORRECT.▲▲▲THE RIGHT ANSWER
     IS:":TEMP$::
760  NEXT Q
770  INPUT "WOULD YOU LIKE TO TRY
     AGAIN?":QUERY$
780  IF SEG$(QUERY$,1,1)="N" THEN 820
790  IF RIGHT<7 THEN 940 ELSE 960
800  NEXT L
810  PRINT "YOU ARE A GRANDMASTER
     OF▲▲▲▲SCRAMBLING. CONGRATULATIONS."
820  END
830  FOR I=1 TO 5
840  CALL SOUND(100,VICTORY(I),0)
850  NEXT I
860  PRINT ::"THAT'S RIGHT! GOOD SHOW!"::
870  RIGHT=RIGHT+1
```

```
880  FOR T=1 TO 300
890  NEXT T
900  GOTO 760
910  DATA "COMPUTER","PHLEGM","NAIVE",
     "EXISTENTIAL","PARTIAL",
     "REITERATE","INTEGRAL","COEXIST",
     "FOREIGN"
920  DATA "DEMEANOR"
930  DATA 444,604,670,704,772
940  PRINT "SORRY, YOU SEEM TO HAVE HAD A
     ROUGH TIME ON THIS LEVEL."::
950  GOTO 410
960  FOR I=1 TO 5
970  CALL SOUND(50,VICTORY(I),0)
980  NEXT I
990  PRINT ::
1000 ON INT(RND*3)+1 GOSUB 1050,1070,1090
1010 PRINT "YOU SCORED";RIGHT*10;"%"
1020 FOR T=1 TO 400
1030 NEXT T
1040 GOTO 800
1050 PRINT "THAT'S GREAT!"
1060 RETURN
1070 PRINT "GOOD JOB. PROCEED TO
     THE▲▲▲▲NEXT LEVEL"
1080 RETURN
1090 PRINT "MAGNIFICENT WORK!"
1100 RETURN
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–240	clears screen and prints greeting
250–260	reserves room for 100 words, of which 10 are used
270–290	reads the 10 words from DATA statements

300	reserves room for the scrambled word and the victory notes
310–330	reads in the five-note victory theme
340–350	requests the starting level and checks for its validity
360–800	the main loop
370	sets the time limit for each level, which decreases as the level increases
380–390	clears the screen and prints the current level
400	sets the number of correct answers to 0
410–760	the loop containing the ten-question quiz
420	restarts the random sequence each time around the loop for greater unpredictability
430–440	sets the user's answer and the computer's scrambled output to the null string
450–470	picks a random word from the list, stores it in TEMP$, and calculates the number of letters in the word
480–510	copies the computer's chosen word to an array
520–580	scrambles the array randomly
590–610	copies the scrambled array back to the computer's output string
620	calculates the time limit
630	prints out the scrambled word
640–670	scans keyboard for the duration of the time limit
680–690	prints a message if the time limit is exceeded, provides the answer, and goes on to the next question
700–740	prints the letter typed each time and returns for more input until the required number of characters have been typed
750	if answer is incorrect, prints appropriate message and provides the correct one
770–780	asks whether user wants another round and if not, terminates program
790	if another round is requested and score is less than passing, repeats the current level; otherwise makes victory sound and advances a level

810–820	congratulates the player on winning the game and terminates the program
830–850	routine to sound the victory theme
860–900	routine to advise user of correct answer and increment the count of correct answers; provides a short delay and returns to the main routine
910–920	contains the data
930	data for the victory sound
940–950	apologetic message to inform the player of failure and returns to the main routine
960–980	another victory roll
990–1040	prints out the final score, provides a short delay, and advances to the next level
1050–1100	the three congratulatory messages, one of which is selected randomly

PROGRAM DESCRIPTION

After the screen is cleared, the usual welcoming message is displayed. It may be interrupted at any time, however, by pressing any key, which triggers the IF . . . THEN statement in line 160, sending control out of the loop to line 230. Within the loop, different notes are generated by the statement in line 170 while the color of the screen background is changed randomly and SCRAMBLER is printed vertically down the screen. The way this is accomplished is by means of the HCHAR statement, which permits a character to be placed anywhere on the screen. For example, the statement

```
CALL HCHAR (23,13,65)
```

places the character A (ASCII 65) at row 23, column 13. This instruction is also capable of accepting a repetition factor. For example, the instruction

```
CALL HCHAR (23,13,65,5)
```

places five successive A's, starting at row 23, column 13, moving to the right. In the program, we examine slices of the string "SCRAMBLER " (notice the space following the letter *R*), using the SEG$ function. In order to obtain the ASCII value of the character so as to use it in a CALL HCHAR statement, we take advantage of yet another function, known as ASC, which converts any character into its ASCII code. Therefore, the instruction

```
190 CALL HCHAR
    (23,13,ASC(SEG$("SCRAMBLER▲",I,1)),5)
```

has the effect of producing a series of horizontal lines composed of five identical characters of the string "SCRAMBLER ", starting with the first and ending up with the space. The space therefore separates the words as they scroll up the screen. It is interesting to remove line 200 and see what happens. Without the PRINT statement, the screen does not scroll and so the letters come out one on top of another. You will notice that the PRINT instruction contains the clause TAB(13). This behaves like the tab function on a typewriter. It begins printing at the thirteenth column from the left of the screen.

Once the display has been scrolled out of the way to the top of the screen (in line 230) the screen is set to light green (color 13) and the main section of the game begins. Within the FOR . . . NEXT loop in lines 270–290, the ten words contained by the DATA statements beginning in line 910 are read and stored in the array WORD$. Next, the victory tune is read in at lines 310–330. The data for this is located at line 930. The player is then prompted to enter the starting level of his or her choice. The permitted range is 1 to 9, with 1 being the easiest and 9 the most difficult. To avoid possible problems, the value inputted in LEVEL is validated to ensure that it is not less than 1, not greater than 9, and that it is an integer. Should any of these conditions be violated, control is sent back to the input statement in line 340, permitting the user to enter another value.

The level factor, stored in the variable LF, is part of the

mechanism by which the time limit is computed. The level factor starts out as 18 times 25 and goes down to 2 times 25. The time limit is calculated in line 620 by multiplying LF by twice the natural log of the length of the word. This simply means that the longer the scrambled word, the more time that is given for deciphering it.

In order to randomly select a word (lines 450–470), a random number between 1 and 10 is generated and stored in R. TEMP$ is then set to WORD$(R) and is substituted for it afterwards. The length of the selected word is then determined by means of the LEN function, which behaves in the following manner:

```
PRINT LEN("COMPUTER")
```

returns the value "8," since there are eight characters in the string COMPUTER.

Armed with the variable LNG, which now contains the length of the word, a loop is entered in which successive one-letter slices of the word are copied into the numeric array SCRAMBLER using the ASCII equivalent of each character. Just to add a little more variety to this, a different sound is emitted each time around the loop. Once the array contains the ASCII representation of the word, we can set about scrambling it with ease. We simply proceed through the length of the word, switching each element with a randomly selected one. In this way, the whole word is scrambled—at least in its numeric form. Now the numeric representations have to be converted back to their character representations. This is done in the FOR . . . NEXT loop extending from 590 through 610, where two new features present themselves. First is the ampersand sign (&). This is the string equivalent of addition, which simply combines (concatenates) the two strings end to end. For example:

```
PRINT "HOT"&"DOG"
```

displays HOTDOG as one character string. Similarly, a

string variable may be set equal to the "sum" of two strings as in the following example:

```
TOGETHER$="PART1 "&"PART2"
```

which assigns the string "PART1 PART2" to the variable TOGETHER$. The second new feature is the CHR$ function, which is simply the reverse of the ASCII function—that is to say, given the ASCII code of a character, it returns the character form. For example

```
PRINT CHR$(65)
```

displays the letter A on the screen.

The effect then of the FOR . . . NEXT loop that goes from 1 to LNG (the length of the selected word) is to set the variable SCRAMBLE$ to the scrambled version of TEMP$. It is for this reason that SCRAMBLE$ was first set to the null string—the character equivalent of 0. If this were not done, SCRAMBLE$ would keep getting larger as more and more characters are added to it. It would therefore only be correct the first time the program is run. On each subsequent occasion, the string would get larger and larger and would never be equal to the scrambled version of the newly selected word.

After the scrambled word is printed out in line 630, the timed loop begins. If a key has not been pressed within the calculated time limit, a message to that effect is printed out, the correct answer is displayed, and the program proceeds to the next question. If, on the other hand, a response is registered, control is sent to line 700, where the character version of the pressed key is stored in T$. After it is printed, it is concatenated to ANSWER$ (which is set to the null string in line 440 for the same reason as above). When the length of ANSWER$ reaches that of the jumbled word, a test is made for equality. If they match, control is sent to line 830 where a victory sound is emitted, an approving message is displayed, and the count of correct responses is incremented by 1. After a small time delay, the computer goes on to pose the next question.

In line 1000 we have an example of the ON condition used with subroutines. Its purpose is to congratulate the user using different messages. We have selected three such messages and each time a score is given, one of these messages is randomly selected to be displayed, thereby giving the game a more human quality. The statement works in the following way:

```
ON VAR GOSUB 100, 200, 300
```

If the value stored in the variable VAR is equal to 1, control is sent to the first subroutine mentioned (the one beginning in line 100). If VAR is equal to 2, control is sent to the second subroutine (line 200) while if it is equal to 3 a branch is made to the third subroutine, the one beginning in line 300.

This technique is used in line 1000, where a random integer between 1 and 3 is generated. If it is equal to 1, control is sent to the subroutine in line 1050; if it's 2, control goes to line 1070, and if it's equal to 3, control branches to line 1090.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The words themselves can be changed any time it is felt necessary. This might be a good idea if the players have become familiar with the words to be scrambled. Also, you might want to make the words easier or more difficult, depending on the players.
2. The number of words can be altered by changing the constant for N in line 260, where it is currently set to 10. The DIMension statement allows for up to 100 words. However, should the number of words be increased, be sure to add the words to the DATA statements in lines 910–920. Also, the calculation of the percentage will have to be amended in line 1010.
3. The game can be made more competitive by changing the program so that one player types in the word to be scrambled. The computer then scrambles that word and the other player is left to unscramble it. This, of course,

would mean that the READ statement in line 280 would have to be changed to an INPUT statement.

4. The time limit can be increased or decreased by suitably amending line 620.
5. Another score can be kept to record the time taken to unscramble a word. In this way, a player who can unscramble words faster than the opponent gets credit for it.

PROGRAM 5

"HANGMAN"

PURPOSE

Like "Scramble," "Hangman" also has the computer select a word at random from a given list. However, here the computer displays a dash (or rather an underscore character) for each of the letters of the word selected. The human player has to guess the letters which compose the word. When a correct letter is guessed, the computer places it in the correct place in the word, eliminating the dash. However, each time an incorrect letter is typed, another letter from the word "Hangman" is displayed near the top of the screen. Once the word "Hangman" is spelled out in full the game is over and the player has lost.

```
100  REM HANGMAN
110  CALL CLEAR
120  RANDOMIZE
130  OPTION BASE 1
140  DIM WORDS$(100),TEMP(20),FLAG(26)
150  N=10
160  FOR I=1 TO N
170  READ WORDS$(I)
180  NEXT I
```

```
190 FOR I=1 TO 20
200 TEMP(I)=95
210 NEXT I
220 PRINT "THIS IS HANGMAN! HERE IS▲▲▲▲YOUR
    WORD: ";
230 TEMP$=WORDS$(INT(RND*N)+1)
240 L = LEN(TEMP$)
250 FOR I=1 TO L
260 PRINT CHR$(TEMP(I));
270 NEXT I
280 PRINT::"[";SEG$("HANGMAN",1,
    DEATH);"]"::"LETTERS USED:";USED$::
290 INPUT "ENTER YOUR LETTER: ":LETTER$
300 IF (LETTER$<"A")+(LETTER$>"Z")+
    (LEN(LETTER$)<>1)THEN 290
310 IF FLAG(ASC(LETTER$)-64)THEN 290
320 FLAG(ASC(LETTER$)-64=-1
330 USED$=USED$&LETTER$
340 F=0
350 FOR I = 1 TO L
360 IF LETTER$=SEG$(TEMP$,I,1)THEN 390
370 NEXT I
380 IF F=0 THEN 450 ELSE 250
390 F=-1
400 REM
410 TEMP(I)=ASC(LETTER$)
420 RIGHT=RIGHT+1
430 IF RIGHT=L THEN 520
440 GOTO 370
450 DEATH=DEATH+1
460 IF DEATH<7 THEN 280
470 PRINT ::"SORRY CHUMP! YOU LOSE"
480 PRINT ::"THE WORD WAS: ";TEMP$
490 END
500 DATA "KEYBOARD","COMPUTER","GAME",
    "VERIFICATION","EDUCATE",
    "INDUSTRIALIZE","RECREATIONAL",
    "INTERVIEW"
```

```
510  DATA "ZONES","PRODUCT","QUOTIENT"
520  PRINT :"YOU WIN! CONGRATULATIONS!"
530  PRINT :"THE WORD IS:";TEMP$
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–120	clears screen and reseeds the random number generator
130	suppresses the 0th elements of arrays
140	reserves space for the arrays
150	sets the number of words in data to 10
160–180	reads the words from DATA statements
190–210	sets the array TEMP to the symbol _
220	prints a prompt
230–240	picks a random word and determines its length
250–270	prints the character equivalents of the array TEMP
280	prints out the incorrect answer status and the letters used so far
290–300	prompts the user to enter letter, which is then validated
310	checks if letter has already been used; if so it is ignored and the prompt reappears
320	marks the letter as having been used
330	adds the letter to USED$ which is printed before each turn
340	sets the flag F to false, representing the fact that the letter has not yet been found
350–370	searches for all occurrences of the letter in the word; if present, the flag F is set to true and the position of the letter is changed from an underbar to the letter itself
380	if the player's letter is not present in the selected word, increments the death count and checks to see if the player is dead

390–440 routine to set the flag for "letter found" and to change the position from an underbar to the letter itself; returns to the main routine

450–490 routine to increment death count, which checks to see if the player is still alive (before the seven chances expire); if alive, continue the game; if not, user is informed of what the word was, and the game ends

500–510 DATA statements containing the list of selected words

520–530 congratulatory message informing the player of victory

PROGRAM DESCRIPTION

After the screen has been cleared, the random number generator is reseeded. To conserve a little memory space, OPTION BASE 1 is selected, which means that no space is reserved for the 0th elements of the DIMensioned arrays.

Since the program includes ten words that are to be guessed by the player, N is set equal to 10 in line 150. In the FOR . . . NEXT loop in lines 160–180, the ten words are read from the DATA statements and are stored in the array WORD$. The array TEMP is DIMensioned at 20, and is filled (by the FOR . . . NEXT loop in lines 190–210) with the ASCII value 95, which represents the underscore character (_). This restricts the maximum length of each of the words to twenty characters, which is probably more than enough for most situations.

A random word is selected and is stored in TEMP$ in line 230. The number of characters in this word (its LENgth) is stored in the variable L. In the FOR . . . NEXT loop in lines 250–270, a number of underscores equal to the number of letters in the randomly selected word are displayed. Line 280 displays the square brackets between which the letters of the word "hangman" will successively be added each time a wrong letter is guessed. Initially, no letters will be displayed. Line 280 also displays

the letters guessed by the player. The variable DEATH, which until now has not been defined, is automatically taken to be 0 by the system. Later on we will use DEATH as a counter. This applies equally to USED$, which is initially set to the null string.

In line 290, the player is invited to guess a letter. Whatever key is typed, that value is stored in LETTER$. However, since there is nothing to stop a user from entering a character other than a letter of the alphabet, or even more than one letter, a test is made in line 300 to insure that this has not happened. If an error is indeed made, control is simply returned to line 290.

FLAG is a twenty-six-element array, one element for each letter of the alphabet. Initially each element has the value "0" (by default), indicating that the corresponding letter has not yet been chosen by the player. (By "corresponding," we mean that the first element represents the letter *A*, the second represents *B*, etc.) Any value other than 0 would indicate that the corresponding letter had already been selected by the player. The inner part of the IF test in line 310 is ASC(LETTER$)−64. This uses the chosen letter's ASCII equivalent value, which is reduced such that if the letter is *A* (ASCII code 65), we get the value "1" after subtracting 64 from it. Similarly, if the letter is *B*, we get 2, and so forth. In other words, we reduce whatever letter is typed in to its positional value in the alphabet. This result is used to select the element of FLAG corresponding to the chosen letter.

The IF test in line 310 takes a form we have not as yet encountered:

```
IF VAR THEN 100
```

Simply stated, if the variable VAR has the value 0 (the computer's way of storing the value "false"), then the test fails and the THEN clause is ignored. Anything other than 0 is regarded as "true," and control would here be passed to line 100. Line 310 in "Hangman," in effect, tests the value of the selected element of the array FLAG. If it is

equal to 0, the test fails, which means the letter has not previously been chosen and everything is in order. If, however, the value is not 0, the letter has already been selected by the player, and so control goes back to line 290, where the opportunity is again given to type in a letter. If this is the first time the letter has been chosen, control automatically falls down to line 320 where the value "−1" is stored in the element of FLAG tested in line 310. As mentioned above, this will indicate in the future that the letter has already been chosen. In line 330 that letter is now concatenated to the string variable USED$.

In line 340, the flag F is set to zero, representing the fact that the letter chosen by the player is not yet matched. The loop in lines 350–370 scans the word for all occurrences of the chosen letter, setting the flag F to −1 if any are found. At the end of the loop, a test is made to ascertain whether F is still 0. If it is, no occurrences of the letter were found and the player gains one more letter, thus getting one step closer to death. If any letters were found, control passes directly to the loop beginning in line 250, where the known elements of the word are printed out, underbars substituting for letters as yet unidentified. The program then proceeds to ask for another letter. Each time an occurrence of the selection is found, 1 is added to the variable RIGHT in line 420. A test is then made to determine if RIGHT is equal to the length of the word (L). In other words, the test checks if all the letters have been matched. If so, the game is over and a congratulatory message is printed out. If not, control passes from the routine back to the loop and the scan for all occurrences of the chosen letter goes on.

If the player chooses a letter that is not found in the word, control is transferred to the routine in line 450, which increments the DEATH count. Once this reaches 7, the player is informed that the game is lost and the program ends after the hidden word is revealed.

Finally, lines 500–510 are the DATA statements, containing the hidden words, which can always be changed at will.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The value of N (in line 150) may be changed to any number up to a maximum of 100 as long as that number of words is included in the DATA statements in lines 500–510.
2. A fresh list of words can be used as soon as the current words prove to be too easy due to familiarity.
3. Rather than terminating after one round, the program can easily be amended to select another word. Should this be desired, the values of RIGHT, DEATH, FLAG, and USED$ must be reset to their original default values (0 for the numeric variables, and the null string for USED$).
4. The game can be amended so that two players can compete against each other. By changing the READ statement to an INPUT statement and eliminating the DATA statements, it will be possible for one player to enter each word (out of view of the opponent, of course) and for the other player to guess the word(s).
5. Although it is traditional to use the seven-letter word "hangman," there is really no reason not to select any other word to increase variety and difficulty. For example, if a shorter word, like "bingo," were used, the game would become more difficult, because the number of wrong guesses permitted would be reduced. On the other hand, selecting a longer word, such as "executioner," in place of "hangman" makes the game easier because it would take 11 errors to lose. Take note that in line 460, the number "7" must be replaced with the number of letters in the chosen replacement for "hangman."

PROGRAM

6

"SOUND/SIGHT SIMON"

PURPOSE

This game is based on an electronic game that became available several years ago. In the game, the player must repeat a random tone pattern of ever-increasing length and speed. Naturally, as the number of tones in the pattern increases, and the pace quickens, the game soon becomes rather difficult, to say the least.

The program which follows is an attempt to simulate the operation of the game of "Simon." In fact, a case could be made for the claim that this TI-99/4A version is an improvement, because it provides the user with not four but no fewer than nine increasing levels of difficulty. In addition, this computerized version of "Simon" provides nine tones rather than just four. As each tone is emitted, a digit corresponding to that tone is displayed in the center of the screen so that even if you are hard of hearing or have poor eyesight, you still have a fighting chance to succeed.

The game works like this: The computer emits a random tone, at the same time displaying a corresponding digit (1–9). The player must "echo" back this tone by striking the correct numeric key on the keyboard. For example, if tone number 7 is emitted (a "7" is also displayed in the

center of the screen), number 7 on the keyboard must be pressed. The computer then repeats the first tone followed immediately by another random tone. The player must now echo the two-tone sequence in correct order. This process continues, with the computer adding a new random tone in each successive round. The length of the sequence is limited by the level of difficulty selected by the player, and the player wins if he or she is able to repeat the final sequence without mishap.

```
100 REM SOUND/SIGHT SIMON
110 CALL CLEAR
120 RANDOMIZE
130 PRINT "PREPARE YOURSELF FOR
    ...":::::::::::::
140 PRINT "▲SSSS▲IIII▲M▲▲▲M▲▲00▲▲N▲▲▲N"
150 PRINT "S▲▲▲▲▲▲II▲▲MM▲MM▲0▲▲0▲NN▲▲N"
160 PRINT "▲SSS▲▲▲II▲▲M▲M▲M▲0▲▲0▲N▲N▲N"
170 PRINT "▲▲▲▲S▲▲II▲▲M▲▲▲M▲0▲▲0▲N▲▲NN"
180 PRINT "SSSS▲▲IIII▲M▲▲▲M▲▲00▲▲N▲▲▲N"
    :::::::::::::::::
190 DIM SEQ(23),NOTE(9)
200 FOR I=1 TO 9
210 READ NOTE(I)
220 NEXT I
230 DATA 446,502,560,598,664,744,832,
    890,999
240 PRINT "TYPE THE DIFFICULTY LEVEL"
250 INPUT "(1 IS LOW,9 IS HIGH) ":LEVEL
260 IF (LEVEL < 1) + (LEVEL > 9) THEN 250
270 FOR L=LEVEL TO 9
280 CALL CLEAR
290 PRINT "YOU ARE NOW ON LEVEL";L
300 TIME=100*(10-L)
310 MAX_NOTES=L*2+5
320 FOR I=1 TO MAX_NOTES
330 SEQ(I)=INT(RND*9)+1
340 FOR J=1 TO I
```

```
350  CALL SOUND(TIME,NOTE(SEQ(J)),0)
360  CALL HCHAR(12,15,48+SEQ(J))
370  FOR T=1 TO TIME/3.7
380  NEXT T
390  NEXT J
400  CALL HCHAR(12,15,32)
410  FOR J=1 TO I
420  CALL KEY(3,A,X)
430  IF X=0 THEN 420
440  A=A-48
450  CALL SOUND(TIME,NOTE(A),0)
460  CALL HCHAR(12,15,A+48)
470  IF A<>SEQ(J) THEN 580
480  FOR T=1 TO TIME/3.7
490  NEXT T
500  NEXT J
510  CALL HCHAR(12,15,32)
520  FOR T=1 TO TIME/2
530  NEXT T
540  NEXT I
550  NEXT L
560  PRINT "YOU WIN"
570  END
580  CALL SOUND(1000,110,0)
590  PRINT "SORRY, YOU LOSE."
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark statement
110	clears screen
120	reseeds random number generator
130–180	presents title of game on the screen
190	reserves storage for the two arrays
200–220	reads in the nine notes from data
230	data for musical notes
240–260	prompts user to input level of difficulty and validates response

270–550	main loop to repeatedly increase the level of difficulty
280–290	clears screen and prints the current level number
300	sets the amount of time that each note will play and each number will appear on the screen
310	sets the number of tones that must be sounded out before the level has been successfully completed
320–540	loop in which the string of notes keeps getting longer and longer up to the limit set in line 310
330	sets the latest note in the series to a random tone
340–390	loop that plays the entire sequence of notes up to the latest
350	plays the note
360	displays the corresponding number on the screen
370–380	creates a small time delay between each note
400	erases the last number from the screen
410–500	requests the same sequence to be duplicated by the player
420–430	scans keyboard until a character is typed
440	subtracts 48 from the ASCII value of the character typed in
450	plays the note whose number was typed
460	displays the number typed in at the center of the screen
470	if the player makes a mistake, emits losing sound and ends the game
480–490	short time delay
510	erases the last number typed from the display
520–530	another small time delay
580–590	routine to emit losing sound, print an appropriate message, and end the game

PROGRAM DESCRIPTION

After the screen is cleared and the introduction is displayed, the nine notes are read in, in numeric form, from the DATA statement in line 230 into the array NOTE. The player is then asked to type in the required initial level of

difficulty, a number from 1 to 9, where 1 is the easiest level and 9 is the most difficult. The response is stored in the variable LEVEL. In line 260, this value is tested to be sure that it is not less than 1 or greater than 9. If it is, control is sent directly back to line 250, where the player is invited to enter another value.

The FOR . . . NEXT loop extending from line 270 through 550 generates successive sequences of notes. The number of sequences generated depends on the level of difficulty selected. An initial sequence will be produced and, if the player successfully repeats the sequence, the level of difficulty is increased one notch and a new sequence is generated. Since 9 is the highest possible level, if the player has chosen level 9 only one sequence will be produced. It may be safely assumed, however, that unless the player is a real pro, he or she will probably not even make it through this level-9 sequence!

The computer always notifies you of the level at which you are playing by displaying this information on the screen. A variable TIME is computed, based on the value of LEVEL, which is used to control the speed at which the sequence is played and displayed (the higher the level of difficulty, the lower the value of TIME and therefore the faster the speed.) MAX_NOTES represents the maximum number of notes in the current sequence, and is calculated based on the value of L, the current level of difficulty.

The FOR . . . NEXT loop beginning with line 320 and extending to line 540 generates one complete sequence of notes. Since the index, I, assumes the successive values 1,2, etc., up to MAX_NOTES, the variable I indicates how many notes of the sequence are being played in the current round. The effect of line 330 is to randomly choose one new note and add it to the sequence. This is done by randomly selecting a number from 1 to 9 and storing that number in the next available location of the array SEQ. Thus, the contents of the array SEQ reflect the notes in the sequence that have been played so far.

The next FOR . . . NEXT loop (340–390) plays the se-

quence of notes generated so far, and simultaneously displays each corresponding digit in the center of the screen. It utilizes sophisticated versions of the CALL SOUND and CALL HCHAR commands. In line 350, the computed value of TIME controls the duration of the emitted tone. The second parameter of this command requires some discussion. SEQ(J) contains the digit corresponding to the tone currently being emitted. However, the computer cannot use this number to generate the proper tone. Instead, this number is used as the index to the array NOTE, whose elements contain numeric values which the computer interprets as the frequencies of the required tones. The third parameter of the CALL HCHAR command, in line 360, could also use some explanation. We have already seen that the third parameter of CALL HCHAR must be an ASCII value. Since the ASCII value of the character "1", for example, is 49, the value 48 is added to the value of SEQ(J) to produce the required ASCII value.

The "empty" FOR . . . NEXT loop in lines 370–380 is used to produce a short delay between notes. Note that the length of the delay is based on the value of TIME, but the actual value of TIME is not used. Instead, it is "scaled down" by dividing it by 3.7. It has been determined that this will produce a suitable delay time. Line 400 has the effect of blanking the last character of the sequence displayed on the screen (the ASCII value 32 is the space character).

We have arrived at that point in the program where we are ready to discuss the player's response. This is encompassed in the FOR . . . NEXT loop extending from line 410 to line 500. Each time through this loop, the player is expected to enter the next required note. Lines 420–430 in effect wait for the player to strike a key. The moment this is done, 48 is subtracted from the ASCII value of the character entered in order to convert it to its corresponding digit (1–9). This value is required in the CALL SOUND and CALL HCHAR commands that reside in lines 450 and 460 and serve to play and display the selected note. Line 470

tests to see whether the value entered by the player is in fact correct. If it is correct, the IF test fails and control falls down to the next "empty" FOR . . . NEXT loop, providing a short delay between notes. If they do not agree, however, control branches to line 580, which sounds a raspberry and displays an appropriate message, ending the game. Line 510 (which is identical to line 400) blanks out from the screen the last digit entered by the player. Then the empty FOR . . . NEXT loop in lines 520–530 provides a short delay before the computer plays and displays the next sequence.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The number of levels of difficulty may be changed, depending upon the ability of the players. If the number of levels is modified, appropriate changes must be made throughout the program.
2. A limit could be implemented on the time allowed the player to enter a note. This could be accomplished by putting lines 420 and 430 within a FOR . . . NEXT loop similar to that at lines 370–380 (line 430 would also have to be modified to allow for exiting the loop in the event that a key is pressed before the time limit has expired).

PROGRAM

7

"BLACKJACK"

PURPOSE

"Blackjack," sometimes known by its other name, "Twenty-one," is not only a popular game in households across the nation, but is also one of the most popular games found in casinos around the world. The reason for this popularity is probably the fact that its rules are rather simple. Although there are many versions of the game in existence, the version we have designed for the TI-99/4A contains all the essentials, but includes certain restrictions in order to keep the program within manageable proportions. This game permits up to three players plus a dealer (the role played by the computer).

The object of the game of "Blackjack" is to acquire a hand that totals 21 or comes as close as possible to 21 without exceeding this limit. (Going over 21 is generally referred to as "busting.") The hand with the highest total not exceeding 21 is considered the winner. Each player is dealt two cards initially. At his or her turn, each player has two options: to receive a card (to be "hit") or to stay with the present hand (to "stand"). The player may ask to be hit as often as desired (or until he busts), but once he stands, his turn is over and the next player takes over. The

dealer goes last, and is special in several senses. First, the second card dealt to the dealer remains face down (hidden) until the end of the game. Second, if the dealer busts, all other players who have not themselves busted become winners.

In "Blackjack," the suits (hearts, clubs, diamonds, spades) have no effect upon the game, and therefore are not displayed. Nevertheless, the game is played with fifty-two cards. All picture cards (jack, queen, king) are regarded as having a value of 10, while the ace can assume the value 1 or 11, at the option of the player.

```
100  REM  BLACKJACK
110  CALL CLEAR
120  PRINT "WELCOME TO":::::::"THE
     CYBERNETIC CASINO!":::::::
130  OPTION BASE 1
140  CALL CHAR(128,"FF818181818181FF")
150  DIM DECK(52), HAND(4,7), X(3), Y(3),
     VALUES$(13), SUM(4)
160  FOR I=1 TO 13
170  READ VALUES$(I)
180  NEXT I
190  DATA "A", "2", "3", "4", "5", "6",
     "7", "8", "9", "T", "J", "Q", "K"
200  CALL CLEAR
210  FOR I=1 TO 3
220  READ X(I),Y(I)
230  NEXT I
240  DATA 4,2,20,2,4,8
250  PRINT "HOW MANY PLAYERS?(1-3):";
260  CALL KEY(3,KEY,STATUS)
270  IF STATUS=0 THEN 260
280  IF (KEY<49)+(KEY>51)THEN 260
290  NUM_PLAYERS=KEY-47
300  PRINT NUM_PLAYERS-1
310  FOR I-1 TO NUM_PLAYERS-1
320  CALL HCHAR(Y(I),X(I)-1,ASC("["))
```

```
330   CALL HCHAR(Y(I),X(I),48+I)
340   CALL HCHAR(Y(I),X(I)+1,ASC("]"))
350   NEXT I
360   FOR I=1 TO 6
370   CALL HCHAR(8,18+I,ASC(SEG$
      ("DEALER",I,1)))
380   NEXT I
390   FOR I=1 TO 52
400   RANDOMIZE
410   DECK(I)=I-INT(I/13)*13+1
420   R=INT(RND*I)+1
430   TEMP=DECK(R)
440   DECK(R)=DECK(I)
450   DECK(I)=TEMP
460   NEXT I
470   FOR I=1 TO 2
480   FOR J=1 TO NUM_PLAYERS
490   CARD_COUNT=CARD_COUNT+1
500   HAND(J,I)=DECK(CARD_COUNT)
510   SUM(J)=SUM(J)-HAND(J,I)*(HAND(J,I)
      <11)-10*(HAND(J,I)>10)
520   IF J=NUM_PLAYERS THEN 550
530   T = ASC(VALUES$(HAND(J,I))
540   CALL HCHAR(Y(J)+2,X(J)+I+I-3,T)
550   NEXT J
560   NEXT I
570   CALL HCHAR(10,19,ASC(VALUES$(HAND
      (NUM_PLAYERS,1))))
580   CALL HCHAR(10,21,128)
590   FOR I = 1 TO NUM_PLAYERS - 1
600   C=2
610   FOR J = 1 TO 8
620   CALL HCHAR(14,J+3,ASC(SEG$("PLAYER▲
      #",J,1)))
630   NEXT J
640   CALL HCHAR(14,12,I+48)
650   CALL KEY(3,KEY,STATUS)
660   IF STATUS=0 THEN 650
```

```
670  KEY$=CHR$(KEY)
680  IF KEY$="H" THEN 980
690  IF KEY$<>"S" THEN 650
700  FOR J=1 TO 6
710  CALL HCHAR(14,13+J,ASC(SEG$
     ("STANDS",J,1)))
720  NEXT J
730  FOR T=1 TO 500
740  NEXT T
750  CALL HCHAR(14,1,32,20)
760  FOR J=1 TO 7
770  IF HAND(I,J)=0 THEN 810
780  IF (HAND(I,J)<>1)+(SUM(I)>11)THEN 800
790  SUM(I)=SUM(I)+10
800  NEXT J
810  NEXT I
820  C=2
830  CALL HCHAR(10,21,ASC(VALUES$(HAND
     (NUM_PLAYERS,C))))
840  IF SUM(NUM_PLAYERS)>16 THEN 1080
850  IF SUM(NUM_PLAYERS)>11 THEN 900
860  FOR I = 1 TO C
870  IF (HAND(NUM_PLAYERS,I)<>1)+
     (SUM(NUM_PLAYERS)+10<18)THEN 890
880  SUM(NUM_PLAYERS)=SUM(NUM_PLAYERS)+10
890  NEXT I
900  C=C+1
910  CARD_COUNT=CARD_COUNT+1
920  HAND(NUM_PLAYERS,C)=DECK(CARD_COUNT)
930  CALL HCHAR(10,17+C+C,ASC(VALUES$(HAND
     (NUM_PLAYERS,C))))
940  SUM(NUM_PLAYERS)=SUM(NUM_PLAYERS)-
     (HAND(NUM_PLAYERS,C)<11)*
     HAND(NUM_PLAYERS,C)-10*
     (HAND(NUM_PLAYERS,C)>10)
950  IF SUM(NUM_PLAYERS)>21 THEN 1160
960  GOTO 840
970  STOP
```

```
980  C=C+1
990  CARD_COUNT=CARD_COUNT+1
1000 HAND(I,C)=DECK(CARD_COUNT)
1010 CALL HCHAR(Y(I)+2,X(I)+C+C-
     3,ASC(VALUES$(HAND(I,C))))
1020 SUM(I)=SUM(I)-HAND(I,C)*(HAND(I,C)
     <11)-10*(HAND(I,C)>10)
1030 IF SUM(I)>21 THEN 1040 ELSE 650
1040 FOR J=1 TO 12
1050 CALL HCHAR(Y(I)+2,X(I)+J-3,
     ASC(SEG$("▲YOU BUST▲▲▲",J,1)))
1060 NEXT J
1070 GOTO 810
1080 FOR I=1 TO NUM_PLAYERS-1
1090 IF SUM(I)>21 THEN 1140
1100 IF SUM(I)<=SUM(NUM_PLAYERS)THEN 1180
1110 FOR J=1 TO 8
1120 CALL HCHAR(Y(I)+2,X(I)+J-2,
     ASC(SEG$("▲YOU WIN",J,1)))
1130 NEXT J
1140 NEXT I
1150 END
1160 SUM(NUM_PLAYERS)=0
1170 GOTO 1080
1180 FOR J=1 TO 8
1190 CALL HCHAR(Y(I)+2,X(I)+J-2,
     ASC(SEG$("YOU LOSE",J,1)))
1200 NEXT J
1210 GOTO 1140
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark statement
110–120	clears screen and displays welcoming message
130	all arrays begin with element 1
140	designs special character for face-down card

150	reserves space for the arrays, the matrix, and the character string array
160–190	reads in the character representation of the 13 kinds of cards, storing these values in the string array VALUES$
200	clears screen again
210–240	reads in the values of arrays X and Y, used to locate the fields on the screen associated with each player
250	a prompt is displayed asking how many players are in the game
260–270	scans keyboard until a key is pressed
280	tests to see if it is equal to 1, 2, or 3
290	converts ASCII value of pressed key to its numeric value (1, 2, or 3) plus 1 (for dealer)
300	prints the original number of players typed in
310–350	for each player, prints the player number in square brackets in its own location of the screen
360–380	prints DEALER in the dealer's location
390–460	generates the 52 cards of the deck and shuffles them, storing the shuffled deck in the array DECK
470–560	deals the first two cards to each player (including dealer)
490–500	deals a card from array DECK into matrix HAND
510	computes the current value of the hand, placing the result in array SUM
520–540	for all players but the dealer, puts the ASCII value of the card's symbol into T and uses it to print the symbol in the player's location
570–580	in the dealer's location, prints the dealer's first card followed by the special face-down card character
590–810	caters to each player, except dealer, individually
600	sets variable C to 2, the number of cards in each player's hand

610–640	displays the number of the player whose turn it is, in the middle of the screen
650–670	waits for a key to be pressed and stores the selected character in string KEY$
680–690	tests whether player wants to be hit (character H) or wishes to stand (character S)
700–720	displays STANDS
730–740	empty delay loop
750	blanks out whatever was printed in the middle of the screen
760–800	decides whether each ace has the value 1 or 11
820	C is reset to 2 for dealer
830	dealer's second card is revealed
840	if the dealer's total is greater than 16, he stands
850	if the dealer's total is greater than 11, he is hit with another card (see below)
860–890	the dealer's aces are regarded as having the value "11," so long as this leads to a total equal to or greater than 18
900	C is incremented by 1 to indicate that a card is being added to dealer's hand
910	CARD_COUNT is incremented by 1 to point to the next card to be drawn from array DECK
920	the next card from DECK is put into the dealer's hand (in matrix HAND)
930	displays the selected card in dealer's location
940	increments dealer's total by the value of the card just dealt
950	determines if dealer has busted
960	goes back to see if dealer needs another card
970	no real purpose, since this is never reached
980	this section of the program is branched to when a player asks to be hit in line 680; the value of C is incremented to show that another card will be added to player's hand
990	CARD_COUNT is incremented to point to the next card to be drawn from array DECK
1000	places the next card from array DECK into the player's hand in matrix HAND

1010	displays this card in the player's location
1020	adds the value of this card to the player's total
1030	tests to see if the player has busted
1040–1070	reached if the player has busted; the loop prints out an appropriate message, and control is returned back into the loop at line 810 to terminate this player's turn
1080–1140	if player has won, prints a congratulatory message, otherwise branches to line 1180
1150	terminates program
1160–1170	branched to if the dealer has busted; sets dealer's total to 0 (so every player is a winner) and branches up to line 1080
1180–1210	branched to if a player has lost; a message to this effect is displayed, and control is returned to the loop at line 1140

PROGRAM DESCRIPTION

Once the screen has been cleared and the welcoming message has been displayed, we reach a statement which we have not as yet encountered. It is the CALL CHAR statement, which permits the programmer to create any character of his own choice and assign it an ASCII value. Since the number "128" is not assigned to any particular character, this number has been assigned to the character represented by the peculiar-looking string of characters enclosed between quotes in line 140. This string is the hexadecimal representation of a string of binary digits which form a square the size of one character. (For a detailed description of how a shape is formed, please refer to the reference manual that comes with your computer.) For our purposes, this square character represents a playing card facing down.

In line 150, space is set aside for the deck of fifty-two cards (DECK), up to four hands of ten cards each (matrix HAND, dimensioned (4,10)), the positions of three locations on the screen (X and Y), the character representa-

tions of the thirteen different types of cards in a deck (VALUES$), and the total card values of each of four hands (SUM). If you are unfamiliar with the concept of a matrix, it is simply a representation of data using rows and columns, where the row is always mentioned first. Thus, in matrix HAND, the row represents the player number and the column represents the card's sequence number.

The data in line 190 consists of character representations for each of the thirteen types of card. Notice that the ace is represented by "A", 10 by "T", jack by "J", queen by "Q", and king by "K". These elements are read into the string array VALUES$ by means of the FOR . . . NEXT loop extending from line 160 to 180. After the screen has been cleared of the welcoming message, the FOR . . . NEXT loop beginning at line 210 reads in the two arrays "X" and "Y." These numbers represent positions on the screen which are used subsequently in the program to display the cards assigned to each player.

In line 250, the user is asked how many players are participating. Since a maximum of three is allowed (besides the dealer, played by the computer), whatever value is keyed in is tested to be sure that it is either 1, 2, or 3. If it is not, control is returned to line 260 for a correct response. As soon as a valid response is accepted, whichever key was pressed is translated into its numeric value plus 1, by subtracting 47 from its ASCII value. The reason for adding 1 to the ASCII value is to allow for the dealer. This value is stored in NUM_PLAYERS. The number of players excluding the dealer is printed in line 300.

Within the loop 310–350, the players' areas are set up. In each location, the player's number (1–3) is printed in square brackets. To locate each area, we use CALL HCHAR with values from the arrays X and Y. These arrays get their names from the cartesian coordinate system, where X, the horizontal coordinate, is specified before Y, the vertical coordinate. However, since CALL HCHAR specifies coordinates in the opposite order, array Y is specified before array X. The label DEALER is printed in the

dealer's area of the screen by the loop extending from line 360 to line 380.

A deck of fifty-two cards is generated and shuffled in the loop extending from line 390 to 460. The manner is which this is done is somewhat subtle, so you might have to read this section of the description several times before it is fully understood. First of all, the loop index I goes from 1 to 52. However, we want a sequence consisting of the numbers 1 through 13 repeated four times. This is accomplished in line 410, where in effect the remainder of the division of I by 13 is generated, gets 1 added to it so that it does not include 0, and is stored as the next element of array DECK. Next, a random number, R, is generated such that it is the index of an element of DECK which has already been assigned a value. Then, in lines 430–450, the element of DECK indexed by R is exchanged with the most recently generated element of DECK (indexed by I). The net result of all these operations (believe it or not!) is that into the array DECK is stored a shuffled deck.

The nest of FOR . . . NEXT loops extending from line 470 to line 560 deals the first two cards to each player, including the dealer. The variable CARD_COUNT, initially 0 by default, is used as an index to array DECK, pointing to the next card to be dealt (it is incremented by 1 before each card is dealt). Line 500 copies the next card into matrix HAND, where loop indexes J and I indicate the appropriate player and card-sequence position, respectively. Next, by a method known as Boolean logic (which we shall not describe here), the cumulative sum of each player's hand is calculated and stored in the array SUM. The next few lines of the program assign to variable T the ASCII equivalent of the character representation of the card just dealt and use this value to display the card's symbol in the appropriate player's area of the screen. Lines 570–580 print the dealer's hand in the dealer's location, but instead of the second card, the program displays the face-down-card character created in line 140 and assigned the ASCII value 128.

The FOR . . . NEXT loop starting in line 590 forms the meat and potatoes, so to speak, of this program. In this section, the user is given the option of being hit or of standing, regardless of his score, so long as he has not busted. If the user responds by pressing the **H** key (meaning he wants to be hit with another card), control is passed to line 980. Even though this may appear to violate one of the golden rules of BASIC programming, namely that one cannot jump out of the range of a FOR . . . NEXT loop and then jump back in again, the TI-99/4A allows one to do this, making it possible to have less cluttered loops. If the player opts to stand (by pressing the **S** key), control falls through to the next loop, which merely prints out the word STANDS in the middle of the screen, right after the message PLAYER # and the number of the player whose turn it is (printed previously). After a short delay loop, the line in the middle of the screen is blanked out.

In the next loop, beginning in line 760, each card of the player's hand is inspected. For each ace present, the program decides whether to assign it the value 1 or 11. This determination is based on the premise that the score should be as high as possible without busting.

Lines 820 through 970 represent the dealer's turn, taken after all the other players have finished their turns—they have either decided to stand or have busted. Line 830 replaces the face-down-card symbol with the dealer's second card.

Lines 980–1070 encompass the section of the program to which control passes from line 680—if the player has chosen to be hit. This section deals the player a card. If the player does not bust, control returns to line 650, where the player is again asked to enter **H** or **S**. If the player busts, a message to this effect is displayed in the player's area and control is returned to line 810, where the loop continues and another player's turn begins.

Line 1080 is reached after the dealer's turn ends. For each player who has not yet busted, the player's score is compared to that of the dealer; if the player's score is

higher, the message YOU WIN is displayed in the player's area. Otherwise, control passes to line 1180, where the message YOU LOSE is displayed.

Lines 1160–1170 are branched to after the dealer busts. The dealer's sum is set to 0, with the result that every player who has not busted wins against the dealer.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The structure of the game may be modified to allow for a full game to be played—in other words the game should play out many hands and shuffle the deck when it runs out of cards.
2. A method of scoring or betting may be implemented, so that a balance is carried from one hand to the next.
3. If a player has a preference for a different version of "Blackjack," this program may be amended to approximate that version as closely as possible.
4. Perhaps some sound effects would help to heighten the player's interest in the game.
5. The game may be modified to allow for more than one deck—a sophisticated way to stop card counting and other professional gambling tricks from running the casino broke.

PROGRAM 8

"ROULETTE"

PURPOSE

The game presented here is a computer version of roulette, a popular gambling game. In the game, a ball is rolled onto a spinning disk divided into little compartments. When the spinning finally stops, the ball rests in one of them. All the compartments are of various colors and are marked with numbers. There are seventy-four in all—the numbers 1–36 in black and red and 0 and 00 in green. The player may bet that the ball will land on any number, on an odd or even number, or on the colors red, black, or green. The odds that are obtained by each are shown below:

Bet	Odds
any #	36:1
red	2:1
black	2:1
green	18:1
odd	2:1
even	2:1

The object of the game is to make as much money as possible. In our computer world, there are no limits to how

much you can make, but the casino has seen fit to impose a table limit of $5.00 minimum and $500.00 maximum, so that you can't win as fast as you might like.

```
100  REM ROULETTE
110  CALL CLEAR
120  DIM COLOR$(3)
130  COLOR$(1)="RED"
140  COLOR$(2)="BLACK"
150  COLOR$(3)="GREEN"
160  MATCH$="RBGOEQ"
170  PRINT "WELCOME TO THE CASINO!":::"HERE
     IS THE ROULETTE TABLE"::
180  INPUT "WHAT IS YOUR NAME? ":NAME$
190  PRINT ::"WOULD YOU CARE FOR
     SOME▲▲▲▲▲CREDIT, ";NAME$;
200  INPUT LOAN$
210  IF SEG$(LOAN$,1,1)="Y" THEN 250
220  IF SEG$(LOAN$,1,1)<>"N" THEN 200
230  PRINT "IF YOU DON'T WANT TO PLAY,▲▲GET
     OUT YOU BUM."
240  END
250  PRINT ::"WELL, ";NAME$;
260  IF DEBT>4999 THEN 870
270  INPUT "HOW MUCH WOULD YOU LIKE
     TO▲▲BORROW? ":LOAN
280  IF DEBT+LOAN<=5000 THEN 310
290  PRINT "SORRY, THAT EXCEEDS
     YOUR▲▲▲▲CREDIT LIMIT."
300  GOTO 270
310  DEBT=DEBT+LOAN
320  CASH=LOAN+CASH
330  PRINT ::"O.K. ";NAME$:"TAKE A SEAT"::
340  PRINT "YOU HAVE $";CASH::
350  INPUT "WHAT ARE YOU BETTING ON?▲▲▲▲(RED,
     BLACK, GREEN, ODD,▲▲▲▲EVEN,
     1-36, 0, 00 OR QUIT): ":BET$
```

```
360  RANDOMIZE
370  IF SEG$(BET$,1,1) = "Q" THEN 770
380  IF POS(MATCH$,SEG$(BET$,1,1),1) THEN 410
390  IF (ASC(BET$)<48)+(ASC(BET$)>57)THEN 350
400  IF (VAL(BET$)<0)+(VAL(BET$)>36)THEN 350
410  INPUT "HOW MUCH ARE YOU BETTING▲▲▲▲(MIN
     = $5, MAX = $500) : ":BET
420  IF BET=0 THEN 340
430  IF (BET<5)+(BET>500)THEN 410
440  IF BET>CASH THEN 190
450  IF BET=INT(BET)THEN 480
460  PRINT "WE DON'T DEAL IN
     SMALL▲▲▲▲▲▲CHANGE HERE."
470  GOTO 410
480  CASH=CASH-BET
490  R=INT(RND*74)+1
500  R1=R
510  COLOR=1-(R>36)
520  R=R-INT(R/36)*36+1
530  ODD=INT(R/2)<>R/2
540  EVEN=(ODD=0)
550  SPIN$=STR$(R)
560  IF R1<73 THEN 610
570  COLOR=3
580  SPIN$="0"
590  IF R1<74 THEN 610
600  SPIN$="00"
610  PRINT ::SPIN$;"▲";COLOR$(COLOR)::
620  IF (BET$=SPIN$)+(BET$="0")*ODD+
     (BET$="E")*EVEN THEN 660
630  IF SEG$(BET$,1,1)=SEG$(COLOR$(COLOR),
     1,1) THEN 660
640  PRINT "YOU LOSE SUCKER!"::
650  GOTO 340
660  PRINT "YOU WIN!!!"
670  FOR I=1 TO 10
680  CALL SOUND(100,I*110,0)
690  NEXT I
```

```
700 IF (ASC(BET$)>47)*(ASC(BET$)<58)THEN 750
710 CASH=CASH+BET*2
720 IF SEG$(BET$,1,1)<>"G" THEN 340
730 CASH=CASH+BET*16
740 GOTO 340
750 CASH=CASH+BET*36
760 GOTO 340
770 CASH=CASH-DEBT
780 ON SGN(CASH)+2 GOTO 790,830,840
790 PRINT "YOU WILL REPAY YOUR DEBT OF $";
    ABS(CASH)
800 PRINT "AT 20% INTEREST -- PER WEEK."
810 PRINT "IF YOU DON'T LIKE THE TERMS
    SPEAK TO MY RIGHT HAND MAN▲▲-- GUIDO
    THE BLADE.":
820 PRINT "PLEASANT DREAMS..."
830 END
840 PRINT "CONGRATULATIONS! YOU'RE ONE OF THE
    ONLY PEOPLE TO HAVE▲▲COME OUT SOLVENT."
850 PRINT "YOU CAME AWAY WITH $";CASH
860 END
870 PRINT "YOU HAVE REACHED YOUR
    CREDITLIMIT. NOW GET OUT."
880 GOTO 770
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears screen
120–150	reserves space for COLOR$, which contains the names of the colors
160	sets the variable MATCH$ to contain the various options
170–180	prints welcoming message and requests the player's name
190–200	asks if the player wants a loan
210	if yes, requests the amount
220	if not no, an inappropriate response has been typed in, goes back and asks again
230–240	if the player does not want a loan, shows him or her the door
250–260	prints player's name and if credit is overextended, goes to threat routine
270	asks the desired amount
280	if the amount asked when added to the current debt is still under the player's credit limit, adds it to the debt and gives the player the cash that was asked for
290–300	if player is overextended, gives a message and asks for a lower loan
310	adds the new loan to the total debt
320	gives the player the cash that was borrowed
330–350	asks player for the desired bet
360	reseeds the random number generator
370	if player is quitting, goes to quit routine
380	if the bet typed is one of the allowable letter choices, asks for the amount of the bet
390	if the bet was not an allowable letter choice and is not a number, requests another bet
400	if the number is not in the valid range, requests another bet
410–420	requests the amount of the bet; if the amount is

0, goes back to ask for another bet

430 if the bet is outside the table limits, goes back and requests another

440 if bet is greater than cash on hand, asks for more credit

450–470 only accepts bets in whole dollars

480 deducts bet from cash on hand

490–500 selects a random number from 1–74 to simulate spinning of the wheel

510 determines if color is red or black

520–540 determines the actual number from the random number computed and also if the number is odd or even (0 is considered odd)

550 places the number generated in SPIN$

560–600 sets SPIN$ and COLOR for the special cases 0 and 00

610 displays the results of the spin

620–630 if the bet and the result match, goes to winning routine

640–650 if not, prints losing message and goes back for another bet

660–690 displays winning message with sound effects

700 if the successful bet was a number, pays at odds of 36:1

710 otherwise pays at 2:1

720 if the bet is not green, goes back for another bet

730–740 if the bet was green, pays off at 18:1 (2:1 was paid off in line 710 and the 16:1 is additional)

750–760 routine to pay at odds of 36:1

770–860 routine to calculate financial condition upon quitting

770 subtracts the debt incurred from cash on hand

780 decides if player is in debt (goes to 790), even (goes to 830), or ahead (goes to 840)

790–830 routine to print out threat message (player is in debt)

840–860 routine to congratulate player on having any money

870–880 routine to inform player that the credit limit has been exceeded; goes to threat message afterwards

PROGRAM DESCRIPTION

After clearing the screen in line 110, space is set aside for the three color names in line 120 and the array is filled with the names. This is done so that a color number (simpler for the computer to manipulate) can be generated and used to subscript COLOR$ to get the color name only when printing it out. In line 160, the variable MATCH$ is set to the possible non-numeric options that exist for a bet. For example, a bet of red is given by R, a bet of even by E and so on.

A message is then printed out inviting the player to make a bet. The player has the option of betting on red, black, green, odd, even, a whole number from 1 to 36, 0 or 00, or quitting. The non-numeric options (such as RED, GREEN, etc.) may be specified by the first letter or the whole word. After reseeding the random number generator and testing for a "bet" of Q (signifying the player's desire to quit), the program goes on to line 380 where we encounter an expression that we have not yet discussed. The POS function searches for an occurrence of one string within another and returns the position of the smaller string or 0 if it does not occur. For example

```
PRINT POS("BINGO","IN",1)
```

tests for any occurrence of IN within BINGO. The third value specified indicates the point at which the search should begin. In the above example, the value printed would be 2, since the second character of the string BINGO is the start of the substring IN. Contrast this with the following example:

```
PRINT POS("BINGO","IN",3)
```

This time, the "search string" ("IN") is not found because the search starts too far to the right (at the third position). As a result the value printed is 0, indicating that the substring was not found. As a final example

```
PRINT POS("TEST STRING","FIRE",1)
```

displays the value 0 because the string FIRE is not found as a substring of TEST STRING.

In the program, this function is used to advantage in line 380, where we test to see whether the single-letter option as expressed by

```
SEG$(BET$,1,1)
```

is found in the option list contained in MATCH$. If POS returns a nonzero value, a valid option has been typed and control is sent to line 410. Otherwise, control drops to line 390, which determines whether a number was typed in. This is done by comparing the ASCII value of the first character in BET$ with 48 (the ASCII value of 0) and 57 (the ASCII value of 9). If the option turns out to be an illegal letter or number, control is sent back to line 350 and a new bet is requested. Otherwise, the program goes on to request the amount of the bet.

After placing the amount of the player's bet into the variable BET (line 410), a special test is made to see if the amount of BET is equal to 0. If BET is indeed 0, control is sent back to line 340 to allow the player to correct any mistakes made in the bet. Line 430 insures that the bet is within the table limits set by the casino. If the bet is not within those ranges, control is sent back to line 410, where the player is again asked for the amount of the bet. In line 440, a test is made to determine if the player's bet exceeds the amount of cash on hand. If it does, control is sent to the routine that allows the player to request further credit.

In line 450, a test is made to pass only whole-dollar bets. This is done by taking advantage of the INT function, which returns the largest integer that is less than or equal to the argument. For example

```
PRINT INT(4.2)
```

displays the value 4, while the statement

```
PRINT INT(5)
```

displays the value 5. It is easy to see that the statement

```
450  IF BET=INT(BET)THEN 480
```

will transfer control to line 480 only if the truncated value of BET is equal to BET itself. In other words, X must have no fractional portion—which is another way of saying that it is a whole number. If BET is not an integer, an appropriate message is displayed and control is sent to line 410, where a new amount is requested.

After taking the money for the bet in line 480, the roulette wheel is spun by selecting a random number between 1 and 74. Line 500 stores a duplicate copy of the value in the variable R1, since the variable R is later changed for other purposes.

In line 510, the value of COLOR is set to either 1 or 2 (representing red or black respectively). This works because the expression (R>36) actually returns a numeric value—0 if false (if R is less than or equal to 36) and −1 if true (R is greater than 36). If the expression is false, COLOR is set to 1. If it is true, COLOR is set to 1 − (−1), which is 2.

Line 520 takes the remainder of R / 36 and adds 1, yielding the desired numbers (1–36) from the randomly generated number R. For the time being we ignore the possibility of the number being greater than 72, which will be handled in line 560.

Following the setting of the number, the flags ODD and EVEN are set. Again taking advantage of the fact that true is considered −1, and false 0, the numeric variable ODD is set to true if R / 2 is not a whole number. The reverse case, EVEN, could have been determined by the statement

```
540  EVEN = INT(R / 2) = R / 2
```

but instead we took advantage of the fact that if a number is odd, it cannot be even. We therefore took the reverse of ODD. If, in line 540, ODD is equal to 0, EVEN takes on the value −1 (true). If, on the other hand, ODD is equal to −1, EVEN takes on the value 0 (false). After we set SPIN$ to the number stored in R, line 560 checks to see if R1 (the original random number) is greater than or equal to 73. If it is, the number is either 0 or 00 and this special case is handled in lines 570–600.

Lines 610 to 630 print out the spin of the wheel and determine if a match is found with BET$. If it is, control is sent to the routine in line 660, which will be described in a few moments. If no match is found, control drops to line 640, where a losing message is printed before going back to line 340 for another bet.

The winning routine starts at line 660 and extends to 760. First, the winning message is displayed with some accompanying sound effects. Then, a test is made as to whether the first character of BET$ is a number. If it is, a number was correctly matched and control is sent to line 750, which pays off at 36:1. If not, a payoff of 2:1 is made. However, if green is matched, the payoff should be 18:1. Therefore, a test is made in line 720 to see if the bet was green. If not, control goes to line 340, where another bet is requested. If the bet was green, another 16:1 payoff is made, for a total (including the 2:1 payoff) of 18:1.

Line 770 is the start of the routine that is accessed when the player decides (or is forced) to quit. At that point the amount of debt is subtracted from cash on hand. In line 780, the three possibilities for cash on hand are tested for using the function SGN, which returns −1 if its argument is negative, 0 if its argument is 0, and 1 if it is positive. For example, the statement

```
PRINT SGN(-5);SGN(0);SGN(4)
```

displays the values

```
-1 0 1.
```

In 780, the expression

SGN(CASH) + 2

returns the values 1–3, depending on the value of CASH. A value of 1, indicating that CASH is negative, signifies that the player is in debt and control is transferred to line 790, where a threatening message is printed, and the program ends in line 830. (The value of the debt is actually a negative number, but is converted to a positive number by the ABS function, which returns the positive distance from 0 on the number line. In other words, ABS(−4) and ABS(4) both return 4.) If the amount of cash left is 0 (the same as that with which the player entered the casino), control is sent directly from line 780 to 830, and the program ends. If a positive value of cash is left, the lucky player has come out ahead and an appropriate message is printed in lines 840–860 before the program ends. Finally, lines 870–880 (to which control is passed from line 260 if the credit limit is exceeded) display a message to the effect that the credit limit has been exceeded and then transfers control to line 770, which prints the threatening message.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. Instead of printing the number followed by its color, the number may be printed in the color itself—in other words, draw a green 0, a red 5, etc., using the COLOR statement.
2. The program only generally approximates a true roulette wheel in terms of odds. It might be desired to set the values to the true casino odds. We have found that this wheel is more fun, however, since it is extremely positive. (It pays quite generously.)
3. The game would be much more exciting if an actual "wheel" were drawn on the screen. This may prove to be a challenging modification.
4. More sound effects can be added to the game, particularly for when the wheel is spinning.

PROGRAM

9

"CONCENTRATION"

THE GAME

This program is a computer version of the very popular card game bearing the same name. In the game, two players are pitted against each other. The fifty-two cards of the deck are laid out face down in four rows of thirteen cards each. The rows are numbered 1 through 4, while the columns are labeled A through M. In this grid system, any one of the fifty-two cards may be selected by first specifying its row and then its column. Each player goes in turn and names two cards by specifying the column and row of each with a letter and number. If the two cards match (suits are ignored in this game), they are removed from play and one point is awarded the player, who gets to go again. If the cards do not match, the other player goes. The game continues until all the cards are off the board, at which time the player with the highest score wins.

```
100   REM CONCENTRATION
110   CALL CLEAR
120   PRINT "WELCOME TO THE GAME OF "
      ::::::::::
130   I=1
```

```
140  FOR N=1 TO 13
150  I=I+I/4
160  PRINT TAB(N+6);SEG$("CONCENTRATION",
     N,1)
170  CALL SOUND(100,I*110,0,I*220,0,
     I*880,0)
180  NEXT N
190  CALL SCREEN(12)
200  CALL CHAR(128,"00FE8181818181FE")
210  OPTION BASE 1
220  DIM C(13,4),DECK(52),SCORE(2)
230  FOR I=1 TO 52
240  RANDOMIZE
250  R=INT(RND*I)+1
260  DECK(I)=DECK(R)
270  DECK(R)=I-INT(I/13)*13+1
280  NEXT I
290  CALL CLEAR
300  FOR I=1 TO 13
310  FOR J=1 TO 4
320  C(I,J)=DECK(J*13-13+I)
330  CALL HCHAR(J+J+3,I+I+2,128)
340  NEXT J
350  NEXT I
360  FOR I=1 TO 13
370  CALL HCHAR(3,I+I+2,I+64)
380  NEXT I
390  FOR I=1 TO 4
400  CALL HCHAR(I+I+3,2,I+48)
410  NEXT I
420  PLAYER=1
430  M$="PLAYER #"
440  CALL HCHAR(14,2,32,250)
450  ROW=14
460  COL=2
470  GOSUB 990
480  CALL HCHAR(14,11,PLAYER+48)
490  M$="ENTER FIRST SELECTION: "
```

```
500   ROW = 16
510   GOSUB 990
520   GOSUB 900
530   X1=X
540   Y1=Y
550   IF C(X1,Y1)<>0 THEN 590
560   CALL HCHAR(ROW,25,32,5)
570   GOSUB 1030
580   GOTO 490
590   CALL HCHAR(Y1+Y1+3,X1+X1+2,
      C(X1,Y1)+64)
600   M$="AND▲▲SECOND SELECTION:"
610   ROW = 17
620   GOSUB 990
630   GOSUB 900
640   X2=X
650   Y2=Y
660   IF (X2=X1)*(Y2=Y1)THEN 630
670   IF C(X2,Y2)<>0 THEN 710
680   CALL HCHAR(ROW,25,32,5)
690   GOSUB 1030
700   GOTO 600
710   CALL HCHAR(Y2+Y2+3,X2+X2+2,
      C(X2,Y2)+64)
720   FOR T=1 TO 500
730   NEXT T
740   IF C(X1,Y1)<>C(X2,Y2)THEN 860
750   CALL HCHAR(15,1,32,200)
760   SCORE(PLAYER)=SCORE(PLAYER)+1
770   IF SCORE(1)+SCORE(2)=26 THEN 1100
780   C(X1,Y1)=0
790   C(X2,Y2)=0
800   FOR S=1 TO 10
810   CALL SOUND(100,S*110,0)
820   NEXT S
830   CALL HCHAR(Y1+Y1+3,X1+X1+2,32)
840   CALL HCHAR(Y2+Y2+3,X2+X2+2,32)
850   GOTO 490
```

```
860  PLAYER = 3 - PLAYER
870  CALL HCHAR(Y1+Y1+3,X1+X1+2,128)
880  CALL HCHAR(Y2+Y2+3,X2+X2+2,128)
890  GOTO 430
900  CALL KEY(3,KEY,STATUS)
910  IF (STATUS<>1)+(KEY<65)+(KEY>78)THEN
     900
920  CALL HCHAR(ROW,25,KEY)
930  X=KEY-64
940  CALL KEY(3,KEY,STATUS)
950  IF (STATUS<>1)+(KEY<49)+(KEY>52)THEN
     940
960  CALL HCHAR(ROW,27,KEY)
970  Y=KEY-48
980  RETURN
990  FOR I=1 TO LEN(M$)
1000 CALL HCHAR(ROW,COL+I,
     ASC(SEG$(M$,I,1)))
1010 NEXT I
1020 RETURN
1030 M$="SORRY, THAT CARD IS TAKEN"
1040 ROW = 18
1050 GOSUB 990
1060 FOR T=1 TO 500
1070 NEXT T
1080 CALL HCHAR(18,1,32,30)
1090 RETURN
1100 CALL CLEAR
1110 PRINT "THE FINAL SCORE IS:":::
1120 PRINT "PLAYER #1 -->";SCORE(1)
1130 PRINT ::"PLAYER #2 -->";SCORE(2)
1140 WINNER=1
1150 IF SCORE(1)>SCORE(2)THEN 1170
1160 WINNER=2
1170 PRINT ::"PLAYER #";STR$(WINNER);
     "▲WINS."
1180 END
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–180	displays starting message with accompanying sound effects
190	sets the background color of the screen
200	creates the box character simulating the back of the cards
210	sets the lowest subscript of all arrays used in the program to 1
220	sets aside space for the arrays
230–280	shuffles the deck
290	clears the screen
300–350	copies the shuffled one-dimensional array DECK into the two-dimensional array C
360–380	draws the column coordinates (A through M)
390–410	draws the row coordinates (1 through 4)
420	starts with player 1
430	sets the string that will be printed in a moment
440	clears the area of the screen where the messages are printed
450–470	sets row and column where message in M$ is to be printed and goes to printing routine
480	prints the current player number (1 or 2)
490	sets a new message to be printed
500–510	sets the row and goes to the subroutine where the message in M$ is printed
520	calls the subroutine that extracts the values of the row and column from the player's typed letter and number
530–540	saves the row and column values
550	tests whether the card has not been removed
560–580	if it has, erases the row and column that were typed, calls the subroutine at line 1030 that prints an appropriate message, and goes back to ask for more coordinates
590	displays the card in the selected position

600–620 sets M$ to a message, sets row, and calls the subroutine to print
630 calls subroutine to input second row-and-column pair
640–650 copies the second row-column pair
660 checks if the same card is specified twice; if so, goes back and gets another second card
670 checks if the card has already been removed; if not, skips over error trap to line 710
680–700 wipes the coordinates typed, prints the message saying that the card is already chosen and goes back for another second card
710 turns over the second card
720–730 delay loop
740 if the two cards do not match, goes to 860
750 erases all printed messages at the bottom of the screen
760 gives a point to the player that found the match
770 if all matches have been found, goes to 1100 to end game
780–790 removes both cards that have been matched from array C
800–820 plays victory music
830–840 removes both cards from the screen
850 goes back to give the same player another turn
860 if player did not make match, switches to other player
870–880 turns cards face down again
890 goes back to get input from the next player
900–980 subroutine which gets a column and a row
900–910 waits until a letter between A and M is pressed
920–930 prints the letter in the current row at column 25 and converts it into a row number, which is placed in variable X
940–950 waits until a number between 1 and 4 is pressed
960–970 prints the digit in the current row at column 27 and converts it from a character to an integer, putting the result in variable Y

980	RETURN to the main routine
990–1020	subroutine to print the string in M$ at the location on the screen specified by ROW and COL
1030–1090	subroutine that notifies player that a card which has been chosen is no longer present
1030–1050	sets M$ to the desired message, sets row and calls the subroutine to print
1060–1070	delay loop
1080–1090	erases the message and returns to the main routine
1100–1180	prints final score, determines who the winner is, and ends the game

PROGRAM DESCRIPTION

When the program is first RUN, the screen clears and the standard welcoming message and sound effects are generated. Next, the screen is set to light yellow and ASCII number 128 is defined to be the shape of a card (face down). OPTION BASE is set to 1 and three arrays are set up. DECK will contain the shuffled deck and will then be copied into C, which represents the playing surface. The array SCORE contains the scores of the two players. In lines 230–280 the deck is shuffled and, after the screen is cleared, it is copied into the array C and printed out at the same time in lines 300–350. The variables I and J are very convenient for subscripting C but it is also necessary to have a subscript for DECK. In line 320, a standard method is used to convert the values I and J to a single value, which is different for all possible combinations of I and J. Once copied, line 330 draws a card symbol in the location on the screen corresponding to the position in C that was just filled. The reason for the seemingly complex expressions

```
J + J + 3
```

and

```
I + I + 2
```

is that the rows and columns are double spaced. We could have written the expressions as

```
J * 2 + 3
```

and

```
I * 2 + 2
```

but the computer performs addition faster than multiplication so that a little speed is gained by the first form. The reason for adding the 2 to the column position of each card is that the leftmost two columns do not appear on most television sets. As a result, the output must be pushed over a few spaces so that it is centered on the screen. In addition, the left side of the screen must contain the row numbers (see below). The row position is increased by 3 for aesthetic reasons and because the column letters have to fit above the cards.

Lines 360–380 display the column letters A through M across the top of the screen. Similarly, lines 390–410 display the row numbers along the left side of the screen. We then make the variable PLAYER equal 1, signifying the first player's turn. The bottom half of the screen is cleared starting at row 14, and the message

```
PLAYER #
```

is printed in row 14 using the special printing subroutine in line 990. The player's number is then printed and a message is printed requesting the first selection, again using the subroutine at line 990. The subroutine at line 900 is then called to wait for the player to type a valid letter A to M, followed by a number 1 to 4. The letter is converted to a number and placed in variable X. The number entered is placed in the variable Y. Since the subroutine is called twice, the first values of X and Y would be lost if they were not copied into the storage variables X1 and Y1. Next, the location in C specified by X1 and Y1 is tested to determine if there is still a card in that position. If there is (indicated by the fact that C(X1,Y1) is not equal to 0) control is passed to line

590. Otherwise the selection typed is cleared from the screen, the subroutine at 1030 is called to print a warning, and control returns to 490 to ask for the first selection again.

As we have said, if the first position specified contained a card, control is passed to line 590, which turns the card face up (displays the card value as a letter A to M). A prompt requesting the position of the second card is then printed and an almost identical process to that just described is performed on it. The only exception is line 660, which checks whether the two positions are the same. If they are, control goes back to line 630, which waits for the second selection to be inputted again.

Lines 720–730 comprise a delay loop to allow the players time to realize what is going on. Immediately following in line 740, a test is made to see whether the cards match. If they do not, control is passed to line 860. If they do, control drops to line 750, where the lower part of the screen is cleared and 1 is added to the score of the player that matched the cards. If the scores of the two players combined equals 26, all the cards have been removed and the game is over. Control is passed to line 1100 for the final goodbye. Otherwise, the two cards are erased from the matrix so that they cannot be picked again and a victory roll is played. The two cards are then erased from the screen and control is sent back to line 490 so that the same player gets another chance.

Line 860 is reached only if the two cards chosen do not match. The statement on this line switches between players. If the value of PLAYER is 1, the result is 3 − 1, or 2. If, on the other hand, the value of PLAYER is 2, the result is 3 − 2, or 1. Following the switching of players, the cards are "flipped over" again by the statements in lines 870–880. Control is then passed back to line 430, which prints out the new player number and goes on with the game.

Lines 900–980 contain the subroutine to enter a letter and number coordinate. The values are returned in the variables X and Y. Lines 900–930 wait for a valid letter, print

it in the twenty-fifth column of the row specified by ROW, and convert it to a number from 1 to 13, placing it in the variable X. A similar procedure is followed for the row number, which is printed and converted to a number between 1 and 4 that is placed in Y. Lines 990–1020 encapsulate a short subroutine to print the message stored in the string variable M$ at the position specified by ROW and COL.

Lines 1030–1090 print a message saying that the card that has been chosen has already been removed from the screen. They use a delay loop to leave the message on the screen for a moment and then erase it and return to the main routine. Finally, lines 1100–1180 are reached at the end of the game, where the screen is cleared, each player's score is printed out, and the winner is determined.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. Instead of typing in the row and column numbers, make a system to move around a blinking cursor which can be used to select a card at the push of the enter key. If joysticks are available, they may be used to control the motion of the cursor, and the button can select the desired card.
2. Turn the game into a one-player game by making the computer your opponent. Warning: This is not as easy as it sounds. The computer, having a perfect memory, will almost never lose. However, the situation can be altered by introducing a random factor to determine whether it can remember the value of a given card. Its percentage of recall could even be adjusted to different levels based on how well its opponent plays.
3. We have used the letters A through M to represent the cards for convenience only. You might like to change the representation to the normal ace, 3, jack, etc. (See "Blackjack" for ideas on how to represent these cards.)

PROGRAM

10

"HIDDEN WORD SEARCH"

THE GAME

This program generates a hidden-word puzzle. The words are selected from a series of DATA statements. Once selected, they are displayed on the screen. The object of the game is for the player to find these words on the screen. However, this is not as easy as it may appear, since the words may be displayed in any one of eight directions—left to right, right to left, vertically in either direction, or diagonally in all possible directions. None of the words on the screen overlap, so there is no sharing of letters. Moreover, to make the task of recognizing the words even more difficult, all the empty spaces on the grid are filled with random letters of the alphabet. Up to thirty-five to sixty words can be placed in one puzzle, depending on the length of the words. Using the words shown in the listing, the maximum is about thirty-nine.

```
100 REM HIDDEN WORD SEARCH
110 CALL CLEAR
120 PRINT "THIS IS THE HIDDEN
    WORD▲▲▲▲▲SEARCH GAME ...":::::::::::
130 RANDOMIZE
```

```
140 OPTION BASE 1
150 LC=2
160 RC=31
170 DIM WORDS$(100),SCRAMBLE(32,24)
180 N=39
190 FOR I=1 TO N
200 READ WORDS$(I)
210 R=INT(RND*I)+1
220 TEMP$=WORDS$(R)
230 WORDS$(R)=WORDS$(I)
240 WORDS$(I)=TEMP$
250 NEXT I
260 DATA "INTRIGUE", "PLAGUE", "TERROR",
    "DARKNESS", "QUEUE", "EXIT", "VARNISH",
    "CLONE", "SANDWICH", "SAUCER",
    "AIRPLANE"
270 DATA "CUP", "VANQUISHED", "DEMOLISH",
    "ATTACK", "LOVE", "WARMTH", "COMPUTER",
    "OCEAN", "RAZOR", "GENIUS", "SEWER"
280 DATA "FRENZY", "CONNOTE", "DEVIOUS",
    "REALITY", "SPACECRAFT", "MISSION",
    "ALTER", "RADIOACTIVE", "PROGRAM",
    "DELIMIT"
290 DATA "ASSUAGE", "SAUSAGE", "MISSILE",
    "REMIT", "CONTRACT", "RETREAD",
    "INVOKE", "COMMAND"
300 PRINT "HOW MANY WORDS SHOULD
    I▲▲▲▲▲SCRAMBLE? (";N;" MAXIMUM):";
310 INPUT "":NWORDS
320 IF (NWORDS<1)+(NWORDS>N)+(NWORDS<>
    INT(NWORDS))THEN 300
330 PRINT ::::::"PLEASE HAVE PATIENCE, I
    AM▲▲WORKING ON IT."::::::
340 PRINT "THINKING▲.";
350 FOR W=1 TO NWORDS
360 C=0
370 CALL SOUND(25,(NWORDS-W+1)*220,0)
380 TEMP$=WORDS$(W)
```

```
390  L=LEN(TEMP$)
400  DX=INT(RND*3)-1
410  DY=INT(RND*3)-1
420  IF (DX=0)*(DY=0)THEN 400
430  RX=INT(RND*30)+1
440  RY=INT(RND*24)+1
450  PRINT ".";
460  C=C+1
470  IF C>500 THEN 820
480  IF C - INT(C/50)*50 = 0 THEN 400
490  EX=RX+L*DX
500  EY=RY+L*DY
510  IF (EX<LC)+(EX>RC)+(EY<1)+(EY>24)THEN 430
520  FOR I=1 TO L
530  IF SCRAMBLE(RX+I*DX,RY+I*DY)<>0 THEN 430
540  NEXT I
550  FOR I=1 TO L
560  SCRAMBLE(RX+I*DX,RY+I*DY)=
     ASC(SEG$(TEMP$,I,1))
570  NEXT I
580  NEXT W
590  CALL CLEAR
600  FOR I=LC TO RC
610  FOR J=1 TO 24
620  IF SCRAMBLE(I,J)<>0 THEN 650
630  CALL HCHAR(J,I,INT(RND*26)+65)
640  GOTO 660
650  CALL HCHAR(J,I,SCRAMBLE(I,J))
660  NEXT J
670  NEXT I
680  CALL KEY(3,CHAR,STATUS)
690  IF STATUS<>1 THEN 680
700  IF CHAR = ASC("X") THEN 800
710  IF CHAR<>ASC("H")THEN 680
720  FOR I=LC TO RC
730  FOR J=1 TO 24
740  IF SCRAMBLE(I,J)<>0 THEN 760
750  CALL HCHAR(J,I,42)
```

```
760  NEXT J
770  NEXT I
780  CALL KEY(3,CHAR,STATUS)
790  IF STATUS=0 THEN 780
800  CALL CLEAR
810  END
820  PRINT ::::::::"SORRY, BUT THE BOARD IS
     TOO CROWDED. TRY AGAIN"::
830  FOR I=LC TO RC
840  FOR J=1 TO 24
850  SCRAMBLE(I,J)=0
860  NEXT J
870  NEXT I
880  GOTO 300
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–120	clears the screen and prints welcoming message
130	reseeds random number generator
140	sets the lowest subscript of all arrays used in the program to 1
150–160	sets the leftmost and rightmost columns used based on the capacity of the TV
170	sets aside storage for the arrays
180	sets the variable N to the number of words stored in data statements
190–250	reads in and randomly shuffles the words
260–290	DATA statements containing words
300–310	prompts user and inputs the number of words to place in the puzzle
320	checks if the number of words is in the valid range and an integer; if invalid goes back and asks for a new number
330–340	machine excuses itself for taking so long
350–580	main loop to place all words within the matrix

360 sets the count of placement tries equal to 0
370 makes a tone at the start of each new word
380–390 sets TEMP$ to the current word and L to its length
400–420 sets X and Y direction increments; if both are 0, tries again
430–440 sets the random position for the beginning of the word
450 prints a dot every time a position is tried
460 increments the try number
470 if 500 tries have been made, calls it quits
480 if 50 tries have been made, tries another direction
490–500 determines the position of the end of the word
510 if either position is out of the valid range, goes back and tries another position
520–540 loop determines if any position overlaps with an already existing word; if so, goes back and tries another starting position
550–570 places the word in the matrix
580 tries placing the next word
590 clears the screen
600–670 prints the words on the screen, filling in with random letters
620 checks if current position is occupied; if so, skips to 650
630–640 prints a random letter and goes on to the next position on the screen
650 prints the letter occupying the current position of the matrix
660–670 moves on to the next position
680–690 waits until a key is pressed
700 if the key is an X control, passes to the routine that clears the screen and ends the program
710 if the character is not H, waits for another character
720–770 if H was pressed, reveals the words
740–750 changes camouflage letters to asterisks
780–790 waits until a key is pressed

800–810 routine clears screen and ends
820–880 control is sent here when the program is unable to fit all the words in the matrix
830–870 zeroes out the matrix in preparation for a new try
880 goes back to ask again for the number of desired words

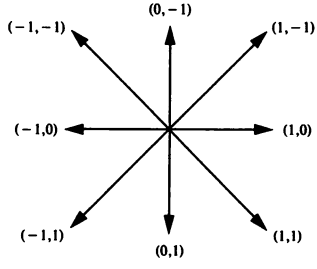
PROGRAM DESCRIPTION

Immediately following the normal setup procedure, the two variables LC and RC are set to 2 and 31, respectively, in lines 150–160. They represent the leftmost and rightmost columns visible on the television screen. Depending on the quality of the particular television, these values may have to be adjusted. Line 170 sets aside storage for the words to be placed in the puzzle and the matrix containing the puzzle. The value of N, set in line 180, reflects the number of words stored in data. If words are added or deleted the value of N should be updated. Note that there is a limit to how many words can be placed in the puzzle.

Lines 190–250 read in and randomly reorder the array WORDS$. The DATA statements follow immediately afterwards. The user is then asked how many words should be placed in the puzzle. The maximum allowed is the number of words stored in data, but as we mentioned, the puzzle is all too finite. With words of the same length as the ones shown, the maximum possible number of words is approximately forty. The next lines ensure that the number is valid. If it is not, the program goes back and asks again. If it is, the program begins to place the words in the matrix.

The loop extending from 350–580 places all the words in the matrix. With each new word, the count C is designed to keep track of how many times the program tries unsuccessfully to fit in a word. If this number reaches 500, the program gives up and ends. A tone is played for every word when the program begins to attempt to fit it in. TEMP$ is set to the current word and L is set to its length, after which the real work of placing the word begins. First, a random

direction is chosen. This is done by generating two values DX and DY—one for the horizontal direction and one for the vertical. The value −1 means to the left or up, depending on its use (in DX or DY). The value 1 means to the right or down (again, depending on its use). The value 0 means no change of column or row. Thus with the combination of DX and DY, all eight directions are randomly generated.



The unacceptable case is when both DX and DY are 0. If this occurs, control is sent back to line 400 where DX and DY are chosen again. Next, RX and RY are randomly generated to be the tentative position of the beginning of the word. Then, a dot is printed and C is incremented to show that the computer is trying to place a word. If 500 tries have been made on a single word, the cause is given up for lost, and control is transferred to line 820. If not, control drops and a second test is made—this time as to whether 50 tries have been made. If they have, control is sent back and the direction and starting point of the word are recomputed. Another 50 tries can be made before the direction is again changed.

The variables EX and EY are set to the positions corresponding to the end position of the word. They are then compared to the screen boundaries. If the word goes over the boundaries, the program goes back and tries again for a new starting position. Similarly, the next three lines test to see if any letter within the word would be within a space already occupied. If so, we also go back to a new starting position. If the above tests are all passed successfully, the

ASCII values of the letters in the word are placed in the puzzle and the program proceeds to the next one. After all words have been placed, the matrix is printed out. Any space that is unoccupied by a letter is printed as a random letter to camouflage the words in the matrix. The program now waits for the player to hit a key on the keyboard. If an **X** is pressed, control passes to line 800, which clears the screen and ends the program. If **H** is pressed, the words are revealed by changing all the camouflage letters to asterisks (this is accomplished in lines 720–770). In line 780, the computer waits for the player to press any key before clearing the screen and ending the program. The next few lines are a routine that is reached only if the program cannot fit all the words into the matrix. A message is printed, the matrix SCRAMBLE is zeroed, and control goes back to line 300 to ask for a new word count.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. Make it possible for words to overlap (share letters in common). Warning: this is not as easy as it may appear.
2. Just like the game "Scramble," this game can be turned into a two-player game by keeping score and by allowing each player to type in the words that the other player must find.
3. The length and the number of words may, of course, be modified to adjust the difficulty level.
4. The character set may be duplicated using the CALL CHAR statement. Twenty-six ASCII codes would be defined as the duplicate capital letters and used only as the camouflage letters. Then, when **H** is pressed, instead of having to redraw the whole screen, the twenty-six ASCII codes could be quickly redefined as asterisks. The screen would be immediately updated. (For an example of this technique used in a different way, see "Changing Patterns" below.)

PROGRAM

"CHANGING PATTERNS"

PURPOSE

This program is meant entirely as a feast for the eyes. It makes use of the TI-99/4A's graphics capabilities to display an endless series of quilt-like patterns on the screen. Like a kaleidoscope, each pattern is different. No input is required; you just sit back and watch.

```
100  REM CHANGING PATTERNS
110  CALL CLEAR
120  PRINT "CHANGING PATTERN
     GENERATOR"::::::
130  PRINT "SETTING UP PATTERN -- PLEASEBE
     PATIENT ..."::::::::
140  DIM A$(12),B$(12)
150  T$="AHAHAHAHAHAHAHAHAHAHAHAHAHA"
160  HEX$="0123456789ABCDEF"
170  FOR I=1 TO 12
180  RANDOMIZE
190  FOR J=1 TO 16
200  A$(I)=A$(I)&SEG$(HEX$,INT(RND*16)+1,1)
210  B$(I)=B$(I)&SEG$(HEX$,INT(RND*16)+1,1)
220  NEXT J
230  PRINT T$
```

```
240  PRINT SEG$(T$,2,28);"H"
250  NEXT I
260  FOR I=1 TO 12
270  X=INT(RND*16)+1
280  Y=INT(RND*16)+1
290  CALL COLOR(5,X,1)
300  CALL COLOR(6,Y,1)
310  CALL SCREEN(INT(RND*16)+1)
320  CALL CHAR(65,A$(I))
330  CALL CHAR(72,B$(I))
340  FOR T=1 TO 500
350  NEXT T
360  NEXT I
370  GOTO 260
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears screen
120–130	introductory message, tells user to wait as screen is set up
140	sets aside space for character-definition strings
150	T$ is assigned pattern for line of screen
160	HEX$ is assigned hexadecimal digits
170–250	sets up screen and designs random characters
180	reseeds random number generator
190–220	for one element of A$ and one element of B$ (indexed by I), randomly selects and concatenates 16 hexadecimal digits from HEX$
230–240	prints two lines on the screen: one starting and ending with A, the other starting and ending with H
250	generates next 11 characters and lines
260–360	generates 12 patterns on the screen
270–280	chooses two random numbers, 1–16
290–300	colors A and H as specified by the two random numbers

310	colors the screen (background) according to a random number from 1 to 16
320–330	redefines the ASCII numbers for A and H as characters specified by the ASCII strings in A$ and B$
340–350	delay loop
360	next pattern
370	goes back to do another 12 patterns

PROGRAM DESCRIPTION

This program sets up a pattern on the screen of alternating characters by printing rows of A's and H's such that every A has an H on either side of it, above it, and below it; likewise, every H has an A on either side, above, and below. (The only exception to this rule is at the edges of the screen.) The next step is to replace these characters with randomly generated ones. By this we mean that the replacement characters are not "normal" characters—letters, numbers, punctuation—but random dot patterns that can be created using the CALL CHAR command. As soon as the ASCII numbers for A and H are redefined with the CALL CHAR command, the A's and H's on the screen immediately change to the replacement "characters" that have just been assigned to their ASCII numbers.

After the screen has been cleared, an introductory message is printed and A$ and B$ are DIMensioned. A$ holds twelve strings of sixteen characters each. Each string will hold a series of hexadecimal digits used as patterns in the CALL CHAR command that redefines the ASCII number for A. B$ similarly holds twelve strings used to assign new characters to the ASCII number for H (perhaps this string array should have been called H$). T$ is assigned a string which is printed as a line on the screen. HEX$ is assigned a string containing all the hexadecimal digits.

In the FOR . . . NEXT loop from line 170 to line 250, A$ and B$ are filled with the patterns for random characters,

and in the same loop the screen is filled with A's and H's. After the random number generator is reseeded, a loop is entered (lines 190–220) which fills string arrays A$ and B$. For the current element of A$ and the current element of B$ (the value of I indicates which element), the loop goes around sixteen times, each time concatenating another hexadecimal digit to the element of A$ and a hexadecimal digit to the element of B$. These digits are produced by applying the SEG$ function to the string HEX$, using the RND function to determine which character in HEX$ is to be selected.

The contents of T$ are printed (in line 230) as one row of the screen. However, it cannot be used on every row, because then every column would contain the same letter all the way down, while what we want is alternating letters. Therefore, in line 240, what is printed as the next row is T$ with its first character (an A) chopped off, followed by an H in the last position of the row. Instead of "AHAHAH . . . HAHA", this row is "HAHAHA . . . AHAH". Every odd row on the screen contains the contents of T$, every even row contains the alternate form as printed by line 240 of the program.

The main body of the program begins in line 260. It is comprised mostly of a FOR . . . NEXT loop extending to line 360. Each time through the loop, two random numbers X and Y are chosen, between 1 and 16. These numbers are used to define the colors of characters on the screen. The CALL COLOR command takes some explanation. The printable characters (ASCII numbers 32–127) must be thought of as divided into twelve groups of eight characters each. The space character through the apostrophe (ASCII numbers 32–39) belong to group 1, the open parenthesis through the slash (ASCII numbers 40–47) belong to group 2, 0 through 7 (ASCII numbers 48–55) belong to group 3, etc. You cannot change the color of one character on the screen; you can only change the color of a group, which means that all characters in that group that appear on the screen are changed to the specified color. Since A belongs

to group 5, while H belongs to group 6, their colors can be changed independently. Take line 290 as an example:

```
CALL COLOR(5,X,1)
```

What this means is that all characters on the screen belonging to group 5 are changed to the color specified by the value in X (a number in the range 1–16). This number has the same meaning as the value specified in the CALL SCREEN command. The third argument, 1, means that the background color of the character is transparent—that is to say, it lets the screen color, as specified by the CALL SCREEN command, show through. (Note that X can also have the value 1. This means the character is "transparent"; it looks like a blank space. This is perfectly acceptable for the program, because the chances of both X and Y being 1 at the same time, resulting in a blank screen, are very small.)

After H is assigned a color, also using the CALL COLOR command, the screen itself, which is the background of the pattern, is colored using the CALL SCREEN command. Now, the ASCII numbers normally assigned to A and H are reassigned to characters specified by the random strings of hexadecimal digits in A$ and B$. The index I of the FOR . . . NEXT loop is used to select which string elements of A$ and B$ are to be used. As soon as the CALL CHAR commands are executed, the characters on the screen change to the newly defined characters. Note that if the CALL CHAR command is used to assign a new character to an ASCII number, the new character belongs to the group specified by the ASCII number. Even when ASCII numbers 65 and 72, normally assigned to A and H respectively, are reassigned to new "characters" (actually random dot patterns), the CALL COLOR commands still work.

Lines 340–350 form an empty FOR . . . NEXT delay loop to keep the current pattern on the screen for a while. Then line 360 continues the FOR . . . NEXT loop starting in line 260, to generate a new pattern. Since A$ and B$

each only have twelve elements, this loop only goes around twelve times. Then control falls to line 370, which immediately sends control back to line 260 to start the loop all over again, repeating the same sequence of twelve "characters," but this time in different, randomly selected colors. The program is an infinite loop, creating new patterns until **FCTN 4 (CLEAR** key) is depressed, or the computer is shut off.

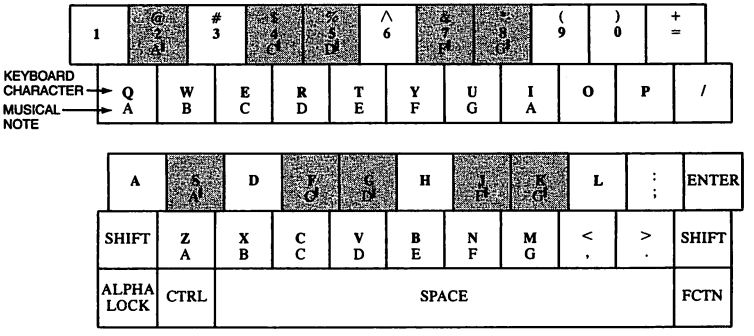
POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The program could be amended to continually generate random character patterns, not just repeat the same twelve. This would not necessarily slow the display down, because the delay loop (lines 340–350) could be shortened accordingly.
2. Instead of only two alternating character patterns on the screen, three or more could be used to add greater variety.
3. For those who do not like stopping programs with the **FCTN 4** key, a provision could be added to the program to stop after a certain number of patterns. The user could also be asked how many patterns he or she would like.
4. There are many possible variations on this program. One is to have moving patterns which also change colors. And of course there is always the addition of sound, either randomly produced tones, "chords" selected from a predefined list, or tones selected under certain musical restrictions that may create more enjoyable melodies.

PROGRAM 12 "ORGAN"

PURPOSE

This program gives you the opportunity to express yourself musically by simulating a two-octave piano or organ keyboard. The key labeled C on the TI-99/4A keyboard substitutes for middle C. Every other note in its octave is in the same relative position as may be found on a piano, as shown in the following diagram:



The note C in the second octave is obtained by pressing the W key. Again, every note in this octave is situated in the same position relative to W as the corresponding keys in

the first octave. The highest note allowed for by this program, A, is obtained by pressing the **I** key.

Notes are played one at a time in this version, but it is possible to modify this basic tool to include multiple voices, volume control, more octaves—it is only limited by your imagination and skill as a programmer.

```
100  REM ORGAN
110  CALL CLEAR
120  DIM NOTE(40),CODE(128)
130  N=25
140  DUR=100
150  VOL=0
160  FOR I=1 TO N
170  READ NOTE(I)
180  NEXT I
190  FOR I=65 TO 65+N
200  READ CODE(I)
210  NEXT I
220  CODE(50)=14
230  CODE(52)=17
240  CODE(53)=19
250  CODE(55)=22
260  CODE(56)=24
270  CALL KEY(3,KEY,STATUS)
280  IF STATUS=0 THEN 270
290  IF CODE(KEY)=0 THEN 270
300  X=NOTE(CODE(KEY))
310  IF X=0 THEN 270
320  CALL SOUND(DUR,X,VOL)
330  GOTO 270
340  DATA 220,236,250,266,275,293,311,328,
     346,365,394,417
350  DATA 440,472,500,528,560,596,622,656,
     692,730,788,832
360  DATA 891
370  DATA 0,8,4,0,16,5,7,0,25,10,12,0,11,
     9,0,0,13,18,2,20,23,6,15,3,21,1
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears the screen
120	sets aside space for the frequencies and a map of the keys required to turn them on
130–150	sets the number of notes to 25, the duration of a note to 100, and the volume to the maximum
160–180	reads in the 25 notes
190–210	reads in the value of the notes for each letter of the alphabet
220–260	sets the map of the keys on the top row (a special case)
270–280	waits for a key to be pressed
290	if no note is stored in the array CODE under that position, go back until a correct key is pressed
300	sets the variable X to the note in question
310	if the note is not one of the ones present, goes back and waits for another character
320	plays the note
330	goes back and waits for a key to be pressed
340	octave #1 data
350	octave #2 data
360	A above middle C data
370	character-mapping data

PROGRAM DESCRIPTION

After clearing the screen in line 110, room is set aside for the variables NOTE and CODE. These arrays are the backbone of the program; indeed, they are the tools that enable it to be so short. We shall be discussing them in a moment. In line 130, the number of notes is set to 25. This general technique, which we have used in many previous programs, allows for easy modification. Instead of using the number "25" everywhere in the program, we set N equal to it and then use N in its place. In this manner only one

change need be made to replace the 25 everywhere. In this particular program, the variable N is used only once, but this is somewhat atypical. In most programs (as can be verified by looking back at some of the longer ones) specific constants are used many times.

After the frequencies of the notes which are stored in the array NOTE are read in, the map pointing to those notes is read from DATA statements into the array CODE. Then five special cases of CODE are defined. This is done because there is a jump in the ASCII code numbers between the digits and the letters. This gap would have had to be filled with zeros if the index of the loop were to have been incremented continuously in the ordinary way. Since this strategy results in a waste of time, we decided to take the route of trading memory for time and specifically set all the required number keys to their correct values in the map.

The map stored in the array CODE simply points to specific notes stored in the array NOTE. Each time a particular note is required, the note number is "looked up" in the table CODE. If the value for CODE at that position is zero, no corresponding note exists for the key that is pressed. If the value is any other number, it is a subscript of NOTE. Accessing the correct position in NOTE yields the value corresponding to the desired frequency. It is this value that is used in the CALL SOUND statement which emits the actual tone.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. It is possible to fit up to four octaves on the TI keyboard if you are willing to sacrifice the similarity to the piano keyboard layout.
2. A modification can be made to allow for more than one simultaneous voice. One easy change is to have the computer play two notes—the one you press and the one whose frequency is exactly in the ratio 1.25:1 (the numeric equivalent of what musicians call a "third").

This produces a pleasant-sounding chord each time a key is pressed.

3. The program can be altered to allow the player to change the length of the notes, the volume, and any other parameters.
4. The program would be vastly improved if it could save the music as played and then play it back later. Perhaps the music could even be saved permanently in data files on cassette tape.
5. While rather difficult, a very interesting modification would be to arrange for the computer to automatically produce an acceptable harmony to any inputted melody.

PROGRAM

13

"MINDSTORMING"

PURPOSE

The next program is a sophisticated TI-99/4A variation on the popular logic game called "Mastermind." In our version, a numeric code is used in place of colors. For the benefit of those who have not yet encountered this game, let it be said that it demands a high degree of logical deduction—more so than you might believe.

The game is played as follows: One player (in this case, the computer) randomly picks out a code number consisting of a series of digits whose length is requested at the outset. The object of the game is for the other player (in other words, you) to match the code, digit for digit.

The game proceeds in turns. With each turn, the player makes a guess as to the code and is given some helpful information about the guess. First, the player is told the number of digits that are correctly matched and that are in the correct position. Then, the player is told how many digits are correct—but that are in the wrong position. This information is displayed under the abbreviated titles:

RD/RP (right digit, right position)

and

RD/WP (right digit, wrong position)

Based on this information, a good player can arrive at the correct answer in a surprisingly short number of moves. The following game illustrates some of the strategies used and also how to play.

Sample Game:
OF POSITIONS (3–10): ?**3**
OF DIGITS (3–10): ?**4**
(The digits 1, 2, 3, and 4 are allowed—assume the computer's randomized code is 241.)

TRY #	CODE	RD/RP	RD/WP
1	**112**	0	2
2	**221**	2	0
3	**421**	1	2
4	**241**	(the winning move)	

If the above looks easy, try playing with ten positions and ten digits (the **0** key is the tenth)

```
100   CALL CLEAR
110   PRINT "YOU ARE ABOUT TO TEST
      YOUR▲▲POWERS OF ...":::::::::
120   FOR I=1 TO 50
130   PRINT TAB(I-INT(I/24)*24+3);
      "MINDSTORMING";
140   NEXT I
150   PRINT :::::::::
160   DIM A(50),COPY(50),B(50)
170   INPUT "#▲▲OF POSITIONS (3-10): ":N
180   IF N<>INT(N)THEN 170
190   IF N>2 THEN 220
200   PRINT :"THAT'S NO CHALLENGE! PICK A
      LARGER NUMBER."::
210   GOTO 170
220   IF N<11 THEN 260
230   PRINT :"THAT WILL BE
```

```
      RATHER▲▲▲▲▲▲▲▲▲DIFFICULT. TRY
      AGAIN.":::
240   GOTO 170
250   PRINT ::
260   INPUT "HOW MANY DIGITS (3-10): ":C
270   IF C<>INT(C)THEN 260
280   IF C>2 THEN 310
290   PRINT :"YOU NEED MORE DIGITS
      THAN";C:"DON'T YOU WANT A FUN GAME?"::
300   GOTO 260
310   IF C<11 THEN 340
320   PRINT :"YOU CAN KEEP TRACK OF
      THAT▲▲MANY DIGITS? WHAT A MIND!▲▲▲TRY
      AGAIN -- I CAN'T HANDLE THAT MANY"::
330   GOTO 260
340   PRINT ::"SHALL I ALLOW REPEATING OF"
350   INPUT "DIGITS? (Y/N): ":REP$
360   IF SEG$(REP$,1,1)="Y" THEN 390
370   IF SEG$(REP$,1,1)<>"N" THEN 340
380   R=-1
390   IF (C<N)*R THEN 170
400   FOR I = 1 TO N
410   RANDOMIZE
420   A(I) = INT(RND*C) + 1
430   FOR J = 1 TO I-1
440   IF (A(J)=A(I))*R THEN 410
450   NEXT J
460   COPY(I)=A(I)
470   NEXT I
480   NC=1000
490   PRINT ::"TRY #▲▲CODE▲▲▲▲▲▲RD/RP RD/
      WP":"-----▲▲----▲▲▲▲▲▲-----▲-----"
500   FOR I=1 TO NC
510   PRINT STR$(I);TAB(3);"--> ";
520   FOR L=1 TO N
530   A(L)=COPY(L)
540   NEXT L
550   FOR J=1 TO N
```

```
560   CALL KEY(3,KEY,STATUS)
570   IF STATUS<1 THEN 560
580   IF KEY <> 48 THEN 600
590   KEY = 58
600   IF (KEY < 49 + KEY > 48+C) THEN 560
610   PRINT CHR$(KEY);
620   B(J) = KEY - 48
630   NEXT J
640   REM NOW FIGURE OUT THE SUMMARY
650   WH = 0
660   BL = 0
670   FOR J=1 TO N
680   IF A(J)*(A(J)=B(J))THEN 830
690   NEXT J
700   FOR J=1 TO N
710   FOR K=1 TO N
720   IF A(J)*(A(J)=B(K))THEN 880
730   NEXT K
740   NEXT J
750   PRINT TAB(20);STR$(BL);TAB(26);
      STR$(WH)
760   NEXT I
770   PRINT "GEE, YOU'RE BAD!"
780   PRINT "WHY DON'T YOU TRY SOME"
790   PRINT "OTHER GAME?"
800   END
810   PRINT "MORE DIGITS -- MORE FUN"
820   GOTO 260
830   BL=BL+1
840   IF BL=N THEN 920
850   A(J)=0
860   B(J)=0
870   GOTO 690
880   WH=WH+1
890   A(J)=0
900   B(K)=0
910   GOTO 730
920   PRINT ::"CONGRATULATIONS! YOU WIN!"::
930   PRINT "YOUR RATING IS:"
```

```
940   RATING = (N + C/2)/I*3
950   IF RATING<=1 THEN 1010
960   IF RATING>=3 THEN 990
970   PRINT "NOT BAD."::"KEEP WORKING AT IT
      THOUGH."
980   END
990   PRINT "FANTASTIC!"::"YOU DESERVE
      A▲▲▲▲NOBEL PRIZE"
1000   END
1010   PRINT "TERRIBLE"::
1020   GOTO 770
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100–150	sets up the screen
160	sets aside array storage
170–240	accepts number of positions and ensures that it is in the valid range
200–210	if position is too small, says so and asks again
220–240	tests that the number of positions requested does not exceed 10. If so, an appropriate message is printed and another number requested
260–330	accepts number of digits (colors) and validates it
340–370	asks player if repeating digits are allowed and accepts only responses beginning in Y or N
380	sets the repeating-not-allowed flag
390	if repeating is not allowed and there are more positions than digits, goes back and gets new values for position and digits
400–470	sets the random code and copies it into COPY
480	sets the maximum number of chances permitted to an outrageously large number
490	displays heading
500–760	main loop to request code and print summary information
510	prints the current try number

520–540	copies the untouched copy of the code in COPY back into the array A
550–640	scans the keyboard for each digit; validates and then displays it
650–660	initially sets number of digits correct but in wrong place and number of digits correct and in right place to 0
670–690	checks all positions for correct digits; if found, transfers control to 830
700–740	checks all digits in incorrect positions; if found, transfers control to 880
750	displays the computed information
770–800	insults bad players and ends
810–820	encourages the player to select more digits
830–870	adds 1 to count of correct in right place and checks if it is equal to the number of positions; if so, goes to winning routine in line 920; otherwise returns to keep checking
880–910	adds one to count of correct in wrong place and returns to keep checking
920–1020	prints congratulatory message and rating and terminates the program

PROGRAM DESCRIPTION

After the introductory screen antics, the player is asked to enter the number of positions and digits desired. The last question asked before the game begins is whether the digits are to repeat or not. If the answer is "no" and there are not enough digits to satisfy the need, line 390 transfers control back to the questions again and the player is given another chance to respond. If the player decides that the digits may repeat, the number of positions is immaterial because any extra positions can always be filled with repeated digits.

Immediately following the code-generating procedure, which places the code into the array A, is the main loop, in which the game is simulated. The computer sets the num-

ber of chances given to the player to 1,000. If the player has not succeeded by the one thousandth try, the computer halts the game with a mildly reproving message. Within the loop extending from line 500 to 760, the computer waits for the player's move, tests to see if the match is made correctly and, if not, displays a summary of the results.

The loop in lines 550–630 reads in the player's typed digits and validates them to make sure that only numbers between 1 and 10 (with 0 standing for 10) are typed in, one at a time. It then stores them in the array B. A check is then made in the loop extending from line 670 to 690 for an exact match of the computer's digit and the player's. If a match is found, control is transferred to line 830, where a counter is incremented and a check made to see if all the digits have been supplied. If this is the case, a congratulatory message is printed and a rating computed. If not, the computer checks for all correct digits in the wrong place (in lines 700–740). Whatever the result, the answers are tabulated by the PRINT instruction in line 750.

When the player has matched all the digits and won, the rating is computed on the basis of how long it took. Obviously, the more complex the code, the longer it will take to match all the digits, so the computer calculates the rating on a sliding scale. It simply allows the player a number of moves equal to the sum of the number of positions and half the number of permissible digits. That is, if there, for example, are six positions and four digits permitted, the maximum allowable number of moves to arrive at the solution with the top rating is 6 + (4 / 2), or 8. The result of this calculation is multiplied by 3 to arrive at the player's rating. If this proves to be less than or equal to 1, meaning that more than three times the number of moves were made as compared to the top rating (in the above case, twenty-four or more), the lowest rating is assigned. If it is greater than or equal to 3, a laudatory message is displayed and the program is terminated. A score anywhere between the above-mentioned values triggers the intermediate rating.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The program can be modified to incorporate color and possibly sound into the game.
2. An interesting twist on the game might be to have a time limit on each turn. This would prevent cool, collected thinking for long periods of time before each move.
3. You might want to amend the validation statements between lines 180 and 310 to allow for different ranges (more positions, etc.) However, the maximum of ten positions was chosen because of the limitation of the screen size—to say nothing of the human difficulty of coping with such a large value.
4. The game can be turned into a competitive two-player game, in which each gives a code for the other to solve.

PROGRAM

14

"TIC TAC TOE"

PURPOSE

Most of us became familiar with the game of "Tic Tac Toe" in early childhood. In fact, this game has been around for hundreds of years. In view of its popularity, it is no wonder that so many have attempted to write programs to simulate this game on various computers. In "Tic Tac Toe," two sets of parallel lines are drawn at right angles, resulting in nine areas into each of which an "X" or an "O" is entered. Two competing players take turns entering their respective symbols. As soon as one of the players has succeeded in completing a horizontal, vertical, or diagonal line of his symbol, he is considered the winner. Even though the game appears to be rather simple, a good player exercises considerable strategy when playing. It is clear, for example, that given a choice, a player should always opt to go first. The reason for this is that since there are only nine boxes available, if he goes first he can make five moves against the opponent's four. Not only that, but having the first move permits him to immediately take the center box, which is common to both diagonals, the center horizontal, and the center vertical lines. At the same time, good strategy demands that you always block your opponent's attempt to

achieve a line of three adjacent (identical) symbols while trying to get a line of three yourself.

The version of the game which we have written for the TI-99/4A permits the player to pit his skills against those of the computer. The human player may choose the symbol "X" or "O" and is even permitted the opportunity to select whether he or the computer goes first.

```
100  REM TIC TAC TOE
110  CALL CLEAR
120  PRINT "WELCOME TO ...":::::::::
130  CALL SOUND(2000,220,0)
140  CALL SOUND(2000,328,0)
150  CALL SOUND(2000,440,0)
160  CALL SOUND(400,560,0,417,0,692,0)
170  CALL SOUND(1000,560,0,417,0,656,0)
180  PRINT TAB(23);"TOE":::
190  PRINT TAB(13);"TAC":::
200  PRINT TAB(3);"TIC":::::
210  OPTION BASE 1
220  DIM BOARD(3,3),PTS(8,3,2),PRIO(3,3),
     CASE(4)
230  CASE(1)=6
240  CASE(2)=1
250  CASE(3)=5
260  CASE(4)=2
270  FOR I=1 TO 3
280  FOR J=1 TO 3
290  BOARD(I,J)=0
300  READ PRIO(I,J)
310  NEXT J
320  NEXT I
330  DATA 2,1,2,1,3,1,2,1,2
340  FOR J=1 TO 8
350  FOR I=1 TO 3
360  READ PTS(J,I,1),PTS(J,I,2)
370  NEXT I
380  NEXT J
```

```
390   DATA 1,1,2,1,3,1,1,2,2,2,3,2,1,3,2,3,
400   DATA 3,3,1,1,1,2,1,3,2,1,2,2,2,3,3,1,
410   DATA 3,2,3,3,1,1,2,2,3,3,3,1,2,2,1,3
420   INPUT "WANT X'S OR O'S? ":QUERY$
430   D$="X.O"
440   IF SEG$(QUERY$,1,1)="X" THEN 470
450   IF SEG$(QUERY$,1,1)<>"O" THEN 420
460   D$="O.X"
470   INPUT "SHALL I GO FIRST (Y/N)? ":QUERY$
480   COUNT=-1
490   IF QUERY$="Y" THEN 640
500   IF QUERY$="N" THEN 530
510   PRINT :"TYPE Y OR N, NOT ";QUERY$;"!"
520   GOTO 470
530   GOSUB 1120
540   PRINT ::"ENTER YOUR MOVE: ";
550   CALL KEY(3,KEY,STATUS)
560   IF STATUS=0 THEN 550
570   IF (KEY<49)+(KEY>57)THEN 550
580   MOVE=KEY-48
590   X=INT((MOVE-1)/3)+1
600   Y=MOVE-X*3+3
610   IF BOARD(X,Y)<>0 THEN 550
620   PRINT CHR$(KEY)
630   BOARD(X,Y)=1
640   FOR T=1 TO 4
650   FOR I=1 TO 8
660   SUM=0
670   FOR J=1 TO 3
680   SUM=SUM+BOARD(PTS(I,J,1),PTS(I,J,2))
690   NEXT J
700   X=-(CASE(T)=SUM+3)*CASE(T)
710   IF X=6 THEN 1270
720   IF X=1 THEN 900
730   IF X=2 THEN 950
740   IF X<>5 THEN 770
750   GOSUB 1060
760   GOTO 530
```

```
770   NEXT I
780   NEXT T
790   FOR I=1 TO 3
800   FOR J=1 TO 3
810   IF BOARD(I,J)<>0 THEN 860
820   IF PRIO(I,J)<=MAX THEN 860
830   MAX=PRIO(I,J)
840   X=I
850   X=J
860   NEXT J
870   NEXT I
880   BOARD(X,Y)=-1
890   GOTO 530
900   GOSUB 1060
910   GOSUB 1120
920   PRINT ::"I WIN!!!"::::
930   PRINT "BETTER LUCK NEXT TIME."
940   END
950   MAX=0
960   FOR J=1 TO 3
970   IF BOARD(PTS(I,J,1),PTS(I,J,2)) <> 0 THEN
      1020
980   IF PRIO(PTS(I,J,1),PTS(I,J,2)) <=MAX THEN
      1020
990   X=PTS(I,J,1)
1000  Y=PTS(I,J,2)
1010  MAX=PRIO(PTS(I,J,1),PTS(I,J,2))
1020  NEXT J
1030  IF MAX=0 THEN 770
1040  BOARD(X,Y)=-1
1050  GOTO 530
1060  FOR J=1 TO 3
1070  IF BOARD(PTS(I,J,1),PTS(I,J,2))=0 THEN
      1100
1080  NEXT J
1090  STOP
1100  BOARD(PTS(I,J,1),PTS(I,J,2))=-1
1110  RETURN
```

```
1120  REM DISPLAY THE BOARD
1130  COUNT=COUNT+1
1140  IF COUNT=5 THEN 1290
1150  PRINT ::
1160  FOR I=1 TO 3
1170  FOR J=1 TO 3
1180  K=I*3-3+J
1190  IF BOARD(I,J)=0 THEN 1220
1200  PRINT "▲";SEG$(D$,2-BOARD(I,J),1);"▲";
1210  GOTO 1230
1220  PRINT K;
1230  NEXT J
1240  PRINT
1250  NEXT I
1260  RETURN
1270  PRINT :::"YOU WIN!!"
1280  END
1290  PRINT ::"THE GAME IS A TIE."
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears screen
120–200	prints welcoming message and plays a tune
210	sets lower bound of subscripts to 1
220	sets aside space for array and matrices
230–260	assigns a value to each element of the array CASE
270–320	initializes matrix BOARD to zero and reads in the nine elements for the matrix PRIO
330	DATA statement containing the 9 elements for the matrix PRIO
340–380	reads in the elements for matrix PTS
390–410	DATA statements containing the elements for the matrix PTS

420	prompts the user for symbol of choice; the response (X or O) is stored in QUERY$
430–460	assigns a string value to D$ according to the player's choice. Any symbol other than "X" or "O" sends control back to line 420
470	player is prompted for choice as to whether he or she wants to go first; the response of Y or N is stored in QUERY$
480	COUNT is initialized to −1
490–520	if QUERY$ is not equal to either Y or N, control is sent back to line 470 for another try. A response of Y sends control to line 640, while a response of N branches to line 530
530	calls subroutine that displays board
540–580	waits for player to enter a number from 1 to 9, representing the desired box; the ASCII value of the response is stored in KEY, which is then converted to its corresponding number and stored in MOVE
590–600	converts the number in MOVE to its row-column coordinates in the matrix, the row number being assigned to X and the column number to Y
610	goes back to line 550 if the chosen space has already been selected
620	prints the chosen digit
630	puts the value 1 into the selected element of the matrix BOARD to indicate that the human player chose this space
640–780	loop checks through all rows, columns, and diagonals for four discrete "cases" or patterns of X's and/or O's that require special strategies on the part of the computer
660–690	using the data in matrix PTS to get correct coordinates, sums the 1's and −1's in a row, column, or diagonal
700–740	tests for the four cases mentioned above, branching according to the result: case 1 trans-

fers control to line 1270; case 2 sends the program to line 900; case 3 causes the program to fall through to line 750; case 4 transfers control to line 950. If none of the cases apply, control passes to line 770.

750–760 case 3; the subroutine at line 1060 is invoked, and upon return, control goes back to line 530 where the player is prompted for another move

770–780 looks at the next row or column or diagonal

790–870 none of the four cases was detected anywhere, so this loop selects the box which will provide the greatest advantage

810 if this box was already chosen, goes on to the next one

820 checks the priority of the current box; if it is not greater than the highest priority found so far, goes on to the next box

830 this is the highest-priority box found so far, so its priority is saved in MAX

840–850 saves the box's row and column in X and Y

860–870 checks the next box

880 puts the value −1 in the matrix element selected by the FOR . . . NEXT loop in lines 790–870, indicating that the computer has chosen it

890 goes back to ask the player for another move

900–940 case 2; calls the routine at line 1060, displays the board by calling the routine at line 1120, declares that the computer has won, and ends the game

950 case 4; MAX is initialized to 0

960–1020 checks through the entire row, column or diagonal (specified by I) to find the highest-priority vacant box

970 if the box is not vacant, goes on to the next box

980 if the box's priority is not the highest found so far, goes on to the next box

990–1000	stores the box's row and column in X and Y
1010	stores the box's priority in MAX
1020	checks the next box
1030	if no vacant box was found, this is not a valid instance of case 4, and so control returns to line 770 to try another row, column, or diagonal
1040	if a vacant box was found, puts the value −1 in that location of matrix BOARD to indicate that the computer has chosen it
1050	returns to line 530 to input another move from the player
1060–1110	this subroutine searches for a vacant box in the current row, column or diagonal (as indicated by the index I) and marks it as chosen by the computer
1060–1080	searches through the three boxes and branches to line 1100 when an empty box is found
1090	the program dies if no empty box is found; however, this will never happen; this line is mainly an indicator to show that control never passes to line 1100 by the loop ending normally
1100	places the value −1 in the box found to show the computer has chosen it
1110	returns to the main routine
1120–1260	routine to print the playing board
1120	REMark
1130	increments COUNT
1140	when COUNT reaches 5, all moves have been made and control passes to line 1290 to indicate a tie
1150	skips two lines
1160–1250	prints the contents of the nine boxes
1180	calculates the box's number (1–9)
1200	if the box is occupied, uses D$ to print an "X" or an "O," depending on who chose the box and the symbol
1220	the box is vacant, so its number is printed

1260	returns to main routine
1270–1280	case 1; tells the player he has won, and ends the game
1290	the game is declared to be a tie, and is ended

PROGRAM DESCRIPTION

The heading and welcoming theme are produced by the code in lines 110–200. Line 210 sets the lowest possible subscript to 1, thereby saving a little space. Line 220 then sets aside storage for the matrix BOARD which keeps track of all the moves; PTS contains the eight distinct paths by which it is possible to win (three vertical, three horizontal, and two diagonal). The array PRIO contains the values assigned to each position on the board; given two equally good moves, the computer will choose the most strategic one, always taking the center first, followed by the corners, and finally the other positions.

The array CASE is used to set the hierarchy of pattern detection strategies. First, all eight winning paths are checked to determine whether the player has already won. If so, the game is terminated. If not, a scan is made to see whether the computer has two in a row, with the third position empty. If so, it places its piece and wins the game. If both of these checks fail, the computer goes on to examine whether any lines contain two of the player's pieces in a row, with the third position empty. If so, it moves to block the opponent. If not, it goes on to make its final test; this consists of checking for any paths with one of the computer's pieces present and the remaining two empty. This being the case, the computer moves to fill the empty space with the higher priority. If none of these tests prove to be positive, the computer moves to the space with highest available priority, as defined in the array PRIO.

The mechanics of the above strategies are implemented in a few rather simple steps. First, the values of CASE(1), CASE(2), CASE(3), and CASE(4) are set in lines 230–260. The reasons for the specific values stored in these elements

will be explained in a few moments. Next the values of the array PRIO are read in. They relate to the board in the following way:

```
PRIO
      2  1  2
      1  3  1
      2  1  2
```

The higher the priority number, the more the computer will tend towards that location. This is the reason why the computer is directed towards the center of the board. You will notice that the center has the unique priority of 3, the highest priority on the board.

Lines 340–380 load in the values of PTS. There are eight paths, each consisting of three points (each point having its own horizontal and vertical component). After asking some preliminary questions in lines 420–520, the subroutine that displays the board is invoked. In lines 540–600 the player enters a move by typing the appropriate number from 1 to 9, in accordance with the following scheme:

```
1  2  3
4  5  6
7  8  9
```

Lines 590–600 convert the number entered by the player into horizontal and vertical components. Line 610 checks if the specified position has already been taken. If so, control returns to line 550, which asks for another number. If the test is successful, control drops to line 620, which prints out the number. Line 630 then sets the matrix BOARD at that position to 1, indicating (for future reference) that it is taken by the player. By contrast, the value of -1 is used to represent a place taken by the computer.

Lines 640–780 take advantage of the fact that paths filled with different combinations of these two values, 1 and -1, have different sums. If, for example, the sum of a line totals 3, the player has obviously won, because the only way that

a line can total 3 is for all three of its positions to contain a 1. In other words, the player has achieved three in a row. Similarly, the only way to arrive at a sum of -2 is if the computer has two in a row, with the third position vacant. The computer's obvious move then is to fill the vacant space with -1, thereby winning the game. The other possibilities are slightly more complex but, in essence, are treated in a similar fashion.

The computer decides which routine to jump to in lines 700–740. It is here that the four discrete steps in its strategy are executed, since T ranges from 1 to 4. Lines 790–880 are the lines of code which determine what to do as a "last resort" move. Line 890 transfers control back to line 530, which transfers control to the subroutine that displays the board and asks the player for his next move. Lines 900–940 contain the routine to which control is transferred if the computer has won the game. The selection of the optimum move is made by the code in lines 960–1050. After the selection is made, line 1050 sends control back to line 530, again displaying the board. Lines 1060–1100 cause the computer to block any line that has two of the opponent's pieces in a row. The subroutine in lines 1120–1260 adds 1 to the count of moves taken. If this count reaches 5 with neither player having won, control is passed to line 1290, where the game is declared a tie and the game is terminated. Finally, lines 1270–1280 encompass the routine that handles a win by the human player. We should point out that this routine is never reached because the computer plays flawlessly and can never be beaten, only tied.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The display shown on the screen can be modified by arranging for each move to update the board already on the screen, instead of printing a new display, as is done in this program. While this is probably more tedious to program, it will be faster and more attractive to watch.

2. The program can be enhanced by including options whereby the computer can play itself and humans can play humans.
3. Perhaps a strategy could be adopted to permit the player to withdraw an unintended move.

PROGRAM

15

"QUBIC"

PURPOSE

If you were impressed with the performance of "Tic-Tac-Toe," you will probably be flabbergasted at this game. Although it is simply a more sophisticated version of "Tic-Tac-Toe," a whole new dimension is added—for a total of three. Yes, "Qubic" is the game of three-dimensional "Tic-Tac-Toe." The board is a 4 × 4 × 4 cube and instead of requiring three in a row to win, four in a row are needed —any straight line of a length of 4 will do. Since the computer cannot display three-dimensional objects, we split the board into the four levels and show each one side-by-side on the screen. Picture the leftmost level as being the one at the bottom and the rightmost at the top. Each level is given a number from 1 to 4. Within each level, a row and a column must be specified by a number (also between 1 and 4). The line which must be typed is therefore of the form LEVEL, ROW, COLUMN.

There are more ways to get four in a row than the ways to get three in "Tic-Tac-Toe." The easiest way to point out these differences is to use the coordinate system we have just defined, so that

1, 1, 1
2, 2, 2
3, 3, 3
4, 4, 4

is a winning series of 4 in a row and, similarly, so is

1, 1, 1
2, 1, 2
3, 1, 3
4, 1, 4

and

1, 1, 1
2, 1, 1
3, 1, 1
4, 1, 1.

If the necessary powers of visualization seem to escape you, try running the program—it is much easier to see how it works by playing and getting a feel for it than by reading about it.

```
100   REM QUBIC
110   CALL CLEAR
120   Z9=0
130   PRINT TAB(5);"THIS IS THE GAME
      OF"::TAB(8);" -- QUBIC -- "::TAB(7);
      "3-D TIC-TAC-TOE":::::::::
140   INPUT "DO YOU WANT INSTRUCTIONS?▲▲▲(Y
      OR N): ":QUERY$
150   IF SEG$(QUERY$,1,1)="Y" THEN 2110
160   IF SEG$(QUERY$,1,1)="N" THEN 190
170   PRINT "PLEASE TYPE Y OR
      N,▲▲▲▲▲▲▲▲▲NOT ";QUERY$
180   GOTO 130
190   DIM X(64),L(76),M(76,4),Y(16)
200   DEF FNL(I)=X(M(I,1))+X(M(I,2))
      +X(M(I,3))+X(M(I,4))
```

```
210   DEF FNM(M)=M+110+6*INT((M-1)/
      4)+60*INT((M-1)/16)
220   FOR II=1 TO 16
230   READ Y(II)
240   NEXT II
250   FOR II=1 TO 76
260   FOR JJ=1 TO 4
270   READ M(II,JJ)
280   NEXT JJ
290   NEXT II
300   INPUT "DO YOU WANT TO MOVE FIRST▲▲▲(Y/
      N) ":QUERY$
310   CALL CLEAR
320   IF SEG$(QUERY$,1,1)="Y" THEN 360
330   IF SEG$(QUERY$,1,1)="N" THEN 450
340   PRINT "PLEASE TYPE Y OR
      N,▲▲▲▲▲▲▲▲▲NOT ";QUERY$
350   GOTO 300
360   GOSUB 1960
370   INPUT "ENTER YOUR MOVE ":K1,K2,K3
380   GOSUB 1920
390   IF (K1<1)+(K2<1)+(K3<1)+(K1>4)+(K2>4)
      +(K3>4)+(K1<>INT(K1))+(K2<>INT(K2))
      +(K3<>INT(K3))THEN 370
400   MM=16*K1+4*K2+K3-20
410   IF X(MM)=0 THEN 440
420   PRINT "THAT SQUARE IS
      TAKEN.▲▲▲▲▲▲▲PLEASE TRY AGAIN"
430   GOTO 370
440   X(MM)=1
450   GOSUB 1260
460   T=0
470   S = 0
480   FOR I=1 TO 76
490   IF L(I)<>4 THEN 560
500   PRINT "CONGRATULATIONS!"
510   PRINT FNM(M(I,1));FNM(M(I,2));
      FNM(M(I,3));FNM(M(I,4))
```

```
520   PRINT
530   PRINT "FINAL POSITION"
540   GOSUB 1960
550   GOTO 1160
560   IF L(I)<>15 THEN 580
570   S=I
580   IF L(I)<>3 THEN 600
590   T=I
600   NEXT I
610   IF S=0 THEN 700
620   I=S
630   FOR J=1 TO 4
640   MM=M(I,J)
650   IF X(MM)>0 THEN 690
660   X(MM)=5
670   PRINT "MACHINE MOVES TO ";FNM(MM)::
680   GOTO 520
690   NEXT J
700   IF T=0 THEN 1010
710   I=T
720   FOR J=1 TO 4
730   MM=M(I,J)
740   IF X(MM)>0 THEN 780
750   X(MM)=5
760   PRINT "NICE TRY -- I WILL MOVE
      TO ";FNM(MM)
770   GOTO 360
780   NEXT J
790   FOR I=1 TO 76
800   LL=FNL(I)
810   IF INT(LL)<>2 THEN 870
820   IF LL>2 THEN 1700
830   FOR J=1 TO 4
840   IF X(M(I,J))>0 THEN 860
850   X(M(I,J))=.125
860   NEXT J
870   NEXT I
880   GOSUB 1260
```

```
890  FOR I=1 TO 76
900  IF (L(I)=.5)+(L(I)=.375)THEN 1800
910  NEXT I
920  GOTO 1380
930  FOR Z=1 TO 16
940  IF X(Y(Z))=0 THEN 970
950  NEXT Z
960  GOTO 1300
970  MM=Y(Z)
980  X(MM)=5
990  PRINT "I WILL MOVE TO ";FNM(MM)::
1000 GOTO 360
1010 FOR I=1 TO 76
1020 LL=FNL(I)
1030 IF INT(LL)<>10 THEN 1090
1040 IF LL>10 THEN 1700
1050 FOR J=1 TO 4
1060 IF X(M(I,J))>0 THEN 1080
1070 X(M(I,J))=.125
1080 NEXT J
1090 NEXT I
1100 GOSUB 1260
1110 FOR I=1 TO 76
1120 IF (L(I)=.5)+(L(I)=5.375)THEN 1800
1130 NEXT I
1140 GOSUB 1920
1150 GOTO 790
1160 Z9=Z9+1
1170 IF Z9=2 THEN 2090
1180 INPUT "ANOTHER GAME? ":QUERY$
1190 IF SEG$(QUERY$,1,1)="Y" THEN 300
1200 IF SEG$(QUERY$,1,1)="N" THEN 1230
1210 PRINT "PLEASE TYPE Y OR
     N,▲▲▲▲▲▲▲▲NOT ";QUERY$
1220 GOTO 1180
1230 END
1240 REM
1250 REM
```

```
1260 FOR S=1 TO 76
1270 L(S)=FNL(S)
1280 NEXT S
1290 RETURN
1300 FOR MM=1 TO 64
1310 IF X(MM)>0 THEN 1350
1320 X(MM)=5
1330 PRINT "I LIKE";FNM(MM)
1340 GOTO 360
1350 NEXT MM
1360 PRINT "THE GAME IS A DRAW."
1370 GOTO 1160
1380 FOR K=1 TO 72 STEP 4
1390 P=INT(L(K))+INT(L(K+1))
     +INT(L(K+2))+INT(L(K+3))
1400 IF (P=4)+(P=9)THEN 1440
1410 NEXT K
1420 GOSUB 1920
1430 GOTO 930
1440 S=.125
1450 FOR I=K TO K+3
1460 GOTO 1810
1470 NEXT I
1480 S=0
1490 GOTO 1450
1500 DATA 1,49,52,4,13,61,64,16,22,39,23,
     38,26,42,27,43
1510 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,
     14,15,16,17,18,19,20
1520 DATA 21,22,23,24,25,26,27,28,29,30,
     31,32,33,34,35,36,37,38
1530 DATA 39,40,41,42,43,44,45,46,47,48,49,
     50,51,52,53,54,55,56
1540 DATA 57,58,59,60,61,62,63,64
1550 DATA 1,17,33,49,2,18,34,50,3,19,35,51,
     4,20,36,52
1560 DATA 5,21,37,53,6,22,38,54,7,23,39,55,
     8,24,40,56
```

```
1570  DATA 9,25,41,57,10,26,42,58,
      11,27,43,59,12,28,44,60
1580  DATA 13,29,45,61,14,30,46,62,15,31,47,
      63,16,32,48,64
1590  DATA 1,5,9,13,17,21,25,29,33,37,
      41,45,49,53,57,61
1600  DATA 2,6,10,14,18,22,26,30,34,38,42,
      46,50,54,58,62
1610  DATA 3,7,11,15,19,23,27,31,35,39,
      43,47,51,55,59,63
1620  DATA 4,8,12,16,20,24,28,32,36,40,44,
      48,52,56,60,64
1630  DATA 1,6,11,16,17,22,27,32,33,38,43,
      48,49,54,59,64
1640  DATA 13,10,7,4,29,26,23,20,45,42,39,
      36,61,58,55,52
1650  DATA 1,21,41,61,2,22,42,62,3,23,43,
      63,4,24,44,64
1660  DATA 49,37,25,13,50,38,26,14,51,39,
      27,15,52,40,28,16
1670  DATA 1,18,35,52,5,22,39,56,9,26,43,
      60,13,30,47,64
1680  DATA 49,34,19,4,53,38,23,8,57,42,
      27,12,61,46,31,16
1690  DATA 1,22,43,64,16,27,38,49,4,23,
      42,16,13,26,39,52
1700  FOR J=1 TO 4
1710  IF X(M(I,J))<>.125 THEN 1790
1720  X(M(I,J))=5
1730  IF LL<5 THEN 1760
1740  PRINT "LET'S SEE YOU GET OUT
      OF▲▲▲▲THIS! I MOVE TO";
1750  GOTO 1770
1760  PRINT "YOU FOX! JUST IN THE NICK▲▲▲OF
      TIME I MOVE TO";
1770  PRINT FNM(M(I,J))
1780  GOTO 360
1790  NEXT J
```

```
1800 S=.125
1810 IF I-INT(I/4)*4>1 THEN 1840
1820 A=1
1830 GOTO 1850
1840 A=2
1850 FOR J=A TO 5-A STEP 5-2*A
1860 IF X(M(I,J))=S THEN 1890
1870 NEXT J
1880 GOTO 1470
1890 X(M(I,J))=5
1900 PRINT "I TAKE";
1910 GOTO 1770
1920 FOR I=1 TO 64
1930 X(I)=INT(X(I))
1940 NEXT I
1950 RETURN
1960 A$="0.X"
1970 FOR I1=1 TO 13 STEP 4
1980 FOR J1=I1 TO I1+48 STEP 16
1990 FOR K1=J1 TO J1+3
2000 LL=ABS(X(K1)-2)
2010 PRINT SEG$(A$,LL,1);
2020 NEXT K1
2030 PRINT "▲";
2040 NEXT J1
2050 PRINT
2060 NEXT I1
2070 PRINT
2080 RETURN
2090 STOP
2100 END
2110 REM INSTRUCTIONS
2120 PRINT ::"QUBIC IS A THREE
     DIMENSIONALVERSION OF TIC-TAC-TOE. THE
     OBJECT OF THE GAME IS TO GET";
2130 PRINT "FOUR IN A ROW. COORDINATES":
     "ARE SPECIFIED AS LEVEL, ROW AND
     COLUMN WHERE EACH IS A▲▲NUMBER
     FROM 1";
```

```
2140 PRINT"TO 4. PICTURE":"THE LEVELS AS
     BEING STACKED ON TOP OF EACH OTHER
     WITH▲▲▲THE LEFTMOST AT THE BOTTOM."
2150 GOTO 190
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears the screen
120	sets Z9 to 0
130	introductory message
140–180	requests Y/N answer from user as to whether or not instructions are desired; if neither Y nor N is typed, asks again
190	sets aside storage for the arrays
200–210	defines the functions FNL and FNM
220–290	reads in data for program logic
300–350	asks the user whether or not to go first; accepts Y/N only
360	calls subroutine that prints the board
370–430	prompts user for the move, validates the numbers to ensure that they are within the range; computes position within the one-dimensional-board array X and checks to see that it is not taken. If it is, goes back and asks again
440	assuming all is well, sets the position in X to "occupied by human" (value 1)
450	calls the routine to sum all the winning paths
460–470	sets T and S to 0
480–600	checks sum of all paths. If equal to 4, prints winning message and asks for another game. If equal to 15, computer places the winning move and asks for another game. If equal to 3, computer moves to block player
610–690	computer's winning routine
700–780	computer's blocking routine
790–960	secondary loop to create two-way traps

970–1000	computer moves; position filled with 5
1010–1150	more two-way checking
1160–1170	error-handling routine
1180–1230	asks if another game is desired and accepts only Y/N
1240–1250	REMarks
1260–1290	routine to sum the paths
1300–1370	last-resort search for empty space to move; if none found, calls a draw and control transfers to line 1160
1380–1490	routine to assist in two-way computation
1500–1690	DATA statements containing the 76 possible winning paths
1700–1790	printout of two-way moves
1800–1950	routines for computing two ways
1960–2100	subroutine to print the board
1960	sets the variable A$ to the characters that make up the board
2110–2150	routine to display the instructions

PROGRAM DESCRIPTION

A detailed description of this program would simply be too long and complex to warrant its inclusion in the book. In its treatment of two-way traps particularly, it employs rather more sophisticated techniques than were found in the "Tic-Tac-Toe" program already described. There is, however, a particular feature which we have not yet encountered in any of the programs described so far.

The DEF FN statement enables a programmer to define a function that behaves much like a built-in function such as SIN or ABS. In line 200, for example, the function FNL is defined to sum up any specified path. That is to say, the statement

```
PRINT FNL(5)
```

prints out the sum of the fifth path. By going in a loop from 1 to 76, all seventy-six of the paths may be summed. In-

deed, this is done in several places throughout the program, which is why the function is used (to save the space that would be needed if it were typed each time).

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The program can be modified to play against itself. Who knows? It might lose that match for a change!
2. It would be very helpful to have a graphic illustration of the computer's attack scheme for teaching strategy. Toward this end the program may be modified to go into a "teach" mode and show what moves it is considering at any given time. Additionally, it could "suggest" what it considers to be the best possible move (this is actually more a modification of item 1 above, since the program would really be playing itself.)

PROGRAM

16

"FLIP-A-DISK"

PURPOSE

The game that we are about to describe is yet another mind-bending test of your strategic abilities. Originally called "Reversi," the game was very popular in the Victorian era in England and is sold today under the trademark "Othello," with some minor changes in the rules. The version represented by this program plays human versus computer and obeys most of the rules found in both of the above versions. The exception is the case in which all of one player's pieces have been eliminated. For simplicity, "Flip-a-Disk" simply terminates the game and announces that the player whose pieces remain on the board wins. A similar rule applies when both players must forfeit their moves. In that case the game is halted and the player with the most pieces wins.

For those who do not know the rules of "Reversi," they are as follows: The starting position is as shown on the screen before the first move. That is, there are two white and two black pieces in the middle of the board in a checkered pattern. The idea of the game is to flank the opponent's pieces—that is, to place a piece on a square such that some of the opponent's pieces are sandwiched in be-

tween the newly moved piece and one which was already on the board. All the flanked pieces are then "flipped" to the color of the flanking pieces. For example, if there is

O X X X

"O" moves to the location indicated

O X X X O

and flips the intervening pieces so that the board now becomes

O O O O O

"capturing" three of the opponent's pieces. The capturing may occur in any direction—horizontal, vertical, or diagonal. It may even occur in all eight directions in one turn—if you're extremely lucky. The rules may sound extremely simple—at first glance almost as easy as Tic-Tac-Toe. And only in two dimensions. Yet, nonetheless, this game is just about as difficult to master as chess. If you think that the game is not so tough, this program will—unless you are extraordinarily good—teach you some humility! If you feel the need to quit, this can be done by typing **Q** (for quit) as the column value. If you do, the computer will ask if you wish to play another game.

If you are smart—as we are sure you are—this game may take you as little as five minutes to learn. To master it though, you might have to spend the rest of your life at it. In any event, good luck.

```
100  REM FLIP-A-DISK
110  CALL SCREEN(13)
120  CALL CLEAR
130  RESTORE
140  T$="003C7E7E7E7E3C00"
150  CALL CHAR(128,T$)
160  CALL CHAR(136,T$)
170  CALL CHAR(137,"0000001818000000")
```

```
180  CALL COLOR(13,16,1)
190  PRINT "▲▲▲SETTING UP -- PLEASE▲▲▲▲▲▲▲
     ▲▲▲▲▲▲BE PATIENT":::::::::
200  DIM A(10,10),B(10,10),C(10,10),I4(8),
     J4(8),CASE(64)
210  FOR X=1 TO 10
220  FOR Y=1 TO 10
230  A(X,Y)=0
240  B(X,Y)=0
250  NEXT Y
260  NEXT X
270  FOR X=1 TO 5
280  FOR Y=1 TO 5
290  READ C(X,Y)
300  C(11-X,11-Y)=C(X,Y)
310  C(X,11-Y)=C(X,Y)
320  C(11-X,Y)=C(X,Y)
330  NEXT Y
340  NEXT X
350  DATA 0,0,0,0,0,0,9,6,7,5,0,6,-8,-4,
     -3,0,7,-4,4,0,0,5,-3,0,0
360  FOR X=1 TO 64
370  CASE(X)=1
380  NEXT X
390  FOR X=2 TO 7
400  CASE(X)=2
410  CASE(X+56)=2
420  CASE(8*X-7)=3
430  CASE(8*X)=3
440  NEXT X
450  BL=-1
460  C1=2
470  H1=2
480  N1=4
490  Z=0
500  WH=1
510  D$=CHR$(128)&CHR$(137)&CHR$(136)
520  FOR X=1 TO 8
```

```
530  READ I4(X),J4(X)
540  NEXT X
550  DATA 0,1,-1,1,-1,0,-1,-1,0,-1,1,
     -1,1,0,1,1
560  A(5,5)=WH
570  A(6,6)=WH
580  A(5,6)=BL
590  A(6,5)=BL
600  FOR I=4 TO 7
610  FOR J=4 TO 7
620  B(I,J)=1
630  NEXT J
640  NEXT I
650  B(5,5)=2
660  B(5,6)=2
670  B(6,5)=2
680  B(6,6)=2
690  INPUT "DO YOU WANT BLACK OR WHITE▲▲(B
     OR W): ":QUERY$
700  COMP=1
710  HUMAN=-1
720  IF QUERY$="W" THEN 770
730  IF QUERY$<>"B" THEN 690
740  COMP=-1
750  HUMAN=1
760  GOSUB 2100
770  INPUT "DO YOU WANT TO GO FIRST?▲▲▲▲(Y
     OR N) ":QUERY$
780  IF SEG$(QUERY$,1,1)="N" THEN 820
790  IF SEG$(QUERY$,1,1)<>"Y" THEN 770
800  PRINT :
810  GOTO 1270
820  B1=-10
830  I3=0
840  J3=0
850  T1=COMP
860  T2=HUMAN
870  I=2
```

```
880  FOR J=2 TO 9
890  IF B(I,J)<>1 THEN 1010
900  U=-1
910  GOSUB 1850
920  IF S1=0 THEN 1010
930  C9=C(I,J)+.1*S1
940  IF C9<B1 THEN 1010
950  IF C9>B1 THEN 980
960  R=INT(RND*10)
970  IF R<5 THEN 1010
980  B1=C9
990  I3=I
1000 J3=J
1010 NEXT J
1020 I=I+1
1030 IF I<10 THEN 880
1040 IF B1>-10 THEN 1090
1050 PRINT "I HAVE TO FORFEIT MY MOVE"
1060 IF Z=1 THEN 1710
1070 Z=1
1080 GOTO 1270
1090 Z=0
1100 PRINT "▲▲▲▲I MOVE TO [";STR$(13-
     1);",▲";CHR$(63+J3);"]"
1110 I=I3
1120 J=J3
1130 U=1
1140 B(I,J)=2
1150 GOSUB 1850
1160 GOSUB 2210
1170 C1=C1+S1+1
1180 H1=H1-S1
1190 N1=N1+1
1200 PRINT :"THAT GIVES ME";S1;"PIECE";
1210 IF S1<=1 THEN 1240
1220 PRINT "S":
1230 PRINT
1240 GOSUB 2100
```

```
1250  IF H1=0 THEN 1710
1260  IF N1=64 THEN 1710
1270  T1=HUMAN
1280  T2=COMP
1290  PRINT "PLEASE ENTER YOUR
      MOVE▲▲▲▲▲▲[ROW, COLUMN]: [";
1300  CALL KEY(3,KEY,STATUS)
1310  IF STATUS = 0 THEN 1300
1320  IF (KEY<48)+(KEY>56)THEN 1300
1330  PRINT CHR$(KEY);",▲";
1340  I=KEY-48
1350  CALL KEY(3,KEY,STATUS)
1360  IF STATUS=0 THEN 1350
1370  IF KEY=ASC("Q")THEN 1710
1380  IF (KEY<65)+(KEY>73)THEN 1350
1390  PRINT CHR$(KEY);"]"
1400  IF I<>0 THEN 1460
1410  INPUT "ARE YOU FORFEITING
      YOUR▲▲▲▲▲TURN? ":QUERY$
1420  IF SEG$(QUERY$,1,1)<>"Y" THEN 1290
1430  IF Z=1 THEN 1710
1440  Z=1
1450  GOTO 820
1460  J=KEY-63
1470  I=I+1
1480  IF A(I,J)=0 THEN 1510
1490  PRINT ::"SORRY, THAT SQUARE
      IS▲▲▲▲▲▲▲OCCUPIED, TRY AGAIN."::
1500  GOTO 1290
1510  U=-1
1520  GOSUB 1850
1530  IF S1>0 THEN 1560
1540  PRINT ::"SORRY, THAT DOES NOT FLANK
      AROW, TRY AGAIN"::
1550  GOTO 1290
1560  Z=0
1570  PRINT ::"THAT GIVES YOU";S1;"PIECE";
1580  IF S1 <= 1 THEN 1600
```

```
1590 PRINT "S":
1600 U=1
1610 GOSUB 2210
1620 GOSUB 2280
1630 B(I,J)=2
1640 GOSUB 1850
1650 H1=H1+S1+1
1660 C1=C1-S1
1670 N1=N1+1
1680 GOSUB 2100
1690 IF (C1=0)+(N1=64)THEN 1710
1700 GOTO 820
1710 PRINT "YOU HAVE";H1;"PIECES, AND":"I
     HAVE";C1;"PIECES"
1720 IF H1=C1 THEN 1780
1730 IF H1>C1 THEN 1800
1740 PRINT "SORRY OLD CHAP! I'M AFRAID▲▲I
     WON THAT ONE"
1750 IF C1-H1<20 THEN 1810
1760 PRINT "WHY DON'T YOU TAKE
     UP▲▲▲▲▲▲▲TENNIS OR SOMETHING?"
1770 GOTO 1810
1780 PRINT ::"IT'S A TIE, BY GOLLY! WELL,
     WE BOTH KNOW WE'RE THE BEST"
1790 GOTO 1810
1800 PRINT "YOU WON, YOU SON OF A GUN!"
     :"I DON'T WANT TO PLAY WITH▲▲▲YOU
     ANYMORE!"::
1810 INPUT "ALL RIGHT, WOULD YOU LIKE
     TOPLAY ANOTHER GAME?":QUERY$
1820 IF SEG$(QUERY$,1,1)="Y" THEN 120
1830 IF SEG$(QUERY$,1,1) <> "N" THEN 1810
1840 END
1850 S1=0
1860 FOR K=1 TO 8
1870 I5=I4(K)
1880 J5=J4(K)
1890 I6=I+I5
```

```
1900  J6=J+J5
1910  S3=0
1920  IF A(I6,J6)<>T2 THEN 2080
1930  S3=S3+1
1940  I6=I6+I5
1950  J6=J6+J5
1960  IF A(I6,J6)=T1 THEN 1990
1970  IF A(I6,J6)=0 THEN 2080
1980  GOTO 1930
1990  S1=S1+S3
2000  IF U<>1 THEN 2080
2010  I6=I
2020  J6=J
2030  FOR K1=0 TO S3
2040  A(I6,J6)=T1
2050  I6=I6+I5
2060  J6=J6+J5
2070  NEXT K1
2080  NEXT K
2090  RETURN
2100  REM PRINT THE BOARD
2110  PRINT ::"▲▲▲▲A▲B▲C▲D▲E▲F▲G▲H"
2120  FOR I=2 TO 9
2130  PRINT I-1;
2140  FOR J=2 TO 9
2150  PRINT "▲";SEG$(D$,A(I,J)+2,1);
2160  NEXT J
2170  PRINT
2180  NEXT I
2190  PRINT ::
2200  RETURN
2210  FOR K=1 TO 8
2220  I5=I4(K)
2230  J5=J4(K)
2240  IF B(I+I5,J+J5)>0 THEN 2260
2250  B(I+I5,J+J5)=1
2260  NEXT K
2270  RETURN
```

```
2280  SUBS=8*(I-2)+J-1
2290  ON CASE(SUBS)GOTO 2300,2470,2310
2300  RETURN
2310  IF I=3 THEN 2400
2320  IF (A(I-2,J)=HUMAN)+(A(I+1,J)=COMP)
      THEN 2370
2330  C(I-1,J)=-4
2340  IF I<>4 THEN 2380
2350  C(I-1,J)=-8
2360  GOTO 2380
2370  C(I-1,J)=8
2380  IF I<>8 THEN 2400
2390  RETURN
2400  IF (A(I+2,J)=HUMAN)+(A(I-1,J)=COMP)
      THEN 2450
2410  C(I+1,J)=-4
2420  IF I<>7 THEN 2440
2430  C(I+1,J)=-8
2440  RETURN
2450  C(I+1,J)=8
2460  RETURN
2470  IF J=3 THEN 2560
2480  IF (A(I,J-2)=HUMAN)+(A(I,J+1)=COMP)
      THEN 2530
2490  C(I,J-1)=-4
2500  IF J<>4 THEN 2540
2510  C(I,J-1)=-8
2520  GOTO 2540
2530  C(I,J-1)=8
2540  IF J<>8 THEN 2560
2550  RETURN
2560  IF (A(I,J+2)=HUMAN)+(A(I,J-1)=COMP)
      THEN 2610
2570  C(I,J+1)=-4
2580  IF I<>7 THEN 2600
2590  C(I,J+1)=-8
2600  RETURN
2610  C(I,J+1)=8
2620  RETURN
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–190	prints welcoming message with color—no sound, this is serious business
200	reserves room for the arrays
210–260	zeros matrices A and B
270–340	reads in values for ¼ of the matrix C and mirrors them to fill the other ¾
350	data for matrix C
360–380	sets all elements of array CASE to 1
390–440	sets the special cases in array CASE
450–510	initializes variables
520–540	reads in values for arrays I4 and J4
550	data for arrays I4 and J4
560–590	sets the initial position
600–680	sets up initial weighting for the board, taking the initial four pieces into account
690–750	gives player the choice of color and sets COMP and HUMAN accordingly
760	calls routine at 2100 to print the board
770–810	gives player the choice of moving first or last; if the player wishes to go first, control passes to line 1270; otherwise the program goes on to line 820
820–870	initializes variables for calculating the computer's move
880–1030	searches for legal moves
1040	if one was found, control transfers to line 1090, where a move is made
1050–1080	otherwise, program indicates that it forfeits; sets a flag and lets the player take another turn; if player has already forfeited the last move, control passes instead to line 1710, where the game is ended
1090	if a move is made, turns off the forfeit flag
1100	prints the computer's move
1110–1150	flips the pieces

1160	calls routine in 2210 to change board status
1170	adds the number of pieces captured plus the one put down to computer's score
1180	subtracts the number of pieces captured from human's score
1190	adds one to number of turns (number of pieces on the board)
1200–1230	prints the number of pieces collected from opponent
1240	calls the routine that prints the board
1250–1260	if the human player loses all pieces or if the board is filled, goes to line 1710 to end the game
1270–1280	sets variables for human's move
1290–1390	inputs player's moves and displays it; if Q is typed as a column, control passes to 1710 to end the game
1400–1450	if the player entered a zero as the row number (indicating forfeit) computer double checks by asking if the player is forfeiting; if the answer is yes, the forfeit flag is set and the computer's turn is begun; if the computer has already forfeited last turn, control passes to 1710 to end the game
1460–1470	I and J are set to the correct board position
1480–1500	if position on the board is already taken, prints a suitable message and goes back to ask again
1510–1550	tests to see if the move flanks a row; if not, goes back and asks again
1560	turns off the forfeit flag
1570–1590	prints the number of pieces given up by computer
1600–1640	calls routine to change board status, changes the computer's strategy matrix, and flips the disks
1650–1670	adds new disks to human score; subtracts captured disks from computer's score; and increments the move counter

1680	calls routine to display the board
1690–1700	if computer has lost all its pieces or the board is filled, control drops to line 1710 to end the game; otherwise, control is sent back to 820 for computer's response
1710–1800	game is finished, with program printing out number of pieces of each player and a message which varies according to the way the game turned out
1810–1840	asks if player wishes to play again; if so, control is sent back to line 120; otherwise, program ends
1850–2090	checks whether move indicated by I and J is a flanking move and if so flips the pieces if U is set to 1
2100–2200	displays the board on the screen
2210–2270	routine to modify the allowable-moves matrix
2280–2620	routine to modify the move-weighting matrix, based on last move made

PROGRAM DESCRIPTION

As can be seen from the length and complexity of the program, "Flip-a-Disk" is no lightweight. To describe the workings of the program in detail, as with "Qubic," would require extensive discussion of strategy. To discuss the strategies behind "Othello" alone could fill a book by itself and would be inappropriate in a book such as this. Therefore, we omit the detailed description of this program, although we would like to point out that the line-by-line analysis given above is quite detailed in its own right.

There are a few points within the program that warrant discussion, however. The RESTORE statement in line 130 is the first point that should be explained. The statement has the effect of returning the data-reading cursor back to the beginning of the DATA statements, so that the DATA elements may be read again. The second technique is that used in lines 270–340. In these lines, the array C is set

based on only one quarter of the data that it would normally require. This is made possible by reading in a quarter of the matrix and, since it is symmetrical, copying the one corner into the three others.

When this program is run you will notice that it takes considerable time before anything happens. The reason for this is that BASIC needs time to prepare the program for execution—the longer the program the more time it will take. Moreover, in view of the fact that so much data is involved in the program, a lot of computer time is spent in reading it in.

The last item to be discussed is the manner in which all moves are entered. As with "Qubic," a move is made on a definite coordinate system. First, a digit from 1 to 8, specifying the row, is typed in, followed by a letter from A to H, indicating the column. To enhance the image on the screen, each coordinate is automatically enclosed in square brackets and a separating comma is also automatically inserted. Of course, pressing the **ENTER** key is unnecessary because we took advantage of the CALL KEY statement once again. Each time a move is made, the board is replenished, showing all disks in their new positions. In addition, a message is displayed informing the player of the number of pieces taken by that move. At the end of the game, the final count of pieces on both sides is displayed.

The CALL CHAR statements followed by the CALL COLOR statement in lines 150–180 allow the computer to draw the black-and-white disks instead of X and O. Line 510, which sets D$ to these characters, is used in place of the statement

```
D$="X.O"
```

for the same reason.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. The game can, as with all the other strategy games, be turned into a two- or no-player game. That is to say, the

computer can either let two humans battle it out and merely keep track, or it can play itself.

2. The program can be modified to display the board in the same place on the screen each time using HCHAR and only drawing what must be changed. This would look better, but it is also more tedious to program because every PRINT statement must be converted into a series of statements. (For the best solution to this problem, see the "Blackjack" or "Concentration" programs.)

PROGRAM

17

"MAGIC SQUARES"

PURPOSE

In case you are unfamiliar with the concept of a "magic square," it is nothing more than an N × N array of all the integers from 1 to N times N (without repetitions), such that each row, column, and diagonal adds up to the same number, which happens to be equal to (N * N * N + N) / 2. For any value of N, there are various transpositions of the magic square which also yield this result. Magic squares have been a source of delight for many hundreds of years, but it was not until recent years that attempts were made to generate magic squares using microcomputers.

Algorithms have been developed for magic squares when N is odd, and different algorithms for when N is even. The program we shall present works only if N is odd. Although we have restricted the maximum value of N to 9, the algorithm works for values larger than 9. However, for values greater than 9, the output does not fit neatly on the screen.

```
100 REM MAGIC SQUARES
110 CALL CLEAR
120 PRINT "THIS PROGRAM GENERATES
```

```
     MAGICSQUARES (N×N WHERE N IS
     ODD)"::::::::
130  DIM M(9,9)
140  INPUT "ENTER AN ODD VALUE FOR N:":N
150  IF (N/2=INT(N/2))+(N<3)+(N>9)THEN 140
160  X = 1
170  Y = INT(N/2)+1
180  FOR C = 1 TO N*N
190  M(X,Y)=C
200  X0=X
210  Y0=Y
220  X = X-1
230  IF X > 0 THEN 250
240  X = N
250  Y = Y + 1
260  IF Y <= N THEN 280
270  Y = 1
280  IF M(X,Y) = 0 THEN 310
290  X = X0 + 1
300  Y = Y0
310  NEXT C
320  CALL CLEAR
330  FOR I = 1 TO N
340  FOR J = 1 TO N
350  PRINT TAB(J*4-3); M(I,J);
360  NEXT J
370  PRINT ::
380  NEXT I
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears screen
120	displays heading
130	sets aside room for matrix M
140–150	prompts player to input value of N, which is then tested for validity

160–170 assigns initial row to X, initial column to Y
180–320 generates the magic square
190 puts the next consecutive integer into the element of M indexed by X and Y
200–210 saves the old values of X and Y in XO and YO
220 moves up one row
230–240 if top of matrix is reached, continues from the bottom row (resets X to N)
250 moves right one column
260–270 if the right side of the matrix is reached, continues from the left side (resets Y to 1)
280 if the element of the matrix indexed by X and Y is vacant, control branches to line 310
290–300 if the element has already been assigned a number, reassigns values to X and Y so as to point to the element just below the last element filled (that element will always be empty, except when the matrix is filled)
320 clears screen
330–380 displays the completed magic square

PROGRAM DESCRIPTION

After the screen has been cleared and a suitable title is displayed, the user is asked to enter an odd value for N. Whatever value is entered is tested in line 150 to be sure it is an integer not less than 3 nor greater than 9. It is also tested for "oddness" by dividing it by 2 and checking whether the result is a whole number. If any of these tests fail, control is simply returned to line 140, where the player is given another opportunity to input a suitable number.

It is at line 160 that the algorithm for generating the magic square commences. It is based on a model in which the bottom of the matrix is considered to be joined to the top, and the left side joined to the right side, in a sort of spherical fashion. For example, if we were taking a "tour" of a 3 × 3 matrix, visiting the first column, then the second column, and then the third, our next step to the right would

bring us back to the first column. If we were on the bottom row, a step down would bring us back to the top. (My, wouldn't Columbus be proud?) This matrix is filled by starting in the middle column of the first (top) row of the matrix and moving diagonally up and to the right, filling the elements with consecutive numbers starting with 1 and ending with the value of N * N. When an element is encountered which has previously been "visited" and assigned a value, the computer goes instead to the space directly below the last-filled space and fills that space instead.

Lines 160 and 170 set X and Y to point to the starting row and column, respectively. The formula in line 170 is used to find the middle column. The program then enters a FOR . . . NEXT loop (extending from line 180 through line 310) that generates the consecutive numbers 1 through N * N. Each time through the loop an element of the matrix is assigned a value. (The completion of the FOR . . . NEXT loop indicates that the entire matrix is filled.) The first thing done within the loop is to place the next consecutive number in the matrix location specified by the values of X and Y. Then it is necessary to find the next location to be filled next time around the loop. The old values of X and Y are saved in XO and YO, in case the position diagonally up and to the right is already occupied and the program will have to use the alternate position (directly below the old position).

The value of X is then decremented by 1 in line 220 to facilitate moving up a row. If this moves X past the top row (X becomes 0), X is set to N so it now points to the bottom row. Y is incremented to point to the next column to the right, and if it goes past the Nth column, Y is set to 1 to indicate the first (leftmost) column. If the new element now indexed by X and Y is empty, control passes to the NEXT C statement in line 310 which continues the loop. Otherwise, X and Y are reset (using the values of XO and YO) to point to the position right below the last element filled, and the loop continues on its merry way.

Finally, once all the consecutive numbers have been

stored in their appropriate locations to yield the magic square, the screen is cleared and the magic square is printed out for the benefit of your visual and intellectual enjoyment. The reason for using the seemingly complex argument for the TAB function in line 350 is that both one-digit and two-digit numbers are being displayed, and the TAB makes sure that they are all aligned. Since only one number is printed per PRINT command, rather than a line, it is necessary to calculate the column number to which the computer must TAB before printing each value, based upon the value's position in the magic square.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. A resourceful programmer may be tempted to search out algorithms that produce magic squares when N is an even number.
2. Even magic squares with dimensions larger than 9 may be generated and displayed neatly. However, more complicated graphic techniques would have to be used. Of course, lines 130 and 150 would have to be amended accordingly.
3. The program may easily be amended to allow the user to request more than one magic square per RUN.

PROGRAM

18

"CALENDAR"

PURPOSE

This program, in effect, provides you with a perpetual calendar. If you are like the majority of people, you probably do not know the day of the week on which you were born. If you're at all curious, this program is made for you. What the program does is to permit a user to type in any year of his choosing (after 1582, when the current calendar was stabilized). Then the computer asks for any month within that year. As soon as the month is typed in and the **ENTER** key pressed, a calendar appears miraculously before your very eyes for the entire month specified, from which you can easily see on which day of the week any of the dates fall. The year and month of your choice do not have to be in the past, either. They may just as well be in the present, or any time in the future.

The program takes into account leap years as well. For example, the year 1984 is a leap year, because when divided by 4 it leaves a remainder of 0. But this is not the whole story. Century years (years whose last two digits are 00) have to be treated specially. If the year in question is a century year, it must be evenly divisible by 400 in order to be a leap year. That is to say that the years 1700, 1800, and

1900, although evenly divisible by 4, are not leap years, since they are not evenly divisible by 400. The year 2000, however, being divisible by 400 without leaving a remainder, is a leap year. As you will recall, any year which is a leap year has one day added to the month of February, giving 29 days rather than 28.

```
100 REM CALENDAR
110 CALL CLEAR
120 DIM DPM(12)
130 FOR I=1 TO 12
140 READ DPM(I)
150 NEXT I
160 DATA 31,28,31,30,31,30,31,31,30,31,
    30,31
170 PRINT "THIS PROGRAM GENERATES
    A▲▲▲▲CALENDAR OF ANY MONTH OF▲▲▲▲YOUR
    CHOICE.":::::::::
180 INPUT "WHAT YEAR (AFTER 1582): ":YEAR
190 IF (YEAR<1582)+(YEAR<>INT(YEAR))THEN
    180
200 INPUT "WHAT MONTH (1-12): ":MONTH
210 IF (MONTH<1)+(MONTH>12)+
    (MONTH<>INT(MONTH))THEN 200
220 DPM(2)=29
230 IF (YEAR/400=INT(YEAR/400))+(YEAR/4
    =INT(YEAR/4))*(YEAR/100<>INT(YEAR/100))
    THEN 250
240 DPM(2) = 28
250 JDAY = 0
260 FOR I=1 TO MONTH-1
270 JDAY=JDAY+DPM(I)
280 NEXT I
290 PRINT :::"SU▲M▲▲▲TU▲▲W▲▲▲TH▲▲F▲▲▲SA":
    "----------------------------"::
300 FOR JULDAY=JDAY TO JDAY+DPM(MONTH)-1
310 TEMP=YEAR+JULDAY+INT((YEAR-1)/400)
    -INT((YEAR-1)/100)+INT((YEAR-1)/4)
320 POSITION=TEMP-INT(TEMP/7)*7
```

```
330  WEEKDAY = POSITION-7*(WEEKDAY=0)
340  PRINT TAB(POSITION*4);STR$(JULDAY-
     JDAY+1);
350  IF WEEKDAY <> 7 THEN 370
360  PRINT
370  NEXT JULDAY
380  PRINT :::
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	clears screen
120	reserves space for the 12 months of the year
130–150	reads in the number of days in the months January through December, giving February 29 in case it is a leap year
160	DATA for array DPM (days per month)
170	displays title
180–190	requests that user type in year, which is then validated (must be after 1582 and an integer). If the year is invalid, control is sent back to line 180, where another opportunity is given
200–210	the number of the month is requested and checked for validity; once again, control returns to the INPUT statement if the month is invalid
220	checks to see whether year is a leap year. If it is, control skips to line 240
230	if not a leap year, the number of days in February is changed to 28
240	the Julian day (to be explained below) is set to 1
250–270	the number of days in the year up to the month specified in MONTH is calculated and put in JDAY
280	prints heading for calendar
290–350	main loop of program; each time through the loop, the corresponding day of the month is calculated and printed

300–310 the day of the week on which the current date falls is calculated, with Sunday = 1, Monday = 2, etc., up to Friday = 6, Saturday = 0 (for mathematical reasons)

340 the day of the month is printed in its correct position

350–360 if the day just printed did not fall on a Saturday, control branches to line 370 to continue the loop; otherwise a null PRINT command is used to advance to the next line on the screen, where a new week is begun

370 goes on to the next day of the month

380 prints a few blank lines for aesthetic purposes

PROGRAM DESCRIPTION

After the screen is cleared, the number of days in the months January through December are read into the array DPM. A program title is then displayed and the user is asked to type in the year of interest. The year must be later than 1582, because it was in that year that the current Gregorian calendar was put into effect. Therefore, the year is tested to be sure it is not 1582 or earlier. In addition, the year is checked to be sure it is an integer (whole number). All being well, the user is then asked to type in the month of his choice, as a number from 1 to 12. This number is also validated, and if it is found wanting, control is returned to line 200, where another opportunity is given.

It is in line 220 that the year is tested to determine whether it is a leap year or not. This is done through the divisibility tests described in the "Purpose" section above. If the year is a leap year, the program skips over line 230, which sets the number of days in February to 28. As you may recall, the number of days in February was initially read in as 29, which is correct for a leap year only.

The formula upon which this program rests depends upon what is known as the "Julian day." If you imagine the days of the year to be successively numbered from 1 to 365 (for a non-leap year) or 1 through 366 (for a leap year),

the corresponding number for a particular day is its Julian day. Therefore, January 1 of any year is Julian day 1, while December 31 is Julian day 365 in the case of a non-leap year, or 366 if it is a leap year. The variable JDAY is set to the Julian day corresponding to the first day of the month selected by the user. JDAY is initialized to 1, and then the number of days in each month prior to the selected month is added to JDAY.

The next thing the program does is to print a heading indicating the days of the week "SU" through "SA", which is underlined. Some blank lines are then skipped for aesthetic purposes. Within the FOR . . . NEXT loop beginning in line 300 and extending to line 370, the index JULDAY always has a value which is the Julian day equivalent of the day currently being printed. (Do not confuse this with JDAY, which remains set to the first day of the month.) The value in JULDAY is used to calculate the day of the week on which the current day falls, in the complicated-looking formula in lines 310-320. This adds together the value of the year, the Julian day, and various other quantities which clearly relate to the calculation of leap years. This sum is then in effect divided by 7, with the remainder, an integer 0 through 6, being saved in the variable POSITION (line 320). This remainder corresponds to the day of the week. The current day of the month is then printed on the current line, with the value in the variable POSITION used to determine in which column to print the number. Line 350 tests to see whether the day just printed is a Saturday (if POSITION has the value 0). If it is, a PRINT statement is executed. This offsets the effect of the semicolon at the end of the PRINT statement in line 340, permitting the next day to be printed on a new line.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. It may be desirable to have the computer ask the user if another calendar is required after one has been displayed.
2. Once the techniques of this program have been mas-

tered, it will then become possible to amend it so that, given any year, the entire year may be displayed month by month.

3. You might want to type in any given date and have the computer tell you simply on which day of the week it falls.

PROGRAM

19

"PHONE TRANSLATOR"

PURPOSE

Throughout the United States and Great Britain (and possibly other countries too) there is a tendency to assign new telephone numbers using only digits rather than a combination of letters and digits. The purpose of this change is to make all telephone numbers uniform.

Contrary to what the public relations personnel of the various telephone companies might say, telephone numbers are not usually that simple to remember. However, if we take advantage of the fact that, on the typical American phone dial, there are three letters of the alphabet associated with each of the digits 2–9, a mnemonic can often be constructed out of these digits.

For example, it is much simpler to dial "BARBARA" than it is to remember its equivalent number, 227-2272. With this in mind, the following program has been written which enables a person to type in any seven-character telephone number, be it composed only of digits or of a combination of digits and letters. Since the digits 0 and 1 on the dial do not have alphabetic letters associated with them, they are left intact if they are part of a telephone number. The program requires only that the user input seven char-

acters, be they letters or numbers; however, no hyphens are permitted. If hyphens (or for that matter, any other character not found on the phone dial) are typed, they are simply ignored. (This includes the letters Q and Z, both of which are not found on the standard American dial.)

The output to this program is quite voluminous, since there are three-raised-to-the-seventh-power combinations that can be constructed from a seven-digit telephone number. In order to print these 2,187 combinations, it is arranged so that they print in two columns—giving the viewer ample opportunity to scan the list as it is displayed to search for a suitable mnemonic.

```
100 REM PHONE TRANSLATOR
110 CALL CLEAR
111 PRINT "RING ... RING ... RING":::::::::
112 PRINT "TYPE IN THE TELEPHONE #▲▲▲▲▲(NO
    DASHES PLEASE) ";
120 CHAR$="000111ABCDEFGHIJKLMNOPRSTUVWXY"
130 FOR I = 1 TO 7
140 CALL KEY(3,KEY,STATUS)
150 IF STATUS<1 THEN 140
160 IF (KEY<48)+(KEY>57)*(KEY<65)+(KEY>89)
    THEN 140
170 IF KEY=ASC("Q")THEN 140
180 PRINT CHR$(KEY);
190 IF KEY<58 THEN 210
200 KEY=INT((POS(CHAR$,CHR$(KEY),1)-1)/3)+49
210 NUMB(I)=KEY-48)*3
220 NEXT I
230 CALL CLEAR
240 FOR A=1 TO 3
250 FOR B=1 TO 3
260 FOR C=1 TO 3
270 FOR D=1 TO 3
280 FOR E=1 TO 3
290 FOR F=1 TO 3
300 FOR G=1 TO 3
```

```
310  PRINT SEG$(CHAR$,NUMB(1)+A,1);
320  PRINT SEG$(CHAR$,NUMB(2)+B,1);
330  PRINT SEG$(CHAR$,NUMB(3)+C,1);
340  PRINT SEG$(CHAR$,NUMB(4)+D,1);
350  PRINT SEG$(CHAR$,NUMB(5)+E,1);
360  PRINT SEG$(CHAR$,NUMB(6)+F,1);
370  PRINT SEG$(CHAR$,NUMB(7)+G,1),
380  NEXT G
390  NEXT F
400  NEXT E
410  NEXT D
420  NEXT C
430  NEXT B
440  NEXT A
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–112	clears screen and announces title of program
120	sets the variable CHAR$ equal to the character equivalents of the numbers on the dial
130–220	enters the phone number, accepting only legal characters (all others are ignored)
230	clears the screen of the title and prepares for the output
240–440	a nest of loops in which all the permutations of the phone number are printed
310–370	prints the character equivalents of positions 1–7 of the phone number

PROGRAM DESCRIPTION

After the screen has been cleared and the welcoming message displayed, the string variable CHAR$ is defined. It consists of all the characters permitted in the American dialing system. Each group of three characters in sequence

is associated with each digit on the dial. For example, since zero has no equivalent on the dial, the first three digits are 000. The digit 2 does, however, so its equivalent is ABC.

Within the FOR . . . NEXT loop extending from line 130 to 220 the keyboard is continuously scanned for the entry of the seven characters of the telephone number to be converted. Lines 140–150 wait until a character (any character at all) has been typed. Line 160 then screens out all characters outside the range of the digits 0–9 and the letters A–Y. Then in line 170, the one anomaly—the letter Q is tested for. If it has been typed, control is sent back to line 140 to read another character, because it, too, is not found on the phone dial.

Once the character has been validated, it is displayed on the screen. If the character is a number, the test in line 190 skips the statement in 200, which converts a letter to its equivalent digit. This is accomplished by taking advantage of the POS function, which, if you will remember, searches for the occurrence of one string within another and returns the position found. Here, we search for the occurrence of the character typed within the list of all the alphabetic characters possible. Since we have screened out all illegal values, there is no possibility of a value of 0 being returned, since the character will always be found at some position. One is then subtracted from the result and it is divided by 3. When the integer portion of this quantity is taken, we are left with a number from 2 to 9, which is the equivalent of the letter typed in. Then, the number is converted so that it points to the first of its representations within the string CHAR$.

Within the nest of seven loops, each of the three equivalent letters of the alphabet corresponding to each of the digits is printed out in the various combinations that exist. For example, assuming a telephone number begins with the digit 2, NUMB(1) is set to 7—the starting position of the group of three letters that are equivalent to 2. Since the index A ranges from 1 to 3, the three letters are accessed in turn.

Notice that each of the lines 310–360 terminates with a semicolon, while line 370—which looks the same in most respects—terminates with a comma. The net effect of this arrangement is to print the combinations in two columns for ease of reading. The letters are printed side by side because of the semicolons, and the final comma produces the two-column format.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. Lines 200 and 210 have been written with an eye to allow for the possibility of moving line 180 to 205. This modifies the program (should it be desired) to print out the digit equivalent of what was typed—regardless of what character it was. In other words, the program, when modified as suggested, will display the number "2" even when an A is typed.
2. With some careful thinking and planning, it is possible to amend the program so that it allows for the inclusion of a telephone area code, which may, or may not (at the programmer's discretion), be included in the final mnemonic.
3. If the display proves to move too fast for the viewer, a time delay (an empty FOR . . . NEXT loop, for example) can easily be inserted into the program to allow for an interval of time to pass between each displayed line.

PROGRAM

20

"MORSE CODE"

PURPOSE

As you may know, in order to qualify for a ham radio license it is necessary to know Morse code. Depending on the level of the license, various degrees of expertise in the code are expected. With this program it is possible to sharpen your skills at learning Morse. All you have to do is to type in your English message and it is displayed on the screen and in sound as Morse.

```
100  REM MORSE CODE
110  CALL CLEAR
120  PRINT "THIS IS A MORSE
     CODE▲▲▲▲▲▲▲▲TRANSLATOR.":::::::::
130  CALL CHAR(46,"0000001818000000")
140  TL = 50
150  TONE = 2000
160  DIM CODE$(26)
170  FOR I=1 TO 26
180  READ CODE$(I)
190  NEXT I
200  INPUT "ENTER YOUR MESSAGE
     IN▲▲▲▲▲▲▲ENGLISH:":MESSAGE$
```

```
210 PRINT ::
220 FOR I=1 TO LEN(MESSAGE$)
230 X=ASC(SEG$(MESSAGE$,I,1))-64
240 LETTER_CODE$="▲▲▲"
250 IF (X(1)+(X>26)THEN 270
260 LETTER_CODE$=CODE$(X)
270 FOR J=1 TO LEN(LETTER_CODE$)
280 TEMP$=SEG$(LETTER_CODE$,J,1)
290 PRINT TEMP$;
300 IF TEMP$-"▲" THEN 390
310 IF TEMP$="." THEN 420 ELSE 440
320 FOR T=1 TO TL*.5
330 NEXT T
340 NEXT J
350 PRINT "▲";
360 NEXT I
370 PRINT ::
380 END
390 FOR T=1 TO 25
400 NEXT T
410 GOTO 320
420 CALL SOUND(TL,TONE,0)
430 GOTO 320
440 CALL SOUND(TL*3,TONE,0)
450 GOTO 320
460 DATA ".-","-...","-.-.","-..","."
    "..-.","--.","....","..","---","-.-"
    ".-..","--","-.","---",".--.","--.-"
470 DATA ".-.","...","-","..-","...-"
    ".--","-..-","-.--","--.."
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–120	clears the screen and displays the welcoming message

130	sets the "." character to be a dot in the center of the line instead of the bottom
140	sets the tone length to 50 ms
150	sets the frequency to 2000 Hz
160	sets aside room for the 26 letters of the alphabet
170–190	reads the code for each of the letters
200	enters the user's message
210	skips two lines
220–360	main loop, in which the code is generated
230	picks out each character in the message and places a corresponding number from 1 to 26 in the variable X
240	sets the letter code to spaces; if letter code is not subsequently set it defaults to spaces to produce breaks between words
250	if X is outside the valid range, LETTER_CODE$ remains at its default value
260	otherwise, LETTER_CODE$ is set to the code of the letter that was typed
270–340	loop to print the contents of letter code and sound the appropriate tones
280	pulls one character out of LETTER_CODE$ and places it in TEMP$
290	prints the dash or dot
300	if the character is a space, prints it and waits for a specified interval
310	transfers control to the routines to sound dot or dash
320–330	delay loop separates every dash and dot
350	separates the dashes and dots by a space
370–380	prints a space and terminates the program
390–410	routine to produce a short delay and go on with the program
420–430	routine to sound out a dot
440–450	routine to sound out a dash
460–470	data containing the letters of the alphabet represented in Morse code

PROGRAM DESCRIPTION

The program begins with the usual starting lines, which extend from 100 to 120. Line 130 redefines the period character (number 46) to be a centered dot so that it is in line with the dash (represented by the minus sign). Then, lines 140 and 150 set the time for each dot in milliseconds and the tone frequency, which is to be played each time a dot or dash is sounded. Lines 160–190 reserve room for the Morse codes corresponding to the twenty-six letters of the alphabet and read them into CODE$.

The user is prompted to enter a message in line 200, and this is stored in MESSAGE$. For each letter of the message a corresponding number between 1 and 26 is stored in the variable X. The variable LETTER_CODE$ is then set to spaces. If this is left unchanged, meaning that a valid character was not read in, it is interpreted as a break in between two words and thus creates a delay. If, however, a character is found, the value is set to the appropriate combination of dots and dashes and a much smaller delay is generated in lines 320–330.

Lines 370–380 print two blank lines and terminate the program once the entire message has been displayed in Morse code. Following them are the various routines used in the program. The first resides in lines 390–410. It produces a delay after a word which is larger than the one in 320–330. A second routine is located in lines 420–430, which sounds out a dot tone, followed by lines 440–450, which sound out a dash. The end of the program consists of DATA statements that contain the Morse code equivalents of the alphabet.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. It is possible to turn this Morse code program into a quiz by storing phrases in data and randomly pulling them

out. The player would then be responsible for typing in the English equivalent of the emitted Morse code.

2. A more sophisticated idea is to create a timed quiz in which the Morse code is displayed at a certain rate and must be echoed back at the same rate. The original is then compared with what is typed in by the user and the results scored according to the time it took and the accuracy.
3. Two easy modifications are to change the length of time that a dot and dash sound for and to change the frequency at which they are played. The first is accomplished by changing the value of TL in line 140, while the second requires a similar change in line 150. The relative spacing between letters is changed by modifying the end value of the delay loop in line 320.
4. If you own a joystick, it may be possible to go the other way—that is, to devise a quiz that expects you to type in the Morse code equivalent. This cannot be done easily on a keyboard, however, due to the timing difficulties intrinsic to it.

PROGRAM 21 "LANDER"

PURPOSE

This game simulates the landing of an Earth–Moon shuttle. The automatic controls have broken down, forcing you, the commander, to take charge. Each second, you must specify the amount of fuel to burn. If you don't burn any, you accelerate. If you burn more fuel than a certain amount (different in every gravity well) you accelerate in the opposite direction. The idea behind the game is to gauge how much fuel to burn to effect as smooth a landing as possible. This is measured by your velocity as you land. A perfect landing is one in which velocity is less than one foot per second. At five feet per second or less, the landing is counted as rough but survivable. A landing of greater than five feet per second is considered fatal. If you do crash, the screen flashes violently and emits the sound of an explosion.

The danger to be wary of is of burning too much fuel. This is very real, since you have not been given much to start with. If you run out of fuel, the game goes on, but you are no longer consulted about how much fuel to burn since you have none. A falling sound is generated and the status

of the ship continues to print out until the inevitable crash, which ends the game.

It is recommended that you try landing on the moon first, since it is easier to land there because the gravity is weaker than Earth's. Happy landings!

```
100   REM LANDER
110   CALL CLEAR
120   PRINT "WELCOME TO THE GAME
      OF ...":::::::::
130   PRINT "MOON L":::::::
140   FOR I=17 TO 23
150   CALL HCHAR(I,8,32)
160   CALL HCHAR(I+1,8,76)
170   NEXT I
180   PRINT TAB(6);"LANDER":::::
190   INPUT "PLEASE ENTER YOUR NAME: ":NAME$
200   INPUT "INSTRUCTIONS? (Y/N) ":QUERY$
210   IF SEG$(QUERY$,1,1)="Y" THEN 1670
220   PRINT
230   PRINT "GOOD LUCK AND
      HAPPY▲▲▲▲▲▲▲▲▲LANDINGS"
240   PRINT
250   X0=0
260   V0=0
270   INPUT "LOCATION: MOON OR EARTH?":LOC$
280   LOC$=SEG$(LOC$,1,1)
290   K=0
300   IF LOC$="M" THEN 350
310   K=1
320   IF LOC$="E" THEN 350
330   PRINT "SORRY, NO SUCH LOCATION"
340   GOTO 270
350   G=5+27*K
360   M=30+60*K
370   IF X0>0 THEN 400
380   X0=500+1500*K
390   X=X0
```

```
400   V0=-50-100*K
410   V=V0
420   X=500+1500*K
430   V=-50-100*K
440   X0=X
450   V0=V
460   F=INT(SQR(M*(V*V+G*G*X)/(M-
      G))*.13+.5)*10
470   PRINT
480   PRINT "INIT HEIGHT:▲▲";X;"FEET"
490   PRINT "INIT VELOCITY:";V;"FT/SEC"
500   PRINT "FUEL SUPPLY:▲▲";F;"UNITS"
510   PRINT "MAXIMUM BURN:▲";M;"UN/SEC"
520   PRINT
530   PRINT "THE AMOUNT OF BURN TO
      CANCELGRAVITY IS";G:"UNITS PER SECOND"
540   PRINT
550   PRINT "HIT ANY KEY WHEN
      READY,▲▲▲▲▲COMMANDER ";NAME$
560   CALL KEY(3,AA,XX)
570   IF XX=0 THEN 560
580   PRINT
590   FOR T=1 TO 32000
600   GOSUB 1520
610   INPUT "THRUST? ":B
620   B1=ABS(B)
630   IF B1<=M THEN 660
640   PRINT "YOUR ENGINE CANNOT
      SUSTAIN▲▲THIS THRUST"
650   GOTO 600
660   T8=2
670   T9=T8
680   IF B1=0 THEN 700
690   T9=F/B1
700   A=B-G
710   R=V*V-2*A*X
720   IF R<0 THEN 780
730   IF A=0 THEN 760
```

```
740  T8=-(V+SQR(R))/A
750  GOTO 780
760  IF V>=0 THEN 780
770  T8=-X/V
780  IF (T8>0)*(T8<=1)+(T9<=1)THEN 870
790  X=X+V+A/2
800  V=V+A
810  F=F-B1
820  IF X<=.0001 THEN 850
830  NEXT T
840  STOP
850  T=T+1
860  GOTO 1120
870  IF (T8>0)*(T8<=1)THEN 1090
880  PRINT T+T9;"OUT OF FUEL"
890  SND=5000
900  B1=0
910  F=B1
920  X=X+V*T9+A*T9*T9/2
930  V=V+A*T9
940  A=-G
950  T8=(V+SQR(V*V-2*A*X))/G
960  IF T8<1-T9 THEN 1600
970  X=X+V*(1-T9)+A*(1-T9)^2/2
980  V=V+A*(1-T9)
990  T=T+1
1000 CALL SOUND(50,SND,0)
1010 SND=SND*.97
1020 GOSUB 1520
1030 T8=(V+SQR(V*V-2*A*X))/G
1040 IF T8<=1 THEN 1090
1050 X=X+V+A/2
1060 V=V+A
1070 GOTO 990
1080 T=T+T9
1090 F=F-B1*T8
1100 T=T+T8
1110 V=V+A*T8
```

```
1120  PRINT "THE TIME ELAPSED IS";T;
      "SECONDS"
1130  PRINT
1140  PRINT "THE VELOCITY ON LANDING
      WAS";V;"FEET PER SECOND"
1150  PRINT
1160  PRINT "FUEL LEFT WAS";F
1170  PRINT
1180  IF V<-1 THEN 1260
1190  ON INT(RND*2)+1 GOTO 1200,1230
1200  PRINT "CONGRATULATIONS,
      COMMANDER▲";NAME$
1210  PRINT "PERFECT LANDING"
1220  GOTO 1600
1230  PRINT "JOB WELL DONE."
1240  PRINT "THE ADMIRAL WAS IMPRESSED"
1250  GOTO 1600
1260  IF V<-5 THEN 1360
1270  ON INT(RND*4)+1 GOTO 1280,1300,1320,
      1340
1280  PRINT "A BIT ROUGH BUT YOU'RE STILLIN
      ONE PIECE"
1290  GOTO 1600
1300  PRINT "I'VE SEEN BETTER, BUT
      I'VE▲▲SEEN A LOT WORSE"
1310  GOTO 1600
1320  PRINT "ANY HARDER AND YOU
      WOULD▲▲▲▲HAVE BOUNCED!"
1330  GOTO 1600
1340  PRINT "HOPE YOUR SHOCKS ARE O.K."
1350  GOTO 1600
1360  FOR C=1 TO 16
1370  CALL SCREEN(C)
1380  NEXT C
1390  CALL SOUND(200,-5,0)
1400  CALL SOUND(200,-6,0)
1410  CALL SOUND(200,-5,0)
1420  CALL SCREEN(13)
```

```
1430  ON INT(RND*4)+1 GOTO 1440,1460,1480,
      1500
1440  PRINT "YOU JUST MADE A HUGE CRATER"
1450  GOTO 1600
1460  PRINT "YOUR NEXT OF KIN WILL
      BE▲▲▲▲NOTIFIED"
1470  GOTO 1600
1480  PRINT "DID YOU PAY YOUR INSURANCE?"
1490  GOTO 1600
1500  PRINT "WE WILL ALWAYS REMEMBER
      YOU.SUCH A BIG DISGRACE TO
      THE▲▲FORCE!"
1510  GOTO 1600
1520  REM DISPLAY STATUS
1530  CALL CLEAR
1540  PRINT "TIME▲▲▲=";T
1550  PRINT "HEIGHT▲=";X
1560  PRINT "VEL.▲▲▲=";V
1570  PRINT "FUEL▲▲▲=";F
1580  PRINT
1590  RETURN
1600  PRINT
1610  FOR T=1 TO 1000
1620  NEXT T
1630  CALL CLEAR
1640  INPUT "WOULD YOU LIKE TO PLAY AGAIN(Y/
      N) ":MORE$
1650  IF SEG$(MORE$,1,1)="Y" THEN 100
1660  END
1670  PRINT ::"YOU ARE THE COMMANDER OF
      A▲▲SMALL LANDER. THE COMPUTER▲▲(NOT A
      TI, OF COURSE) HAS"
1680  PRINT "BROKEN DOWN AND YOU
      MUST▲▲▲▲LAND MANUALLY. CAN YOU
      DO▲▲▲IT? EACH SECOND YOU MUST"
1690  PRINT "SPECIFY THE AMOUNT OF FUEL▲▲TO
      BURN. THE OBJECT IS TO▲▲▲LAND AT THE
      LOWEST SPEED"
```

```
1700 PRINT "POSSIBLE.":"GET READY FOR YOUR
     MISSION, COMMANDER ";NAME$::
1710 GOTO 220
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–180	clears screen and displays the animated heading
190	requests player's name
200–210	asks if instructions are required; if yes, transfers control to line 1670
220–240	prints blank lines and a message
250–260	initializes variables for flight
270–340	asks for target, setting K to 0 for moon, 1 for Earth; if neither, asks again
350	sets the gravity factor
360	sets the maximum thrust possible
370–460	sets initial flight variables
470–540	displays the initial conditions of the landing
550–570	prints a message explaining that the computer will wait until a key is pressed; waits until one is
580	prints a blank line
590–830	main loop, which keeps track of position and speed as time goes on
600	calls routine to print the status
610	requests the thrust to be used this second
620	ignores the sign if included
630	if amount of fuel to burn does not exceed the maximum possible, skips over the error routine
640–650	error routine for excessive thrust request
660–800	calculates changes in acceleration and velocity
810	subtracts the amount of fuel used from the total fuel supply

820	if you are within a very close distance of the target, you have landed; transfers control to line 850
850–860	adds 1 to count of seconds and transfers control to line 1120
870–1110	tests to see if any fuel remains; if not, prints OUT OF FUEL and keeps playing until the crash
1120–1170	prints the final status
1180–1510	prints the rating of the player with a random message based on how fast the player landed
1520–1590	subroutine to clear the screen and print the status
1600–1660	prints a blank line, creates a small time delay, and asks user if another game is desired
1670–1710	routine to print instructions and continue with the game

PROGRAM DESCRIPTION

Of necessity, this program contains many technical details. To a large extent, the equations contained in the program are physically accurate and are thus only meaningful to someone with a solid background in the branch of physics known as mechanics. It would therefore be inappropriate to enter into the details here. Suffice it to say that the equations are correct, although the units are somewhat simplified.

There are a few points that warrant discussion, however. The first is the manner in which the somewhat exotic heading is displayed. To clarify exactly what does happen, first run the program. The screen clears, and the phrase: WELCOME TO THE GAME OF . . . appears, followed by many blank lines. Afterwards the phrase: MOON L is displayed, again followed by several blank lines. Within the FOR . . . NEXT loop contained by lines 140–170, the L character is successively erased (by printing a space char-

acter in its place) and redrawn in the position immediately below the old one. This creates an interesting illusion of motion in which the L (the first letter of the word "landing") seems to fall vertically to the bottom of the screen. When it "hits" the lower edge, the word LANDER appears in its place. The screen then scrolls upwards and the game begins with a request for the player's name. Following that, the player is asked if instructions are required. If they are, they are printed out. Otherwise, an encouraging message is displayed and, after some constants are set, the site of the target is requested. After this final question has been answered to the computer's satisfaction (M for Moon, E for Earth), the flight starts.

First, the initial parameters of the flight are displayed in lines 470–540. Then, in line 550, the player is asked by name to hit any key to begin the flight. Lines 560–570 have the effect of scanning the keyboard until a key is pressed. Control then drops into the main loop for the main action of the game. In line 600, control is sent to the subroutine beginning in line 1520, which displays the current status of the landing vehicle. Then a value for the thrust is requested. This is tested against the maximum permissible level. If it is too great, it is ignored and control is sent back for another attempt. Otherwise, a series of calculations is begun that compute the change in position, velocity, acceleration, and fuel. If the fuel runs out, control drops to line 880 and goes on to crash land as you look on in horror.

Lines 1120–1510 contain the section of code that computes the player's rating. The general class of pilot is determined by the performance, but within the class, a random number determines which of a series of such messages is printed out.

The subroutine in lines 1520–1590 displays the current status for each second of flight. It is called from both the main loop and the auxiliary one which takes charge when the lander runs out of fuel. Lines 1600–1660 ask if the player would like another game. If so, control is sent back to line 100, where the game recommences.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. A great deal of excitement may be added to "Lander" if it is turned into a real-time game. That is to say, the program should go on whether you type in values for thrust or not. If not, it should take the value as either 0 or the last value typed in, at your discretion.
2. The controls can be modified to use joysticks to allow for a more realistic feel.
3. The gravity values of additional planets may be added for more variety and more difficult game levels.
4. The relationship of the maximum thrust possible to the gravity can be changed to allow for greater complexity of play.
5. The scoring can be changed to reflect the number of times the thrust was used (meaning that few adjustments of the flight path were necessary).
6. The scoring can also be changed to reflect the amount of fuel used.
7. Finally, the score can be based on the gentleness with which the velocity is modified (that is, the pilot thought ahead and carefully controlled the landing so that no violent last-minute corrections were necessary).

PROGRAM

22

"ROBOT ATTACK"

PURPOSE

In this arcade-style game, ferocious attack robots converge on you—the hapless player, who appears as a red-colored human on the screen. The robots outnumber you by ten to one and if they touch you, you are dead. You have no weapons with which to defend yourself and so your only alternative is to take advantage of the robots' one weak point—they are extremely stupid and thus move straight towards you no matter where you are. Since they are destroyed if they run into each other or into one of the electrified blockades (shown as blue boxes on the screen) they themselves are destroyed. The trick is to keep dodging the robots and position yourself so as to cause them to collide either with themselves or with the obstacles. Either way they die, although the visual effects are different for each case. Remember not to let them get too close, though, because they are attracted to you as though you were a magnet and thus follow you no matter where you move.

The player moves the human around the screen by means of the eight keys located in a circle around the **D** key. They are: **W, E, R, S, F, X, C,** and **V.** Pretend that the human is standing on the **D** key. In order to move in any direction,

press the key in the corresponding direction. You may take your time about placing your fingers on the required keys because no action is possible until all the ten robots are distributed on the screen.

```
100  REM ROBOT ATTACK
110  CALL CLEAR
120  PRINT "PREPARE YOURSELF FOR
     ...":::::::::
130  PRINT "A ROBOT ATTACK"
140  NB=10
150  NR=10
160  OPTION BASE 1
170  DIM ROBOT(100,2),DXT(26),DYT(26),
     LIVE(100),CLR(2)
180  CLR(1)=2
190  CLR(2)=16
200  FOR I=1 TO 8
210  READ INDEX,DXT(INDEX),DYT(INDEX)
220  NEXT I
230  DATA 3,0,1,5,0,-1,24,-1,1,22,1,1,6,1,
     0,18,1,-1,23,-1,-1,19,-1,0
240  CALL CHAR(128,"1898FF3D3C3CE404")
250  CALL COLOR(13,7,1)
260  CALL CHAR(136,"FFFFFFFFFFFFFFFF")
270  CALL COLOR(14,5,1)
280  CALL CHAR(144,"FFA5A5BDBDA5A5FF")
290  CALL COLOR(15,12,1)
300  CALL CHAR(152,"003C3C3C3C3C3C00")
310  CALL CLEAR
320  DEAD=0
330  X=INT(RND*28)+3
340  Y=INT(RND*24)+1
350  CALL HCHAR(Y,X,128)
360  FOR I=1 TO NB
370  X1=INT(RND*28)+3
380  Y1=INT(RND*24)+1
390  CALL GCHAR(Y1,X1,STATUS)
```

```
400  IF STATUS<>32 THEN 370
410  CALL HCHAR(Y1,X1,136)
420  NEXT I
430  FOR I=1 TO NR
440  LIVE(I)=1
450  X1=INT(RND*28)+3
460  Y1=INT(RND*24)+1
470  FOR DX=-1 TO 1
480  IF (X1+DX>32)+(X1+DX<1)THEN 540
490  FOR DY=-1 TO 1
500  IF (Y1+DY>24)+(Y1+DY<1)THEN 530
510  CALL GCHAR(Y1+DY,X1+DX,STATUS)
520  IF STATUS<>32 THEN 450
530  NEXT DY
540  NEXT DX
550  CALL HCHAR(Y1,X1,152)
560  ROBOT(I,1)=X1
570  ROBOT(I,2)=Y1
580  NEXT I
590  FOR R=1 TO NR
600  CALL SOUND(40,1760,0,1600,0,880,0,
     -4,0)
610  CALL COLOR(16,CLR(R-INT(R/2)*2+1),1)
620  IF LIVE(R)=0 THEN 800
630  CALL KEY(3,KEY,STATUS)
640  IF STATUS THEN 820
650  X0=ROBOT(R,1)
660  Y0=ROBOT(R,2)
670  CALL GCHAR(Y0,X0,STATUS)
680  IF STATUS=152 THEN 710
690  LIVE(R)=0
695  CALL HCAR(ROBOT(R,2),ROBOT(R,1),144)
700  GOTO 800
710  DX=SGN(X-X0)
720  DY=SGN(Y-Y0)
730  CALL HCHAR(Y0,X0,32)
740  ROBOT(R,1)=X0+DX
750  ROBOT(R,2)=Y0+DY
```

```
760  CALL GCHAR(ROBOT(R,2),ROBOT(R,1),
     STATUS)
770  IF STATUS=128 THEN 1070
780  IF STATUS<>32 THEN 980
790  CALL HCHAR(ROBOT(R,2),ROBOT(R,1),152)
800  NEXT R
810  GOTO 590
820  IF (KEY<65)+(KEY>90)THEN 710
830  KEY=KEY-64
840  IF (DXT(KEY)=0)*(DYT(KEY)=0)THEN 710
850  CALL HCHAR(Y,X,32)
860  X0=X
870  Y0=Y
880  X=X+DXT(KEY)
890  Y=Y+DYT(KEY)
900  IF (X>32)+(X<1)+(Y>24)+(Y<1)THEN 940
910  CALL GCHAR(Y,X,STATUS)
920  IF STATUS=152 THEN 1070
930  IF STATUS=32 THEN 960
940  Y=Y0
950  X=X0
960  CALL HCHAR(Y,X,128)
970  GOTO 650
980  DEAD=DEAD+1
990  CALL SOUND(1000,110,0)
1000 IF DEAD=NR THEN 1100
1010 IF STATUS<>152 THEN 1050
1020 DEAD=DEAD+1
1030 IF DEAD=NR THEN 1100
1040 CALL HCHAR(ROBOT(R,2),ROBOT(R,1),144)
1050 LIVE(R)=0
1060 GOTO 800
1070 CALL CLEAR
1080 PRINT "THE HUMAN IS DEAD. LONG LIVETHE
     TI-99/4A."
1090 GOTO 1120
1100 CALL CLEAR
1110 PRINT "YOU HAVE WON FOR THE
     PRESENT▲▲▲▲▲-- YOU VILE HUMAN."
```

```
1120 PRINT :::::"CARE TO TRY AGAIN?";
1130 CALL KEY(3,KEY,STATUS)
1140 IF STATUS=0 THEN 1130
1150 IF KEY=89 THEN 310
1160 IF KEY<>78 THEN 1130
1170 END
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110–130	clears screen and displays the title
140–150	sets the number of blockades and robots
160	sets the lowest subscript to 1
170	reserves room for the arrays
180–190	sets the colors that the flashing robots emit
200–220	reads in the character codes that define the motion of the player
230	data for motion definition
240–300	sets the character representation for the player, the blockades, and dead and live robots
310	clears screen
320	initializes the count of the dead to 0
330–340	sets the coordinates of the player on the screen randomly
350	draws the player at the initial random position
360–420	loop to randomly pick the positions of and draw the blockades; checks before drawing if the space is occupied; if so, computes another position
430–580	loop to randomly pick the locations of the robots, check if the position is empty, and, if so, draw the robot
590–800	main loop, which plays the game
600–610	emits a beep and flashes the color of the robots
620	if the robot is dead, skips its turn
630–640	if a character is typed, control is transferred to line 820

650–660	defines the old coordinates of the robot
670–680	checks if the space on which the robot currently rests is occupied by something; if not, control is transferred to line 710
690–700	kills the robot, erases it from the screen and transfers control to line 800
710–720	sets the direction in which the robot moves
730	erases current position of robot
740–750	modifies coordinates of robot
760–770	checks if the robot has hit something; if it is the human, control transfers to the robot-win routine in line 1070
780	if the robot collides with anything but the human, it dies; control is transferred to line 980
790	if the robot has not collided with anything, draws it in its new position
810	goes back and cycles through the robots again
820–970	player's command routine
980–1060	robot-death routine
1070–1090	routine for robot win
1100–1110	routine for human win
1120–1170	asks if user wishes to play again; if so, goes back; otherwise program ends

PROGRAM DESCRIPTION

The two BASIC features that require explanation in this program are the SGN function and the CALL GCHAR statement. The SGN function, as you will recall, returns the value -1 if the argument is negative, 0 if the argument is 0, and $+1$ if it is positive. It is used to advantage in this program, where it facilitates the programming of the homing qualities of the robots. We simply subtract the coordinates of the robot's position from those of the human's in lines 710 and 720. If the answer is positive, the robot must move in a positive direction. Similarly, if the difference is negative, the motion of the robot is in the opposite direction.

The CALL GCHAR statement is vital to the operation of

this program. It permits the programmer to find out what character is on the screen at a specified position. For example, the program

```
100 CALL HCHAR(5,8,65,20)
110 CALL GCHAR(5,8,STATUS)
120 PRINT STATUS
```

produces the value 65 because the ASCII code of the character in row 5, column 8 is a capital A, having been placed there by the HCHAR statement in line 100. The command is used frequently in this program because it is much faster to check the position on the screen to which a piece (either robot or human) is about to move rather than to keep track of every piece and search for any conflicts. We simply check each time to make sure that the space to which a robot is about to move is blank. If not, it compares the value to the code of the character representing the human. (If it has landed on the human, the game is over.) If this test fails, the robot has either collided with another or with a blockade, in either of which cases it dies.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. We have left the program without a RANDOMIZE statement to enable you to test that it is correctly typed in. Once you are convinced that the program is working as it is supposed to, you can put one in. Then again, you might wish to have the same starting setup every time. If so, use the program as is.
2. The robots can be made to move faster by modifying lines 710–720. Multiplying the value of the expressions by 2 will make them move twice as fast and allow them to leapfrog barriers, unless you get them to position themselves so that they land exactly right.
3. A more sophisticated modification than the one above is to have the robots home in quickly, but, once within a certain radius, change to the normal speed.

4. You can change the number of blockades and/or robots by modifying lines 140 and 150. By decreasing the number of blockades or increasing the number of robots the game is made more difficult and vice versa. You might even want to leave it up to the player and put in statements that allow the player to type in these values.
5. You might want to add in a "hyperspace" button to allow the player to disappear and appear at a random place on the board. This can help the player in a tight spot from which escape is otherwise impossible. Of course, there are certain dangers to hyperspace: Since the player reappears at a random spot, what is to prevent him from rematerializing on a robot (instant death)? You can check for this case or leave it in at your own discretion.

PROGRAM

23

"SNAZZLE"

PURPOSE

"Snazzle" is another fast-action graphics game. This time, you are playing the role of a snake. You start off small (five blocks long) but every time you eat a piece of food, you get longer. Your object is to eat all the pieces of food that appear on the screen while not running into the blue electrified wall—or yourself. If you do hit either of the above-mentioned barriers, you die. You avoid your horrible fate by deftly changing the direction of the snake's motion using the **E, S, D,** and **X** keys to represent up, left, right, and down, respectively. If you survive long enough to eat ten pieces of food, you then move on to the second level, where you have more barriers to run into, making your task even harder. This may sound like a raw deal but, after all —it's a snake's life!

```
100   REM SNAZZLE
110   RANDOMIZE
120   DIM S(100,2),TABLE(26)
130   TABLE(5)=1
140   TABLE(4)=2
150   TABLE(24)=3
```

```
160  TABLE(19)=4
170  FOR LEVEL=1 TO 3
180  CALL CLEAR
190  PRINT "▲▲▲YOU ARE NOW
     ENTERING":::::::::::
200  PRINT "L▲▲▲▲EEEE▲V▲▲V▲▲EEEE▲L"
210  PRINT "L▲▲▲▲E▲▲▲▲V▲▲V▲▲E▲▲▲▲L"
220  PRINT "L▲▲▲▲EEE▲▲V▲▲V▲▲EEE▲▲L"
230  PRINT "L▲▲▲▲E▲▲▲▲▲VV▲▲▲E▲▲▲▲L"
240  PRINT "LLLL EEEE▲▲VV▲▲▲EEEE▲LLLL"
250  PRINT :::TAB(11);"#▲";LEVEL
260  FOR T=1 TO 1000
270  NEXT T
280  CALL CLEAR
290  SCORE=0
300  FOR I=1 TO L
310  S(I,1)=0
320  S(I,2)=0
330  NEXT I
340  ON LEVEL GOSUB 1150,1160,1200
350  X=INT(RND*28)+2
360  Y=INT(RND*22)+2
370  CALL GCHAR(Y,X,T)
380  IF T<>32 THEN 350
390  D=INT(RND*4)+1
400  RC=31
410  CALL CHAR(128,"FFFFFFFFFFFFFFFF")
420  CALL COLOR(13,5,1)
430  CALL CHAR(136,"FFFFFFFFFFFFFFFF")
440  CALL CHAR(144,"00FF7E4C4C7EFF00")
450  CALL COLOR(14,7,1)
460  CALL HCHAR(1,1,128,RC)
470  CALL HCHAR(24,1,128,RC)
480  CALL VCHAR(1,1,128,24)
490  CALL VCHAR(1,RC,128,24)
500  L=4
510  GOSUB 1090
520  REM
```

```
530  FOR I=0 TO L
540  IF S(I,2)=0 THEN 560
550  CALL HCHAR(S(I,2),S(I,1),32)
560  S(I,1)=X
570  S(I,2)=Y
580  CALL GCHAR(Y,X,T)
590  IF T<>32 THEN 790
600  CALL HCHAR(Y,X,136)
610  CALL KEY(3,KEY,STATUS)
620  IF STATUS THEN 740
630  ON D GOSUB 660,680,700,720
640  NEXT I
650  GOTO 530
660  Y=Y-1
670  RETURN
680  X=X+1
690  RETURN
700  Y=Y+1
710  RETURN
720  X=X-1
730  RETURN
740  IF (KEY<65)+(KEY>90)THEN 630
750  KEY=KEY-64
760  IF TABLE(KEY)=0 THEN 630
770  D=TABLE(KEY)
780  GOTO 630
790  REM
800  IF (T=128)+(T=136)THEN 1010
810  CALL SOUND(50,1700,0)
820  CALL SOUND(50,892,0)
830  L=L+3
840  SCORE=SCORE+1
850  IF SCORE=10 THEN 920
860  SCORE$=STR$(SCORE)
870  FOR J=1 TO LEN(SCORE$)
880  CALL HCHAR(1,J+10,ASC(SEG$(SCORE$,
     J,1)))
890  NEXT J
```

```
900  GOSUB 1090
910  GOTO 600
920  FOR I=1 TO 16
930  CALL SCREEN(I)
940  CALL SOUND(50,I*110,0)
950  NEXT I
960  NEXT LEVEL
970  CALL CLEAR
980  PRINT "CONGRATULATIONS!"
990  PRINT " YOU HAVE WON"::::::
1000 END
1010 FOR I=4 TO 1 STEP -.25
1020 CALL SOUND(50,I*110,0)
1030 CALL SCREEN(15)
1040 CALL SCREEN(2)
1050 NEXT I
1060 FOR T=1 TO 1000
1070 NEXT T
1080 END
1090 RX=INT(RND*(RC-2))+2
1100 RY=INT(RND*22)+2
1110 CALL GCHAR(RY,RX,T)
1120 IF T<>32 THEN 1090
1130 CALL HCHAR(RY,RX,144)
1140 REM
1150 RETURN
1160 CALL HCHAR(4,4,128,22)
1170 CALL HCHAR(12,12,128,6)
1180 CALL HCHAR(20,4,128,22)
1190 RETURN
1200 FOR I=5 TO 25 STEP 5
1210 CALL VCHAR(7,I,128,17)
1220 NEXT I
1230 CALL HCHAR(4,12,128,26)
1240 RETURN
```

PROGRAM LINE ANALYSIS

LINE(S)	ACTION(S)
100	REMark
110	reseeds the random number generator
120	sets aside storage for arrays
130–160	sets the letters E, S, D, and X to be direction commands
170–960	main loop in which levels progress
180–250	clears screen and prints the current level in block letters
260–270	delay loop to make the level visible for a few seconds
280	clears screen to prepare for the new level
290	sets score at this level to 0
300–330	initializes snake
340	calls the routines that set up the barriers
350–360	sets random location for snake
370–380	if something is already there, tries again
390	sets the direction randomly to start
400	sets the value of the rightmost column to 31
410–450	sets the shape of the snake, the wall, and the food, together with their colors
460–490	draws the initial positions on the board
500	sets the initial length of the snake
510	calls routine to draw the food
520	REMark
530–650	snake-motion loop
540	tests for special case when snake begins moving
550	blanks out the old tail
560–570	sets new head
580–590	checks if position of new head is not blank; if so, control is transferred to line 790
600	otherwise, draws the head
610–620	if player types a key, control is transferred to line 740

630	calls routines to compute the new position of the head, depending on what direction is current
660–730	movement subroutines
740–780	keyboard input routine
790–910	routine to determine what snake has hit and take appropriate action
790	REMark
800	if the snake runs into itself or the wall, death follows and control is transferred to line 1010
810–820	plays "eating music"
830	adds to the length of the snake
840	adds 1 to the score
850	if the score is 10, goes to the next level
860–890	otherwise, prints the new score
900	puts a new piece of food on the screen
910	transfers control to line 600 to continue moving the snake
920–960	advances a level
970–1000	after all levels are won, prints the winning message
1010–1080	routine to handle snake's death
1090–1140	routine to place a piece of food on the screen
1150–1240	subroutines to place the blockades on the screen at each level

PROGRAM DESCRIPTION

Immediately after the program begins, the random number generator is reseeded and space is set aside for the arrays. Then the table corresponding to the direction of motion of each key is set. You will notice that the program never initializes most of the elements of the array, which are consequently set to 0. This is because only four keys are needed to control the directions in which the snake moves. Whenever a key is typed, its ASCII code (minus 64) is used to subscript the array TABLE and thus determine what effect it has. If the value in TABLE is 0, the computer

continues moving the snake in the same direction that it was already moving, since a zero represents an unused key. After setting TABLE in lines 130–160, the program jumps right into the main loop. Starting at level 1, the level is printed in block letters with a time delay after it is printed, to allow the player time to see it. The screen is then cleared by line 280 and the score for the level is set to 0. The computer then zeros the snake's "links" (picture the computer snake as a chain with many links, each of which has an X and Y coordinate).

At this point the background for the level is drawn depending on what level is currently being played. The subroutines in lines 1150, 1160, and 1200 each draw the background for one level. Following this, the horizontal and vertical position of the snake are determined. Line 370 checks to ensure that the snake does not start on a barrier. If it does, line 380 sends control back to try a new starting position. Line 390 sets the direction to a number between 1 and 4, representing up, right, down, and left respectively.

Since most televisions are incapable of displaying the thirty-second column of the screen, the variable RC (representing rightmost column) is set to 31. If necessary, it can be set even farther to the left. Lines 410–490 set the special characters needed for the game and draw them in their initial positions on the screen. Then the length of the snake (stored in the variable L) is set to 5, its starting value. Line 510 calls the subroutine in line 1090, which places the food on the screen at a random point. The loop from line 530 to line 650 controls the motion of the snake, depending upon which key is pressed. If no key is pressed the snake continues along the path it is already on. If one of the four special keys (**E, S, D,** or **X**) is pressed, control is sent to line 740 where, after validating the key, D (containing the snake's current direction) is set according to the key that was pressed. Whether a direction was set or not, the snake continues to move. If none was set, the direction remains the same. Based on the direction, control is then transferred to one of the subroutines beginning in lines 660, 680,

700, and 720. These subroutines change the horizontal or vertical position as stored in the variables X and Y, depending on the direction.

As the snake crosses the screen it may hit an obstacle or a piece of food. The routine in lines 790–910 determines, once it is known that the snake has hit something, which of the two it has hit. If it dies, control is sent to line 1010. If it has merely eaten some food, control drops, some notes are played, its length is increased (making the play harder), and the score is incremented and displayed on the screen.

If a complete level is won (ten pieces of food are successfully engulfed) control is sent to the routine in line 920. This plays victory notes, shakes the screen with brilliant flashes of color, and ends the game with a congratulatory message.

For those who are not so fortunate, the routine between lines 1010 and 1080 handles the death notes and delay loop to afford the player time to see what happened before the screen colors turn to black on blue again.

The subroutine in lines 1090–1150 places a piece of food on the screen in a random position. This is performed every time a piece is eaten so that there is always exactly one piece on the screen at any time.

Lines 1150–1240 are the setup subroutines that draw the background for each of the levels. They are called from line 340 of the main routine.

POSSIBLE MODIFICATIONS AND ENHANCEMENTS

1. It is easy to add more levels. Simply make a subroutine at the end to draw a pattern of blocks, copying the model of the subroutine in lines 1200–1240. Then, change the ON GOSUB statement in line 340 to include your subroutine and change the FOR statement in line 170 to reflect the new number of levels.
2. By modifying line 830, the amount of length that the snake gains by eating a piece of food may be changed. The faster the snake grows, the more difficult the game becomes because it is so easy for the snake to hit itself.

One particularly worthwhile modification is

```
830  L = L + INT(RND * 10) + 1
```

which increases the size of the snake by a random number of links that may be as little as one or as much as ten.

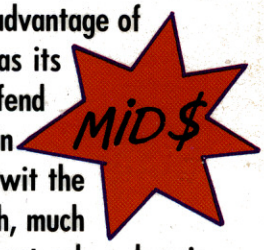
3. You might want to implement a limit on the number of moves of the snake before it must eat the food. If the player doesn't make it in time, have the program put more food on the screen. That is an adequate penalty because it will make the snake much longer (and that much harder to control).
4. You might wish to have some moving objects that kill the snake if they hit its head. They could move either randomly or according to some pattern of your own choosing.
5. You might want to give the player several "lives" in order to prolong the game.
6. In order to make the game even more challenging, you might try to restrict the method of control from the four direction keys to right/left turn keys. In other words, modify them so that the player can make only either a right turn or a left turn—from the snake's point of view. This will tend to disorient the player more.

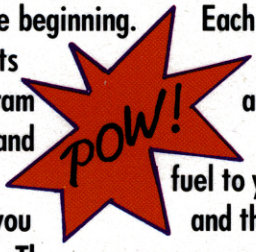
About the Authors

HENRY MULLISH is Senior Research Scientist and Lecturer in Computer Science at the Courant Institute of Mathematical Sciences of New York University. He is the author of over a dozen books on computer programming.

DOV KRUGER went to high school in New York City and attended his first computer course, sponsored by NYU, at the age of 11. At the age of 16 he entered Stevens Institute of Technology where he is now a freshman in the electrical engineering/computer science department. He has previously co-authored *Applesoft BASIC: From the Ground Up* with Henry Mullish.

How would you like to play some great new games on your TI-99/4A *and* learn how to program in BASIC at the same time? With *Zappers* you can do just that. Best-selling authors Henry Mullish and Dov Kruger have created 23 games you can easily input into your TI-99/4A, each one specially designed to take advantage of the 99's terrific color graphics and sound capabilities as well as its ability to play sophisticated games of strategy. You'll have to defend yourself against a deadly Robot Attack, gamble your fortune in Roulette, bring a spaceship to a planet's surface in Lander, outwit the computer in a mind-boggling game of 3D Tic-Tac-Toe, and much, much more. But that's only the beginning. Each chapter includes not only a description of the game and its program listing, but a line-by-line description of the program and a detailed discussion of how the program is constructed and how it actually works. In this way, if you want to add more fuel to your Lander or selectively place some more barriers between you and those robots, you'll be able to program those changes yourself. There are even some suggestions for modifications and enhancements to help you get started. Before you know it you'll be creating your own games—or any other type of program. So whether you just want to play some challenging games on your TI-99/4A or are looking for a fun way to begin programming in BASIC, *Zappers* is the book for you.





Computer Book Division/Simon & Schuster, Inc.
New York

0484-0970

$9.95

0-671-49862-2

Mul
an
Kru
ZAPPERS
COMPU
BOO
SIMO
SCHUS