SPEED WALKER

Fun to Program™ Your TI-99 Series

By Howard Budin

Illustrated by Cris Hammond

US/523-42247-4 * A PINNACLE BOOK * $2.95
CAN/523-43239-9 * $3.50

SPEED WALKER

Fun to Program™ Your TI-99 Series

By Howard Budin

Illustrated by Cris Hammond

Pinnacle Books New York

SPEED WALKER Fun To Program™ Your TI-99 Series

An original Pinnacle Books edition, published for the first time anywhere.

First printing/September 1984

ISBN: 0-523-42247-4

CAN. ISBN: 0-523-43239-9

Printed in the United States of America

PINNACLE BOOKS, INC.
1430 Broadway
New York, NY 10018

9 8 7 6 5 4 3 2 1

Contents

Introduction

Welcome to Fun To Program with Speed Walker! This book has been designed for anyone who already knows some BASIC and is ready to use it to create interesting programs. We assume that you have learned these BASIC commands:

REM	FOR/NEXT	END
PRINT	IF/THEN	
INPUT	GOTO	

Being a programmer is like being a detective--you must constantly uncover mysteries in programs. Programmers spend half their lives searching for clues that tell them what went wrong in their programs. To make it easier for you to read, to modify, and to debug your programs, in this book we use structured programming. This means that we plan the main parts of the program first and then refine the details. We use plenty of REM statements to document the meaning of the variables we use and what the parts of the program do. And we indent parts of the program to show how they relate to each other.

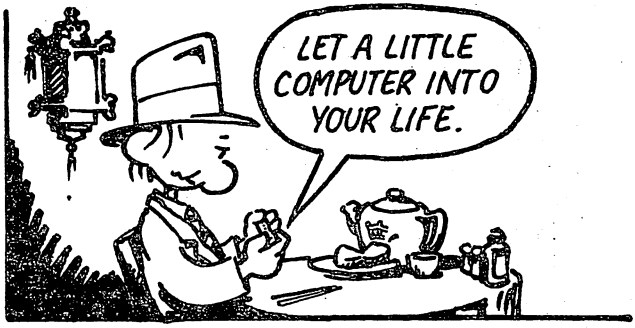
In the vein of being detectives, we are going to learn techniques that are involved with mysteries: telling fortunes, coding and decoding secret messages, and moving objects around the screen. Each chapter adds new concepts and commands and leads to a new game using combinations of these techniques. Every new idea is explained thoroughly, with examples for you to type in and experiment with. By the last chapter we will have built up a much more complex game than the one we start with, but by that time you will be familiar with all the parts that make up the program.

At the end of each chapter we present ideas for variations that you could make in the program. In a sense, programs are never finished--you can always add one more improvement. And that's the beauty of programming: you can take your own original ideas and implement them to the best of your ability. So, although we present programs in these chapters for you to enter and run, the programs are really open-ended. We want you to modify and improve them, add new elements, and make up your own games. Above all, we want you to have fun!

1 What's Your Fortune?

Imagine yourself opening a fortune cookie. Did you ever wonder how that particular fortune got into your cookie? Did somebody want you to have that one? Probably not. Someone must have sat down and made up a lot of fortunes, and someone must have put them all into the cookies, but they did this in a "random" way. In other words, you could have gotten any fortune that had been written.

Computers can do many things, and one of them is telling fortunes. They don't work by magic, though. As you already know, someone has to



program them--tell them exactly what steps to follow. We are going to write a fortune-telling program, and we'll follow the same steps as we would in making fortunes for cookies:

1. Make up a lot of fortunes and store them.
2. Pick a fortune at random.
3. Show the fortune.

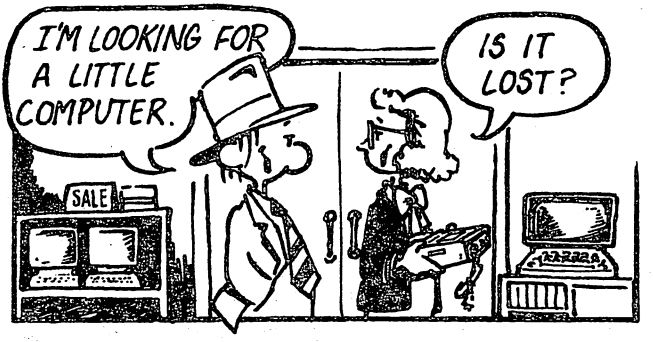
In this book we are going to structure programs so that they are easy to read and to change. We will do this by dividing the program into steps such as the three above, called subroutines. The main body of the program tells the computer where to find each subroutine. The main body reads as if it were a table of contents to the program.

We also use a lot of REMarks to tell us what everything in the program does: what variables we are using, what each subroutine does. Here is how the main body of the fortune-telling program looks:

```
 10 REM ----WHAT'S YOUR FORTUNE?----
 20 REM F$(4) : ARRAY OF 4 FORTUNES
 30 REM J : COUNTER
 40 REM N : RANDOM NUMBER
 50 REM ---------------------------
100 GOSUB 1000
200 GOSUB 2000
300 GOSUB 3000
400 END
```

The first REMark is the name of the program. The next three lines name all the variables we will use in the program. Lines 100, 200, and 300 show where the three subroutines we will use are located. GOSUB 1000 tells the computer to go to line 1000, execute all the instructions there until it comes to the command RETURN, at which point it should come back to where it left off.

STORING THE FORTUNES IN AN ARRAY



Now let's begin writing the subroutines. The subroutine at line 1000 will store as many fortunes as we want in DATA statements. Here is the beginning of the subroutine:

```
1000 REM ----STORE THE FORTUNES--------
1010 DATA ''YOU WILL BE RICH''
```

```
1020 DATA ''YOU ARE VERY FUNNY''
1030 DATA ''YOU LOVE COMPUTERS''
1040 DATA ''YOU WILL GO ON A LONG TRIP''
```

Feel free to use your own fortunes--just make sure you begin each line with DATA and put quotes around the data you write. DATA statements can come anywhere in a program. When the computer finds a READ statement it automatically looks for DATA to read. It starts looking at the beginning of the program and keeps looking until it finds some. The next time it gets a READ command it continues from the last piece of data. We are using four pieces of data, so we need four READ commands:

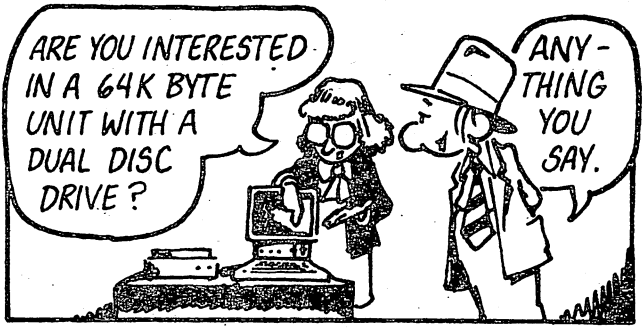
```
1050 DIM F$(4)
1060 FOR J = 1 TO 4
1070     READ F$(J)
1080 NEXT J
1090 RETURN
```

Line 1050 notifies the computer that F$ is not a single variable but will have four elements, or cells. The group of four fortunes is called an array --several pieces of data sharing the same variable name.

The loop in lines 1060 to 1080 instructs the computer to READ four pieces of DATA and store them in F$: the first string (or fortune) is called F$(1), the second is called F$(2), and so on.

Line 1090 makes the computer return to the main program and continue from where it left off. It will now execute line 200, which tells it to go to the subroutine at line 2000 and pick a fortune.

CHOOSING A FORTUNE AT RANDOM



Now that we have four fortunes stored as F$(1), F$(2), F$(3), and F$(4), we need to pick one of them at random. The TI has a built-in function that can choose a different number each time you use it. If we can get it to pick the number 1, 2, 3, or 4, we can use that number to select one of the four cells of F$. We will use the random function so much in this book that we had better spend some time understanding how it works.

Let's work on a little test program. If you are typing in the FORTUNES program, save it and type NEW. Then type in the following:

```
10 FOR X=1 TO 10
20     PRINT RND
30 NEXT X
```

That's the whole program. Run it a few times and study the numbers you get. They should all be decimals between 0 and 1. However, you should get the same random numbers each time. To get different numbers, we need a new TI command, RANDOMIZE. Insert this line in your program:

```
5 RANDOMIZE
```

and run the program a few more times. Now the numbers should all be different. Every time we use the RND function we will use RANDOMIZE at the beginning of the program. Notice that the numbers you are getting are all decimals between 0 and 1. We need bigger numbers, so make this change:

```
20 PRINT 4 * RND
```

Run this version a few times. Now all the numbers should be between 0 and 3.999.... They won't quite get up to 4.

For our fortunes we need the integers 1, 2, 3, or 4--we don't want the decimal part of the number. BASIC has an integer function that chops off the decimal part. Change line 20 again:

```
20 PRINT INT(4*RND)
```

and run it. Now the computer picks decimal numbers at random, multiplies them by four, and chops off the decimal. But there is still one thing wrong--notice that the range of the numbers is from 0 to 3. We want a range from 1 to 4, so one more change is necessary:

```
20 PRINT INT(4*RND)+1
```

This version should give us the numbers we want. Using this formula, we can write the whole subroutine in three lines:

```
2000 REM ----PICK A RANDOM NUMBER-----
2010 RANDOMIZE
2020 N=INT(4*RND)+1
2030 RETURN
```

We are getting a random number from 1 to 4 and storing it as N. All we have to do now is to show fortune number N.

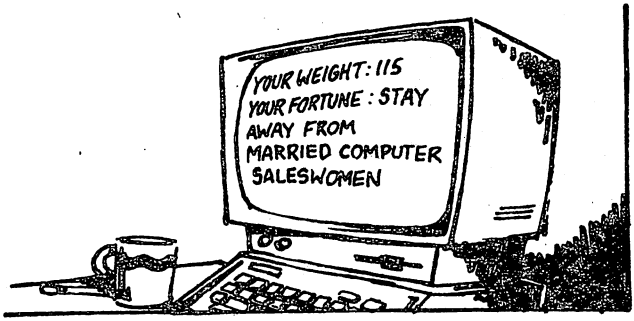
DISPLAYING THE FORTUNE

You should have no trouble in seeing how this subroutine works:

```
3000 REM -----DISPLAY THE FORTUNE-----
3010 CALL CLEAR
3020 PRINT ''HERE'S YOUR FORTUNE:''
3030 PRINT
3040 PRINT F$(N)
3050 RETURN
```

CALL CLEAR erases the screen and line 3040 prints fortune N. What is fortune N? If N equals 1 then the computer will display F$(1),"YOU WILL BE RICH". If N is 3, then F$(3) will be shown, "YOU LOVE COMPUTERS".

PUTTING THE PROGRAM TOGETHER

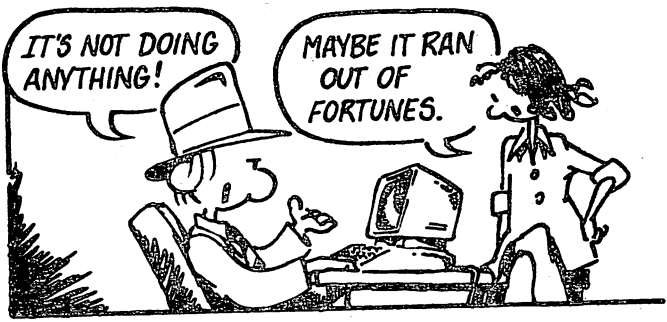


All the subroutines are written and the program is ready to go. Even though you can see the whole program by looking at various places in this chapter, it's a good idea to see it all in one place:

```
 10 REM ----WHAT'S YOUR FORTUNE?-----
 20 REM F$(4) : ARRAY OF 4 FORTUNES
 30 REM J : COUNTER
 40 REM N : RANDOM NUMBER
 50 REM ------------------------------
100 GOSUB 1000
200 GOSUB 2000
300 GOSUB 3000
400 END
1000 REM-----STORE THE FORTUNES------
1010 DATA ''YOU WILL BE RICH''
1020 DATA ''YOU ARE VERY FUNNY''
1030 DATA ''YOU LOVE COMPUTERS''
```

```
1040 DATA ''YOU WILL GO ON A LONG TRIP''
1050 DIM F$(4)
1060 FOR J=1 TO 4
1070     READ F$(J)
1080 NEXT J
1090 RETURN
2000 REM ----PICK A RANDOM NUMBER----
2010 RANDOMIZE
2020 N= INT(4 * RND) + 1
2030 RETURN
3000 REM -----DISPLAY THE FORTUNE----
3010 CALL CLEAR
3020 PRINT ''HERE'S YOUR FORTUNE:''
3030 PRINT
3040 PRINT F$(N)
3050 RETURN
```

VARIATIONS



There are always many things you can do to improve a program and to make it more elabo-

rate. At the end of each chapter we will suggest a few variations you can try and give you hints on how to get started. Here are variations for our fortune cookie program.

1. Four fortunes may not seem like enough of a variety to you. You can make up and store as many as you wish. You will, however, have to change several parts of the program:

a. Add more DATA lines in the first subroutine.
b. Change the number in lines 1050 and 1060.
c. Also change the 4 in the random function (line 2020).

2. Display two (or more) fortunes: Suppose two people want to see their fortunes at the same time. You would need to pick two fortunes -- let's call them N1 and N2:

```
2020 N1=INT(4*RND)+1
2025 N2=INT(4*RND)+1
```

In the last subroutine, you would display both N1 and N2. In this way you could store and print any number of fortunes.

2 A Secret Code

Have you ever used a secret code to communicate with a friend? All codes have rules to follow—if you know the rules, you can decode a message with no trouble. Computers can code and decode very easily, as long as you program in the rules. In this chapter we will write a game program for two people to play. The first will type in a message. The TI will take the message, change it into a code, and then show it to the second player, who will try to decode it.

There are many, many different codes we could use. Here we'll use a reverse code: the computer will take the message and display it backwards. The second player will have to read from right to left to figure it out:

?SIHT DAER UOY NAC

If you can, you'll be able to decode everything in this game. Here is the main body of the program:

```
 10 REM ------SECRET CODE-----------
 20 REM M$ : THE MESSAGE
 30 REM L$ : A LETTER IN THE MESSAGE
 40 REM C$ : THE CODE
 50 REM A$ : ANSWER
 60 REM J : COUNTER
 70 REM ----------------------------
100 GOSUB 1000
200 GOSUB 2000
300 GOSUB 3000
400 END
```

The main body shows us that there are five variable names we will use, and three sections of the program.

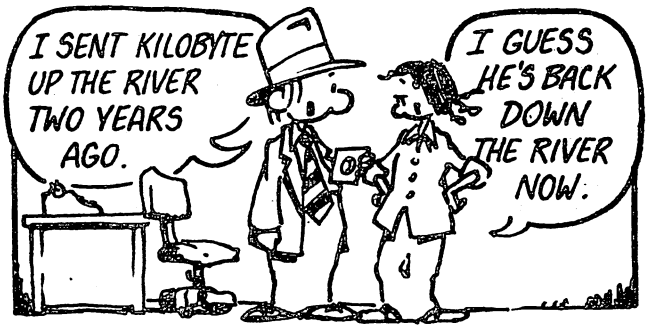
GETTING THE MESSAGE

In the subroutine at line 1000, we ask the second player not to watch while the first player

types in the message. The message will be stored as M$:

```
1000 REM -------GET THE MESSAGE------
1010 CALL CLEAR
1020 PRINT ''ASK YOUR FRIEND NOT TO WATCH''
1030 PRINT ''WHILE YOU TYPE YOUR MESSAGE.''
1040 PRINT
1050 PRINT ''TYPE THE MESSAGE,''
1060 PRINT ''THEN PRESS RETURN:''
1070 INPUT M$
1080 RETURN
```

REVERSING THE MESSAGE



Now the program has a string of characters called M$ -- any group of letters or numbers or other keyboard characters such as dollar signs or parentheses could be in the string. We need to

reverse all the characters in M$ and store them in a new variable we will call C$ (for code). The procedure will be as follows:

1. Take the last character of M$ and make it the first character of C$.
2. Make the second-to-last character of M$ the second character of C$.
3. Make the third-to-last character of M$ the third character of C$.
4. Follow this procedure until we get to the last character of M$, which will become the first character of C$.

Suppose the message is just the word "HELLO". The last character of M$ would be "O", so "O" will be the first character of C$. Then, L will be the second character of C$. When we finish the coding, C$ should be

```
"OLLEH".
```

The TI has a special built-in function that can pick out any character you wish from inside a string of characters. Let's experiment to see how it works. First type

```
M$ = "HORSE"
```

Then ask the TI to

```
PRINT SEG$(M$,4,1)
```

(and press return, of course.)

The TI should respond with the letter S, because S is character number 4 in M$. Now try this command:

```
PRINT SEG$(M$,3,2)
```

The TI will show you two characters in M$, starting at character number 3. You should see the letters RS when you press return. To use the SEG$ function you need three pieces of information inside the parentheses:

(name of string, starting character, how many characters)

In place of the numbers you may use variable names. For example, tell the TI that

```
J = 5
```

and then tell it to

```
PRINT SEG$(M$,J,1)
```

In other words, you want to see 1 character of M$, starting at character number J, or 5. You should see the letter E. What if J is 4? Then SEG$(M$,J,1) is the letter S. By changing the value of J we could look at all the characters in M$.

But how do we know how many characters there are in M$?

Another TI function tells us this fact. Try this:

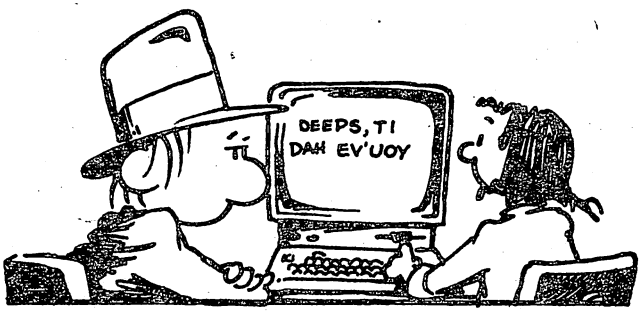
```
PRINT LEN(M$)
```

LEN stands for the length of any string you put inside the parentheses. Let's change M$ as follows:

```
M$ = ''HI THERE!''
PRINT LEN(M$)
```

If you type in these commands, after the second time you press return you will see that the length of M$ is 9. The space and the exclamation point count because they are both part of the string of characters.

Before we write out the subroutine, let's use these new functions in a short test program, just as we did in the first chapter for the random function. The best way to understand any new computer concept is to use it in the simplest program you can imagine.



If you are already typing in our program, save it, type NEW, and type in the following test:

```
10 M$ = ''HELP''
20 FOR J = 1 TO 4
30    PRINT SEG$(M$,J,1)
40 NEXT J
```

Run this program--the output should be the four letters in HELP displayed one to a line on the screen. The first time through the loop J equaled 1 so the first letter of M$ was printed. The second time the second letter was printed, and so on. Now make this change:

```
10 FOR J = 4 TO 1 STEP -1
```

Run the program again. This time the letters are displayed in reverse order. The first time through the loop J equaled 4 so the fourth letter was displayed. J decreased by 1 every time through the loop, so the second time J equaled 3 and the third letter was displayed. Now make one more change:

```
10 FOR J = LEN(M$) TO 1 STEP -1
```

The program should run exactly the same. The computer simply substituted the number 4 for LEN(M$) because there are four characters in M$.

Because the computer can do this, we do not have to know how many characters there are in M$ in advance.

Now we're ready for the coding subroutine:

```
2000 REM -----REVERSE THE MESSAGE------
2010 FOR J = LEN(M$) TO 1 STEP -1
2020     L$ = SEG$(M$,J,1)
2030     C$ = C$ & L$
2040 NEXT J
2050 RETURN
```

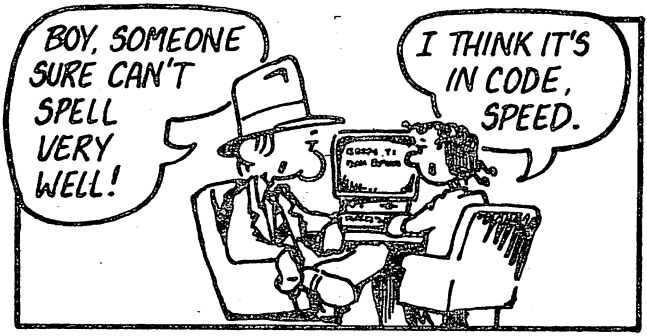
The routine has only a few steps. The FOR/NEXT loop tells the computer to count down starting from the last character until it gets to the first character of M$. If M$ is the word "CAT" the loop will be executed three times. The first time J will equal 3, the second time J will be 2, and the third time J will be 1.

L$ stands for one letter in M$. If M$ is "CAT", the first time through the loop L$ will be "T", the second time L$ will be "A", and the last time L$ will be "C".

Line 2030 takes each L$ and adds it on to the end of C$ (the code). Before we start there is nothing in C$, so the first L$ becomes C$ by itself. After the first time through the loop, C$ will be "T". After the second time, L$ will be "A" and gets added to the end of C$--C$ will now be "TA". After the last time through, "C" will get

added and make C$ "TAC". The original message is now reversed.

QUIZZING THE SECOND PLAYER

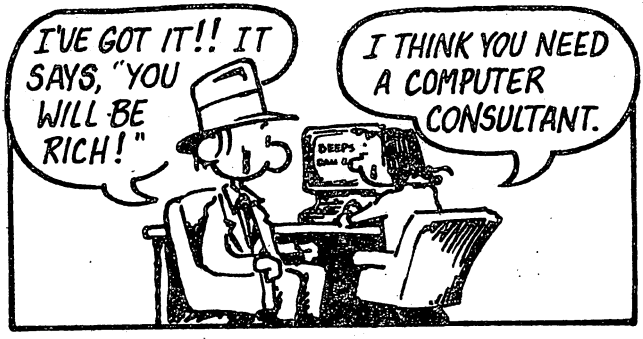


Now that we have the original message (M$) and the coded version (C$) the rest is easy. We clear the screen and ask the second player to look at C$ and type in the original message. We'll call the second player's answer A$. If A$ matches M$ then the player is correct.

```
3000 REM ---------QUIZ------------
3010 CALL CLEAR
3020 PRINT ''ASK YOUR FRIEND TO TRY''
3030 PRINT ''TO DECODE THIS MESSAGE:''
3040 PRINT
3050 PRINT C$
```

```
3060 PRINT
3070 PRINT ''TYPE YOUR ANSWER HERE:''
3080 INPUT A$
3090 IF A$ = M$ THEN 3130
3100 PRINT ''SORRY, THE MESSAGE WAS:''
3110    PRINT M$
3120    GOTO 3140
3130    PRINT ''THAT'S IT!''
3140 RETURN
```

THE WHOLE PROGRAM



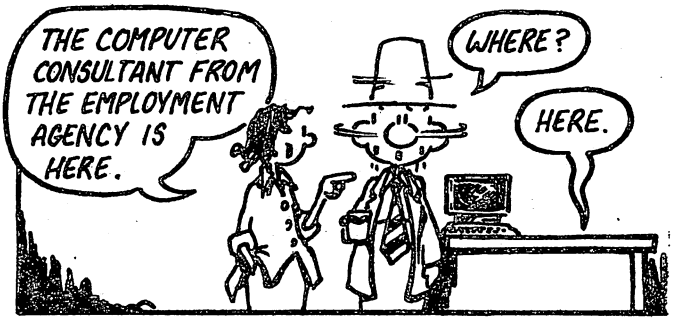
Once again, we will print the whole program in one piece so you can see all the parts of it at once.

```
10 REM --------SECRET CODE------------
20 REM M$ : THE MESSAGE
30 REM L$ : A LETTER IN THE MESSAGE
40 REM C$ : THE CODE
```

```
  50 REM A$ : ANSWER
  60 REM J : COUNTER
  70 REM ----------------------------
 100 GOSUB 1000
 200 GOSUB 2000
 300 GOSUB 3000
 400 END
1000 REM -----GET THE MESSAGE-------
1010 CALL CLEAR
1020 PRINT ''ASK YOUR FRIEND NOT TO WATCH''
1030 PRINT ''WHILE YOU TYPE YOUR MESSAGE.''
1040 PRINT
1050 PRINT ''TYPE THE MESSAGE,''
1060 PRINT ''THEN PRESS RETURN:''
1070 INPUT M$
1080 RETURN
2000 REM -----REVERSE THE MESSAGE-------
2010 FOR J = LEN(M$) TO 1 STEP -1
2020   L$ = SEG$(M$,J,1)
2030   C$ = C$ & L$
2040 NEXT J
2050 RETURN
3000 REM ---------QUIZ------------
3010 CALL CLEAR
3020 PRINT ''ASK YOUR FRIEND TO TRY''
3030 PRINT ''TO DECODE THIS MESSAGE:''
3040 PRINT
3050 PRINT C$
3060 PRINT
3070 PRINT ''TYPE YOUR ANSWER HERE:''
3080 INPUT A$
3090 IF A$ = M$ THEN 3130
3100 PRINT ''SORRY, THE MESSAGE WAS:''
```

```
3110    PRINT M$
3120    GOTO 3140
3130    PRINT ''THAT'S IT!''
3140 RETURN
```

VARIATIONS



1. As it stands, our program always uses the same code. If someone plays a number of times, he or she will probably catch on to the code and the game will stop being fun. Let's work on a subroutine that uses a different code.

There are probably thousands of different codes we could make up. Our fist code simply reversed the message. Another code might insert a letter after every letter of the message. In this case the code for HELLO could be HXEXLXLXOX. One easy change in the coding subroutine will take care of this:

```
2000 REM ------CODE IT----------
2010 FOR J = 1 TO LEN(M$)
2020     L$ = SEG$(M$,J,1)
2030     C$ = C$ & L$ & "X"
2040 NEXT J
2050 RETURN
```

There are really two changes. In line 3010 we count forward instead of backward because we don't want to reverse the characters of the message this time. In line 3030 we add a letter of the message and also an X to the code each time through the loop.

Of course you could use any letter in place of the X. But isn't this code too easy to read also? What if we picked a letters at random and inserted them into the code each time through the loop?

To program this, you have to learn about two more BASIC functions. Every character used by the computer has a code number. The code is called the ASCII code and most computers use it. The ASCII number for a capital A is 65. B is 66, C is 67, and so on until Z, which has the code number 90. You can prove this for yourself. Type:

```
PRINT ASC("A")  or  PRINT ASC("T")
```

or whichever character you want.

The ASC function takes a letter and changes it into a number. We want to do just the opposite -- when we use the RND function we get a random

number. We need to change the number into a letter. The BASIC function CHR$ does just this. Type:

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

and the computer returns the letter A, because A is character number 65 in the ASCII list. The CHR$ function is the opposite of the ASC function.

Now let's change our CODE IT subroutine so that each time through the loop we get a random integer from 1 to 26 (because there are 26 letters in the alphabet) and use that integer in the CHR$ function to get a letter. In this way we get random letters that we can add to the code instead of adding an X every time:

```
2000 REM ----------CODE IT------------
2010 FOR J = 1 TO LEN(M$)
2020     L$ = SEG$(M$,J,1)
2030     R = INT(26 * RND) + 65
2040     C$ = C$ & L$ & CHR$(R)
2050 NEXT J
2060 RETURN
```

In line 2030 notice that we add 65 to the random number. We want the number to be between 65 and 90 because these are the ASCII codes for A and Z. We don't want any random numbers between 1 and 64.

Also notice that line 2040 adds three strings together each time through the loop: what was in

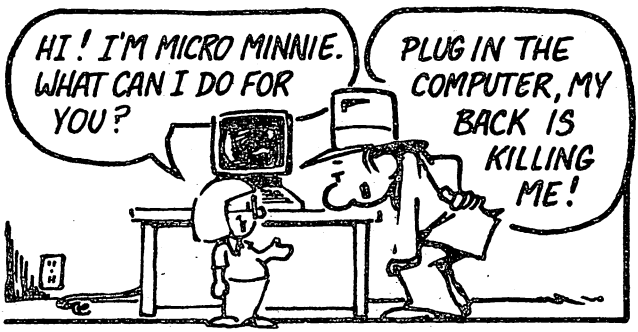
the code already (C$), the next letter of the message (L$), and the letter picked at random (CHR$(R)).

2. Another change would be to give the player more than one chance to get the message correct. Here's how you could go about it. The beginning of the QUIZ subroutine, from line 2000 to line 2070, would remain the same; you would display the coded message and ask for the answer. Next you would set up a FOR/NEXT loop that goes around however many times you wish. Inside the loop you would get the answer and evaluate whether it is right or wrong. If it is wrong you would give the appropriate message and go through the loop again. We leave it to you to develop the code completely.

3 An Initial Race

We have used several important techniques in writing games: different ways to make codes, to store and display messages, and to use the random function. These are found over and over again in games. But so far we have ignored a game technique that the computer does especially well: animation.

Many computer games use some kind of animation (moving objects around the screen). Pro-



grammers spend years studying the most advanced new animation techniques. In this book we will introduce you to character animation--that is, using any of the characters on the keyboard and making it seem to move.

We will start with a race between two initials. On the left side of the screen we'll place the first initials of the two players. When someone presses ENTER to start the race, the initials will take off, at different speeds. When one of them reaches the finish line, the program will anounce who won. Each time you run this program, a random speed will be picked for each initial, so that each could win each time.

The four main parts of the program, as well as the variables we will use, are shown in the main body:

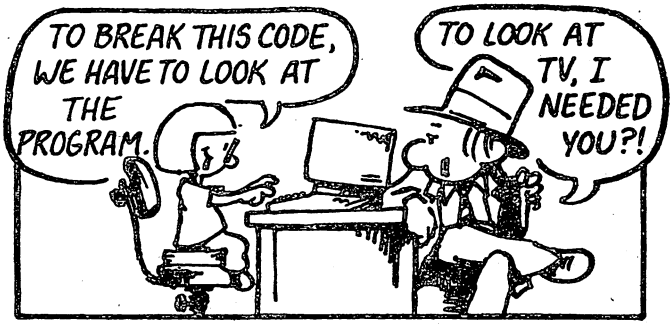
```
10 REM -----INITIAL RACE----------
20 REM A$ : THE FIRST INITIAL
25 REM B$ : THE SECOND INITIAL
30 REM A : ASCII CODE FOR A$
35 REM B : ASCII CODE FOR B$
40 REM SA : SPEED FOR FIRST INITIAL
45 REM SB : SPEED FOR SECOND INITIAL
50 REM CA : COLUMN OF FIRST INITIAL
55 REM CB : COLUMN OF SECOND INITIAL
60 REM W$ : THE WINNER
65 REM R$ : A RESPONSE
70 REM X : A COUNTER
75 REM M$ : A TEXT MESSAGE
```

```
 80 REM L$ : A LETTER IN M$
 85 REM C : ASCII CODE FOR L$
 90 REM ------------------------
 95 RANDOMIZE
100 GOSUB 1000
200 GOSUB 2000
300 GOSUB 3000
400 GOSUB 4000
500 END
```

Subroutine 1000 gets two initials called A$ and B$ for the race. Subroutine 2000 selects speeds for the two initials at random. Up to this point all the commands should be familiar to you. Starting at subroutine 3000 (the race) and continuing through the rest of this book, however, we will handle all input and output in a different way.

Up till now, when we wanted to put a message on the screen we used the command PRINT, and when we wanted the user to respond we used the command INPUT. You may have noticed that each of these commands made the whole screen scroll upward--that is, everything moved up a line. For the rest of this book we want to set up the screen and not have it scroll upward. We also want to be able to place text and get input at any location of the screen. For these purposes we need new functions and commands. Subroutine 4000 of this chapter introduces the CALL HCHAR subprogram for putting output wherever we want it--it announces the winner of the race.

GETTING THE INITIALS



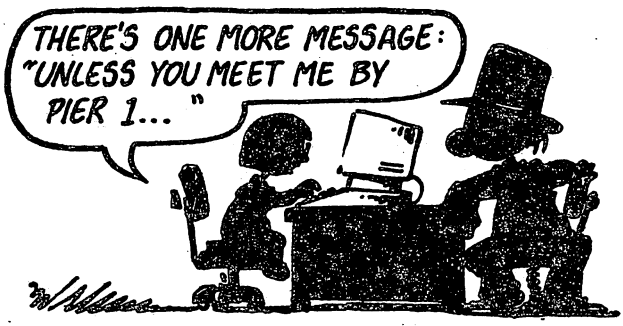
Everything in the first subroutine should be familiar:

```
1000 REM --GET INITIALS---
1010 CALL CLEAR
1020 PRINT ''TYPE THE FIRST INITIAL''
1030 PRINT ''OF PLAYER #1:''
1040 INPUT A$
1050 PRINT
1060 PRINT ''AND THE FIRST INITIAL''
1070 PRINT ''OF PLAYER #2:''
1080 INPUT B$
1090 PRINT
1100 PRINT ''PRESS ENTER TO RACE''
1110 INPUT R$
1120 RETURN
```

We clear the screen with CALL CLEAR, get the first initial and call it A$, get the second initial and

call it B$, and then wait until the player presses ENTER to start the race.

ANIMATING



Before we get to the race itself, let's investigate what animation on a TI is all about. As we did before in learning a new concept, we will write a small test program. If you are currently typing the program from this chapter, make sure you save it (and type NEW) before entering this one.

The kind of animation we will be doing is called "character graphics." This means that we make people believe that something is moving across the screen by printing a character from the keyboard on the screen, erasing it, and printing it a little distance away. By repeating these steps over and over, it appears that the character is moving. These are the steps we will always follow in animating:

1. Display a character on the screen.
2. Repeat some number of times:
   a. Erase the character.
   b. Display the character a little distance away.

The TI screen is divided into 32 rows from left to right and 24 columns from top to bottom. See your User's Reference Guide for a screen map that shows the rows and columns. To place a character at a certain screen location we use the CALL HCHAR subprogram. It is a subprogram because it is stored inside the TI as a little program that lets you place characters on the screen. To use CALL HCHAR you must know three things: the row number you want, the column number, and the character code. Remember the ASCII codes that we used in Chapter 3--character 65 is A, 66 is B, and so on.

Here is the command for putting the letter C (character code 67) on the 5th row and the 20th column of the screen:

```
CALL HCHAR(5,20,67)
```

If you know these three pieces of information that's all there is to it. By the way, there is also an option for repeating the same character as many times as you want horizontally (HCHAR stands for horizontal characters)--simply add a comma and the number of repititions inside the parentheses. Similarly, there is a VCHAR subprogram for displaying and repeating characters vertically on the screen. In this book, however, we do not repeat the same character, so the only command we need is CALL HCHAR with three numbers in the parentheses, as shown above.

Let's use CALL HCHAR to animate the letter Z across the screen. It will move across row 10, from column 5 to column 25.

```
 10 CALL CLEAR
 20 CALL HCHAR(10,5,90)
 30 FOR C = 5 TO 24
 40   CALL HCHAR(10,C,32)
 50   CALL HCHAR(10,C+1,90)
 60   FOR P=1 TO 50
 70   NEXT P
 80 NEXT C
 90 FOR P = 1 TO 2000
100 NEXT P
```

First we clear the screen (line 10) and put a Z on row 10, column 5 (line 20). The loop in lines 30 to 80 erases the Z (character 32 is a space) and

draws it one column over (C+1), from column 5 to column 24. The last time through the loop C will be 24, so the Z will be drawn at C+1, or column 25. The pause loop in lines 60 and 70 slows down the speed of the Z a little. The pause at the end of the program (lines 90 and 100) simply waits a while before ending the program and scrolling up the screen.

Test this program out and try modifying it. Change the row number, the column numbers, the character code, or the length of the pauses. All of our animation will be similar to this. One difference in the race in this chapter, however, is that we don't necessarily want to move one step at a time across the columns. Therefore we will pick random speeds for the two initials and calculate the new column position before we move each initial.

SELECTING THE SPEEDS

Each initial will get a speed of either 1 or 2. This means that before each move we will add either a 1 or a 2 to the column number of that initial and use the result as the new column number.

```
2000 REM--RANDOM SPEEDS--
2010 SA = INT(2*RND) + 1
2020 SB = INT(2*RND) + 1
2030 RETURN
```

SA and SB stand for the speeds of initials A and B. Both SA and SB could be either 1 or 2 each time the program runs. If SA is 1, initial A will move 1 step at a time across the screen. If SB is 2, initial B will move 2 steps at a time and it will win the race.

THE RACE

Storing the speeds of the two initials is not quite enough. We also need to know their column positions at each move. Suppose we decide that whichever initial crosses column 28 first will be the winner. We will start both of them at column 3, but we will not know how far each goes on each move. The two variable CA and CB, standing for initial A's column and initial B's column, will tell us their current position.

Speaking of the winner, when the race is over it would be nice to announce who the winner is. For this, we need another variable, W$, in which to store the initial that crosses the finish line first.

Two last variables (called simply A and B) will store the character codes of the initials. Remember that to use CALL HCHAR we need the character number, not the character itself. Also remember from Chapter 3 that we can find out the code numbers by using the ASC function. ASC(A$) will be the character code for the first initial and ASC(B$) the code for the second.

Now we're ready for the race subroutine. Let's examine it in chunks.

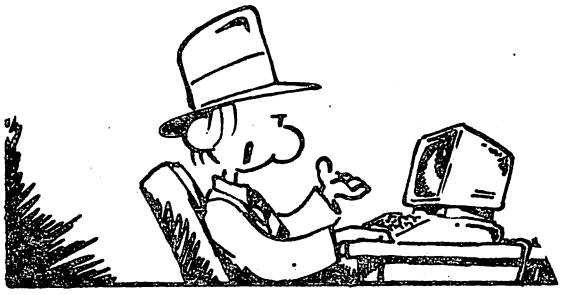
```
3000 REM -----THE RACE--------
3010 CALL CLEAR
3020 A = ASC(A$)
3030 B = ASC(B$)
3040 CA = 3
3050 CB = 3
3060 CALL HCHAR(4,CA,A)
3070 CALL HCHAR(6,CB,B)
```

We clear the screen, use the ASC function to store the code numbers A and B, set CA and CB to 3 because we want them to start on column 3, and finally we use CALL HCHAR to put initial A on row 4 and initial B on row 6, both on column 3.

```
3080 FOR X = 1 TO 1000
3090 NEXT X
```

A little pause before we actually start the race lets the player adjust to seeing the intials before they take off.

```
3100 CALL HCHAR(4,CA,32)
3110 CA = CA + SA
3120 CALL HCHAR(4,CA,A)
3130 IF CA < 28 THEN 3160
3140    W$ = A$
3150    GOTO 3250
```



This section moves initial A. Line 3100 puts a space at its current position. Line 3110 calculates what its new column position should be--either 1 or 2 is added to CA. Line 3120 draws the initial at its new column position. Line 3130 checks the column position. If it is less that 28 (the finish) the program skips to line 3160 and moves initial B. If it is not less than 28, that means initial A has won, so we store A$ in W$ and skip to the end of the subroutine.

```
3160 CALL HCHAR(6,CB,32)
3170 CB = CB + SB
3180 CALL HCHAR(6,CB,B)
3190 IF CB < 28 THEN 3220
3200    W$ = B$
3210    GOTO 3250
```

This section moves initial B in exactly the same way we just moved initial A: we erase the initial where it is, calculate a new position and draw it there, then check whether it has won the race.

```
3220 FOR X = 1 TO 10
3230 NEXT X
3240 GOTO 3100
3250 RETURN
```

If the initial has not yet won (if its column position is less than 28) the program moves to line 3220. Here we pause briefly and go back to line

3100 to move the initials again. If either initial has won, line 3250 returns from the subroutine.

THE WINNER

Only one piece of business remains: telling who won. We could simply say:

```
4000 PRINT ''THE WINNER IS '';W$
```

This would work, but the PRINT command would make the whole screen scroll up a line, including the initials. Here we'll learn a way to avoid that and let you put a message on any part of the screen you want. To do this we will go through several steps:

1. Store the message as a variable (M$).
2. Investigate M$ character by character:

a. Use the SEG$ function to get one character at a time.
b. Use the ASC function to get the code number for the character.
c. Use CALL HCHAR to draw each character next to the last.

This may sound like a complicated way to put a message on the screen, but once you get used to it, it becomes routine. It has the advantage of letting you put text on different parts of the screen without disturbing the screen, and it is fairly short:

```
4000 REM ---THE WINNER-------
4010 M$ = ''THE WINNER IS''
4020 FOR X = 1 TO LEN(M$)
4030   L$ = SEG$(M$,X,1)
4040   C = ASC(L$)
4050   CALL HCHAR(18,X,C)
4060 NEXT X
```

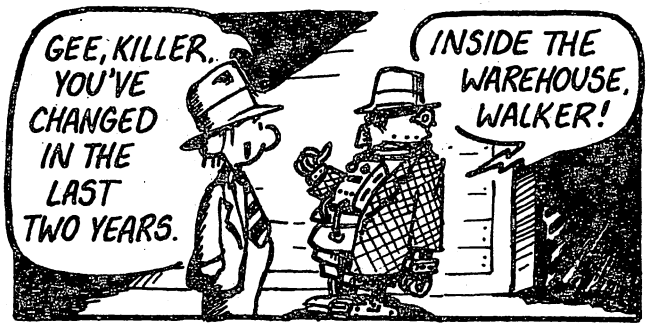
First we want to print THE WINNER IS, so we store this string as M$. The loop in lines 4020 to 4060 needs to go around as many times as there are characters in M$. The LEN function, remember, gives us this number. Line 4030 picks out each character from M$ (one at a time) and calls it L$. Line 4040 gets the code number for L$ and calls it C. Line 4050 draws each character on row 18 of the screen. The first character will be drawn in column 1, because X is 1 the first time though. The second character will be in column 2, and so on. If

we had wanted to begin at column 5, in line 4050 we would have to CALL HCHAR(18,X+4,C), since the first time though X+4 would equal 5.

```
4070 CALL HCHAR(18,15,ASC(W$))
4080 FOR X = 1 TO 2000
4090 NEXT X
4100 RETURN
```

Finally we print W$ at column 15 of row 18 and pause briefly. Notice that we can use the ASC function inside of CALL HCHAR to get the correct character code number.

VARIATIONS



Because of the length of the animation and display subroutines, from here on we are not printing the whole program at the end of the chapter. You

can put it together easily yourself, however, by looking through the chapter. Remember, of course, that the test animation program we wrote is not part of the Initial Race program.

Once you know how it's done, animation gets easier and easier. All you need is practice. Here are several suggestions for improving this game; you can probably think of many more:

1) SCREEN FORMAT: Use CALL HCHAR to make the screen look fancier before the race starts. Put a title for the race at the top of the screen, using subroutine 4000 for a model. Put a border around the racetrack. The command

```
CALL HCHAR(2,1,ASC("-"),30)
```

will put 30 dashes across row 2 of the screen, starting at column 1.

2) MORE SPEEDS: We used speeds of only 1 or 2. You may want more variation. You should

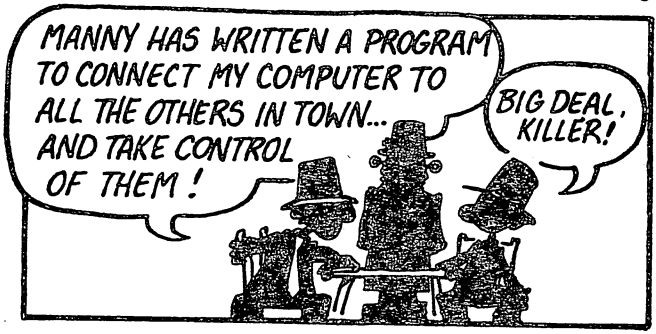
know by now how to use the random function to change the range of the numbers you get. Be careful of one thing, however. If you set the speed very high, the initial will look like it is jumping across the screen, and the screen has only 32 columns across.

3) MORE RACERS: Get three initials instead of two. Set three random speeds. Set all three initials on their marks. And move all three of them. You could even add a fourth or a fifth.

4 Burst the Balloon

The initial race was a fairly simple game -- the whole game was the race. Animation can be used as a part of more complex games, though. Imagine this screen: on the right is a "balloon" and on the left is a person's initial. On the lower half of the screen the player gets arithmetic problems which he or she tries to solve. When an answer is correct, the player's initial moves across the screen and "bursts" the balloon, which turns out to have a message inside it.

This game combines many elements we have used before: animation, random numbers, using



DATA statements, giving the player several tries. Good games are created by combining many different elements, but none of the elements has to be complicated by itself.

One more TI programming concept needs to be introduced. As we mentioned in the last chapter, from now on we must handle input and output differently if we want to preserve our screen setup. We have already looked at CALL HCHAR to produce output. In this game we also need player input. The CALL KEY subprogram gets a character number when a key on the keyboard is pressed and transfers it to the program.

With both CALL HCHAR and CALL KEY we need to switch back and forth from character names to character codes, and also from codes to names. This increases the number of variables we must use. Sometimes we need numbers like X and Y (the random arithmetic numbers in this game), but we also need to make them into strings (X$ and Y$) so that they can be part of a string for CALL HCHAR to output.

```
  5 REM ----BURST THE BALLOON-----
 10 REM I$ : THE INITIAL
 15 REM R : ROW NUMBER
 20 REM C : COLUMN NUMBER
 25 REM M$ : TEXT MESSAGE
 30 REM L$ : LETTER IN TEXT
 35 REM A : ASCII CODE NUMBER
 40 REM J : A COUNTER
 45 REM K : KEY NUMBER
```

```
 50 REM S : STATUS OF KEYBOARD
 55 REM R$ : RESPONSE
 60 REM X : RANDOM NUMBER
 65 REM Y : RANDOM NUMBER
 70 REM X$ : STRING OF X
 75 REM Y$ : STRING OF Y
 80 REM V : VALUE OF RESPONSE
 85 REM ------------------------
 90 RANDOMIZE
100 GOSUB 1000
200 GOSUB 2000
300 GOSUB 3000
400 END
```

SETTING UP THE SCREEN

This subroutine has several little parts. First, we put a title for the game across the top of the screen. Second, we draw the balloon. Third, we get the initial and put it to the left of the balloon.

Last, we explain the game. Let's look at each part individually.

```
1000 REM -----SET UP----------
1010 CALL CLEAR
1020 M$ = ''BURST THE BALLOON''
1030 R = 2
1040 C = 7
1050 GOSUB 1500
```

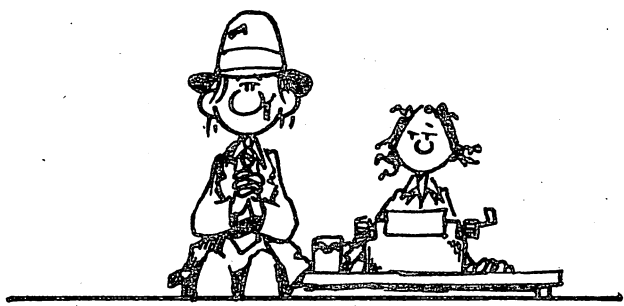
Every time we have text to put on the screen we will handle it the same way we handled the message THE WINNER IS in Chapter 4, using CALL HCHAR. In this chapter we have many messages to output, so we will make the code to draw them a separate subroutine, located at line 1500. This subroutine needs three pieces of information: the row number (R), the column number (C), and the text message (M$). We'll look at subroutine 1500 at the end of this section. For now, it's enough to know that the text BURST THE BALLOON will be put on row 2, staring at column 7.

```
1060 DATA 4,25,4,26,5,23,5,24,5,27,5,28
1070 DATA 6,22,6,29,7,22,7,29,8,21,8,30
1080 DATA 9,21,9,30,10,22,10,29,11,22,11,29
1090 DATA 12,23,12,24,12,27,12,28,13,25,13,26
1100 FOR J = 1 TO 24
1110    READ R,C
1120    CALL HCHAR(R,C,ASC(''O''))
1130 NEXT J
```

This part draws the balloon. The balloon is made up of 24 letter O's. The first one goes on row 4, column 25. The second is on row 4, column 26. Instead of using 24 CALL HCHAR commands to draw the balloon, we READ from lines of DATA. The first two data numbers are the first row and column number. The loop in lines 1100 to 1130 reads in two pieces of data at a time, calls them R and C, and uses them in the CALL HCHAR command. Thus 24 O's are drawn where we want them.

```
1140 M$ = ''TYPE YOUR FIRST INITIAL:''
1150 R = 16
1160 C = 1
1170 GOSUB 1500
```

Here we put the message TYPE YOUR FIRST INITIAL: on row 16, column 1, using subroutine 1500.



At this point we need the player to give us input. As explained before, if we use the INPUT command the screen will scroll up and the input will be at the bottom of the screen. The TI has another subprogram called CALL KEY that transfers input from a key of the keyboard to the program. Here is the form of the command:

```
CALL KEY(mode,key number,status)
```

There are several keyboard modes, in which the keys can have different code numbers. For our purposes, we will always use mode 5, the BASIC mode. You know that every key (every character) has its own code number. The CALL KEY command checks the keyboard to see if a key has been pressed. If it has, the ASCII code number for that key is transferred into the program and given the name of the variable you put inside the parentheses. We will always use the variable name K to stand for the key number. The final variable inside the parentheses (which we call S for status) stores either the number 0 if a key has not been pressed or a 1 if it has. This is called the status of the keyboard. Here's how we use CALL KEY at this point:

```
1180 CALL KEY(5,K,S)
1190 IF S = 0 THEN 1180
1200 CALL HCHAR(9,5,K)
1210 I$ = CHR$(K)
```

We need to know the status of the keyboard because CALL KEY does not sit and wait for someone to press a key. It simply checks the keyboard at the instant the program comes to the CALL KEY command. Therefore line 1190 checks to see if a key has been pressed. If no key has been pressed we go back to CALL KEY repeatedly until a key has been pressed. Then line 1200 uses the key number (K) for the CALL HCHAR command to draw the intital on row 9, comumn 5. We need to store the initial for later in the program. K, remember, is a code number. The CHR$ function gives us the character name, or string, which we store as I$.

```
1220 M$ = ''ANSWER 1 QUESTION CORRECTLY.''
1230 R = 18
1240 C = 1
1250 GOSUB 1500
1260 M$ = ''AND YOUR INITIAL WILL''
1270 R = 19
1280 GOSUB 1500
1290 M$ = ''BURST THE BALLOON.''
1300 R = 20
1310 GOSUB 1500
1320 RETURN
```

We have three more lines of text to put on the screen. We assign each of them to M$, give them column and row numbers, and call subroutine 1500. Notice that after line 1240 assigns the number 1 to C, we don't have to worry about C

again, because we want each message to start at column 1. We do have to change the row number for each message, though.

Now for the subroutine that puts the text on the screen.

```
1500 REM ----TEXT ON SCREEN----
1510 FOR J = 1 TO LEN(M$)
1520    L$ = SEG$(M$,J,1)
1530    A = ASC(L$)
1540    CALL HCHAR(R,C+J,A)
1550 NEXT J
1560 RETURN
```

This routine does just what we did in the last subroutine of the Initial Race. It takes M$, examines it character by character with the SEG$ function, translates the character to its code number using the ASC function, and use CALL HCHAR to draw each character on the screen at successive column positions.

Let's trace how this routine works. Suppose that M$ is BURST THE BALLOON. This string has 17 characters (counting spaces), so LEN(M$) is 17. The first time through the loop L$ is B, the first character of M$. A is the ASCII code for B (66). Therefore the CALL HCHAR subroutine puts the letter B on row R, column C+1. The next character will be put on row R, column C+2, and so forth.

ASKING THE QUESTION

In this section we pose arithmetic problems until the player gets one right. To pose the problem, we pick two random numbers called X and Y, ask how much is X plus Y, get the input, and evaluate whether the input is correct. Here is the code:

```
2000 REM ---QUESTION(S)----
2010 X = INT(50 * RND) + 1
2020 Y = INT(50 * RND) + 1
2030 GOSUB 2500
2031 REM (2500 ERASES THE BOTTOM OF THE
     SCREEN)
2040 X$ = STR$(X)
2050 Y$ = STR$(Y)
2060 M$ = ''HOW MUCH IS '' & X$ & '' + '' & Y$ &
     ''?''
2070 R = 16
2080 C = 1
2090 GOSUB 1500
```

```
2100 R = 17
2110 C = 3
2120 GOSUB 2700
2121 REM (2700 GETS INPUT)
2130 V = VAL(R$)
2140 IF V <> X+Y THEN 2010
2150 RETURN
```

Lines 2010 and 2020 get two random numbers between 1 and 50. You could substitute any number you want for the 50. The subroutine located at line 2500 clears the bottom of the screen so that we can put new text on it. We'll look at it after this section.

Next we want to put a text message on the screen. The message is composed of five parts:

1) the words HOW MUCH IS
2) the first number (X)
3) a plus sign (+)
4) the second number (Y)
5) a question mark (?)

If X and Y were 34 and 12, then the message should like like this: HOW MUCH IS 34 + 12?

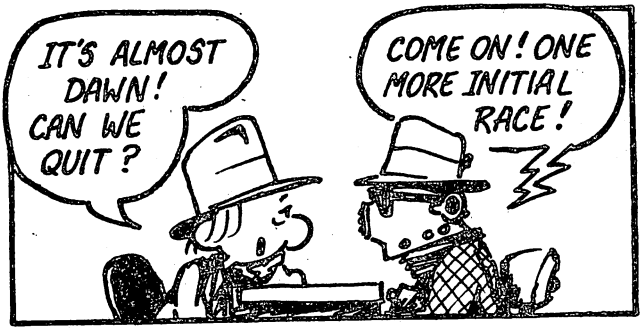
Each of these five parts of the text must be a string. Since X and Y are numbers, not strings, we have to make strings out of them with the STR$ function. STR$ examines what follows it in the parentheses. If this is a number, the STR$ function makes it into a string. Thus, X$ is the string made

out of X, and Y$ is the string made out of Y. X and X$ will look identical when put on the screen, but they are stored differently, they have different code numbers, and you can use X$ as part of another string. Line 2060 puts five strings together to make up M$, which is then output on the screen by subroutine 1500.

Next we must get the player's input. We set the row to 17 and the column to 3 and call another subroutine at line 2700 which uses the CALL KEY command over and over again to get characters and output them on the screen with CALL HCHAR. This subroutine (which will be explained after this section) gets input one character at a time and puts the characters together into a string called R$. We need to evaluate whether the input is correct, but we cannot compare a string to a number. So we need another function which is the opposite of the STR$ function. The VAL function takes a string and changes it into a number, so that in line 2130 V becomes the number that represents R$. One thing to be careful of: the input must be all numbers. If the player presses a key which is not a number, that character will get into R$ and the VAL function will not work--a number must be the input to VAL in the parentheses.

Line 2140 compares V to X+Y. If they are equal, the answer is correct and we return from the subroutine. If they are not equal, the program branches back to the beginning of the subroutine and picks two more random numbers.

CLEARING THE SCREEN

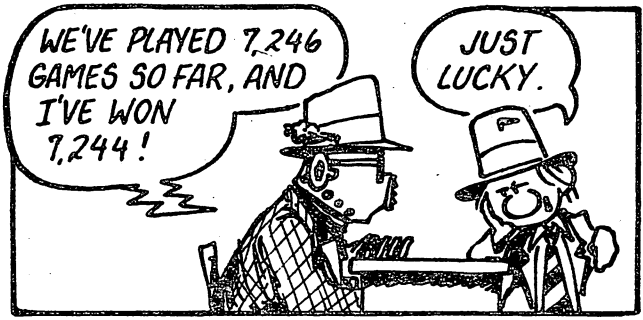


This subroutine uses the CALL HCHAR subprogram to output spaces (character number 32) at all the column numbers on rows 16 to 20. We need to erase all these five rows because in the SET UP subroutine we put text from rows 16 to 20.

```
2500 REM ---ERASE 16 TO 20---
2510 FOR R = 16 TO 20
2520   FOR C = 1 TO 32
2530       CALL HCHAR(R,C,32)
2540   NEXT C
2550 NEXT R
2560 RETURN
```

Each time we come to line 2530 R and C are different. Therefore character 32 (the space) will be drawn at 5 times 32 different screen locations (5 rows times 32 characters per row).

GETTING INPUT WITH CALL KEY



We have already used this subprogram to get the initial. That was only one character of input, but now we need more than one character. Rather, we do not know how many characters there will be--that is up to the player. Therefore, we need a subroutine that repeats until the player presses the ENTER key (ENTER is code number 13). After each key is pressed, two things will happen: that character will be drawn on the screen, and it will be added to R$ so that later we will know what the whole input was.

```
2700 REM ---GET INPUT----
2710 R$ = ""
2720 CALL KEY(5,K,S)
2730 IF S = 0 THEN 2720
2740 IF K = 13 THEN 2810
2750 CALL HCHAR(R,C,K)
```

```
2760 R$ = R$ & CHR$(K)
2770 C = C + 1
2780 FOR J = 1 TO 30
2790 NEXT J
2800 GOTO 2720
2810 RETURN
```

Line 2710 initializes, or sets, R$ to an empty string--there are no spaces between the quotation marks--since we don't want any text stored in R$ each time we get new input. If there was text in R$, line 2710 will get rid of it. Line 2720 and 2730 get input of one key (as shown earlier in this chapter). After a key has been pressed, line 2740 evaluates whether the code number for the key is 13. If it is, that means that ENTER has been pressed, so we exit from the subroutine.

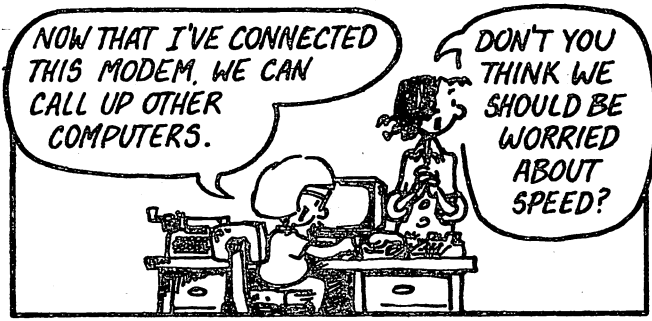
If K does not equal 13, then line 2750 draws that character on the screen. Remember that R was set to 17 and C was set to 3 before this subroutine was called. Therefore the first character of input will be drawn at row 17, column 3. Line 2770 adds 1 to C so that the next character will be drawn one column over.

Line 2760 adds the input to R$. Since all we know about the input is its code number (K), we use the CHR$ function to get a character to add to R$.

Before we go back to the CALL KEY statement to get the next character of input, we add a little pause (lines 2780 and 2790); we make the TI

count to 30. We need to do this because computers work so quickly. With no pause, if the player holds down a key for even a second, the computer will be back to line 2720 before that second is up and it will think the same key is the next piece of input. To avoid having keys repeated if held down, we make the computer wait a little bit before checking the keyboard again. The drawback with doing this is that if the player presses keys too quickly, they will not register. You can adjust the length of the pause to suit you.

BURSTING THE BALLOON



The QUESTION(S) subroutine asks random arithmetic problems until the user gets one correct. Each time it clears the screen, asks a new question, gets new input, and evaluates it. When the input is

correct, the program proceeds o the subroutine located at line 3000.

The BURST subroutine must do three jobs: 1) animate the initial over to the balloon; 2)make the balloon disappear; and 3)print the message RIGHT! "inside" the balloon. We'll look at these three tasks separately.

```
3000 REM -----BURST------
3010 FOR C = 5 TO 20
3020     CALL HCHAR(9,C,32)
3030     CALL HCHAR(9,C+1,ASC(I$))
3040     FOR J = 1 TO 10
3050     NEXT J
3060 NEXT C
```

The initial (I$) begins at column 5. We want to move it over to the balloon, to column 21. The loop in lines 3010 to 3060 increases C from 5 to 20. Each time through it draws a space at column C and draws the initial at column C+1. ASC(I$) is the code number of the initial, and remember that CALL HCHAR needs this code. The last time through the loop C will be 20, so a space will be drawn at column 20 and the initial will be drawn at column 21. This loop looks like our sample animation program in the last chapter. Here we know exactly where we want to start and end, and we want the initial to move one column each time. Notice that we include a slight pause to slow down the animation.

```
3070 RESTORE
3080 FOR J = 1 TO 24
3090     READ R,C
3100     CALL HCHAR(R,C,32)
3110 NEXT J
```

We have all the row and column numbers of the balloon stored in data statements--we used them to draw the balloon. The command RESTORE sets the data pointer back to the beginning of the data. In this way we can use all the same data over again. This loop does just what the loop in the SET UP subroutine did to draw the balloon. It READs in all the row and column data and uses these numbers to draw spaces (characters 32), thus erasing the balloon.

```
3120 M$ = ''RIGHT!''
3130 R = 9
3140 C = 23
3150 GOSUB 1500
```

This little section should be familiar by now. We set M$, R, and C and pass them along to the TEXT ON SCREEN subroutine. The word RIGHT! gets drawn at column 23 of row 9, looking like it was inside of the balloon.

```
3160 FOR J = 1 TO 2000
3170 NEXT J
```

We insert a pause so that the player gets a final look at the screen before the program stops running.

VARIATIONS

Once again, you can put the program together by looking through the chapter. This game has introduced several new concepts and BASIC words: the VAL and STR$ functions, CALL KEY to get input, erasing parts of the screen, and RESTORE to reset the data list. Here are two suggestions for improving the program:

1) The GET INPUT subroutine works correctly as it stands. However, it can be improved. We already mentioned the length of the pause before each new CALL KEY statement. Another problem involves mistakes: What if the player makes a mistake? The subroutine does nothing about this.

You can define any key you want as a delete key. Let's say that you want the D key to delete the last character that the player typed. If so, you should probably put a message somewhere on the screen so that the player will know this.

When a key is pressed, the program evaluates whether it was the ENTER key. If it was, the subroutine ends. If the key was not ENTER, you can make the program evaluate if the key was D:

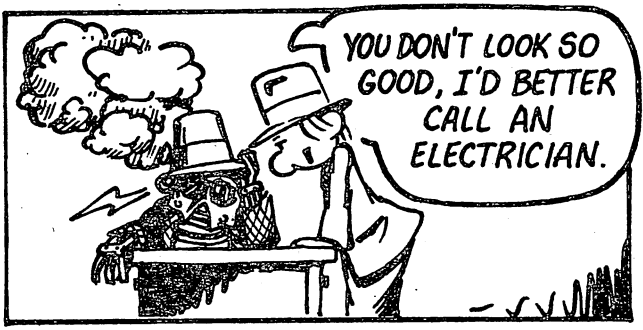
```
2745 IF K = ASC(''D'') THEN (line number)
```

If the key is a D, the program will go to a line number of your choice. There it should subtract 1 from C and draw a space at column C, then go back to the CALL KEY command. With these hints we leave the execution to you.

2) Subroutine 2500 erases rows 16 to 20. It erases all these rows because in the SET UP section we put text on all of them. However, from the time we begin asking questions, we really only use rows 16 and 17, and the program seems slow because subroutine 2500 erases the other three rows even if nothing is on them. You could write another subroutine which erases only rows 16 and 17, and direct the program to it when you want only those lines erased.

5 Scrambled States

All of the techniques we have learned, plus some new ones, will be combined here to create a longer game program than our others. In this game it will take three moves to get from the start to the finish line. To move, the player will be presented with the name of one of the states of our nation, but the name will be scrambled and the player will have to figure out which state it is. We will give the player six attempts to make the three moves. If he or she cannot make it to the finish line by then, the game will end.



The variable names and subroutine list look like this:

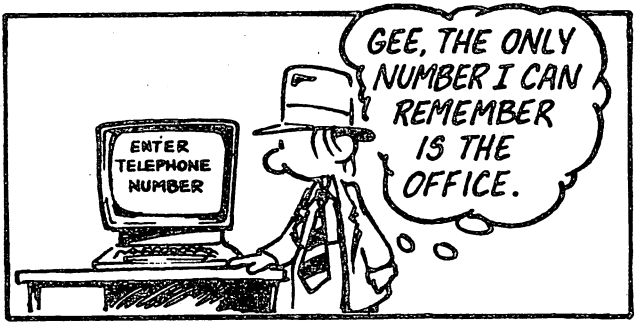
```
  4 REM ---------SCRAMBLED STATES--------
  8 REM R : ROW NUMBER
 12 REM C : COLUMN NUMBER
 16 REM M$ : TEXT MESSAGE
 20 REM L$ : LETTER IN TEXT
 24 REM A : ASCII CODE NUMBER
 28 REM I : A COUNTER
 32 REM J : A COUNTER
 36 REM K : KEY NUMBER
 40 REM S : STATUS OF KEYBOARD
 44 REM I$ : THE INITIAL
 48 REM T$(4) : ARRAY OF 4 TEXTS
 52 REM S$(10) : ARRAY OF 10 STATES
 56 REM U$(20) : ARRAY OF USED LETTERS
 60 REM A$ : ANSWER
 64 REM N$ : NAME OF STATE
 68 REM T : TURN NUMBER
 72 REM P : POSITION OF INITIAL
 76 REM M : MOVE
 80 REM O$ : GAME OVER?
 84 REM L : LENGTH OF N$
 88 REM X : RANDOM NUMBER
 92 REM ------------------------------
 96 RANDOMIZE
100 GOSUB 1000
200 GOSUB 2000
300 GOSUB 3000
400 END
```

We introduce no new BASIC concepts or commands in this game, but we use the ones we know in more complex ways. You can tell by the long list of variable names that we need to store many different pieces of data. We also use several subroutines--the main body of the program calls only three of them, but these subroutines call others. Since we use eight subroutines altogether, it might be a good idea to list them here for reference:

```
1000   Set up the screen
1500   Put text on the screen
2000   Load array of states
2300   Erase rows 15 to 21
2700   Get input
3000   The game
3500   Scramble a state
3800   Move the initial
```

Some of these will be familiar; we have already used subroutines to get input, erase part of the screen, put text on the screen, move an initial, and load an array of text. These little subroutines are useful tools--they become a kind of library for the programmer to use over and over again. The only really different routines in this program are the GAME and the SCRAMBLE subroutines.

SETTING UP THE SCREEN



To prepare the player for the game, we set up the screen to show that the initial must move through three stations to get to the finish. Once again, we leave it to you to make the "board" look more elaborate. You can easily add decorations. In this subroutine, we represent the stations with plus signs, get the initial and place it at the start, and explain how the game works.

```
1000 REM--SET UP THE SCREEN----
1010 CALL CLEAR
1020 M$ = ''START''
1030 R = 4
1040 C = 3
1050 GOSUB 1500
1060 CALL HCHAR(5,5,ASC(''+''))
1070 CALL HCHAR(5,13,ASC(''+''))
1080 CALL HCHAR(5,21,ASC(''+''))
1090 CALL HCHAR(5,29,ASC(''+''))
```

```
1100 M$ = ''FINISH''
1110 R = 4
1120 C = 25
1130 GOSUB 1500
```

The subroutine at line 1500 puts text on the screen. It is identical to subroutine 1500 of Chapter 5. It uses the variables M$, R, and C for the message, row, and beginning column number. Here we put the words START and FINISH on row 4 and four plus signs on row 5. They should look like this:

```
START                          FINISH
  +        +        +        +
```

Next we ask the player for his or her initial and use CALL KEY to get input. We use CALL HCHAR to draw the initial where the first plus sign was.

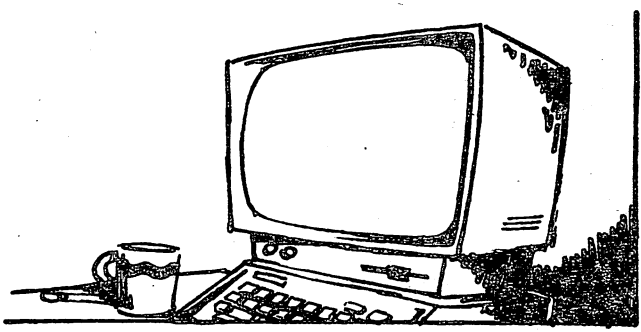
```
1140 M$ = ''TYPE YOUR FIRST INITIAL:''
1150 R = 15
1160 C = 1
1170 GOSUB 1500
1180 CALL KEY(5,K,S)
1190 IF S = 0 THEN 1180
1200 I$ = CHR$(K)
1210 CALL HCHAR(5,5,K)
```

CALL KEY gets input of a code number which represents the key pressed. We assign CHR$(K) to

the variable I$ so that later we will know what the initial is when we want to move it. Line 1210 draws character K on row 5, column 5.

Now we need four lines of text to explain the game. Every time we have put a single line of text on the screen so far we have needed four program lines: one to assign the text to M$, one to assign a row, one to assign a column number, and one to GOSUB 1500. Lines 1140 to 1170 above do just this. In this way, to put four texts on the screen would take 16 program lines. Here is a shorter way, using an array of texts:

```
1220 T$(1) = "YOU HAVE 6 TRIES TO WIN."
1230 T$(2) = "TO MOVE, UNSCRAMBLE A STATE"
1240 T$(3) = "OF THE UNITED STATES."
1250 T$(4) = "PRESS ENTER TO BEGIN:"
1260 C = 1
1270 FOR I = 1 TO 4
1280    M$ = T$(I)
```

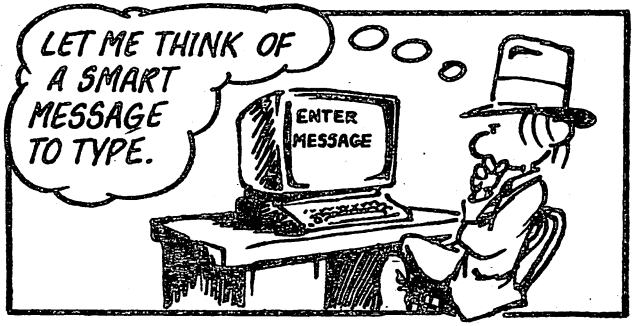


```
1290    R = I + 16
1300    GOSUB 1500
1310 NEXT I
```

Using this technique you could handle as many lines of text as you want. First assign them to cells of an array--here we have 4 cells of T$. Line 1260 assigns 1 to C. Since we want all the lines to start at column 1, we don't have to assign C again. The loop in lines 1270 to 1310 repeats 4 times--once for each text. Line 1280 assigns each text of the array to M$. Line 1290 makes sure that R is first 17, then 18, then 19, then 20 the last time through. Now that we have M$, R, and C assigned we call subroutine 1500 to put M$ on the screen. Notice that we need a new counter (I) because subroutine 1500 uses J for a counter and we want to avoid mixing up the variable names.

```
1320 CALL KEY(5,K,S)
1330 IF S = 0 THEN 1320
1340 RETURN
```

Our last line of text asked the player to press enter to begin. Lines 1320 and 1330 get one key of input. We don't really care what key it is--as soon as the player presses any key the program will move on. The program now moves to STORING THE STATES, but first let's look at input and output



These two subroutines are the same ones we used in the last game. We reproduce them here for your convenience.

```
1500 REM --PUT TEXT ON SCREEN---
1510 FOR J = 1 TO LEN(M$)
1520     L$ = SEG$(M$,J,1)
1530     A = ASC(L$)
1540     CALL HCHAR(R,C+J,A)
1550 NEXT J
1560 RETURN

2700 REM -------GET INPUT--------
2710 A$ = ""
2720 CALL KEY(5,K,S)
2730 IF S = 0 THEN 2720
2740 IF K = 13 THEN 2810
2750 CALL HCHAR(R,C,K)
```

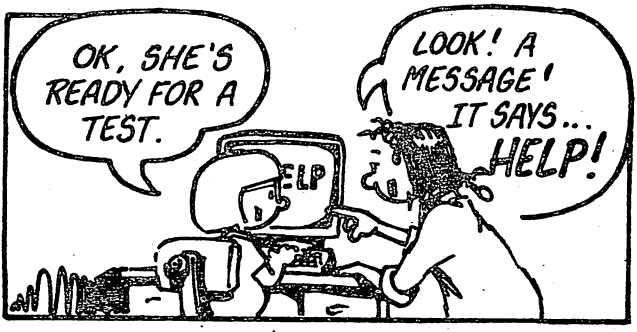
```
2760 A$ = A$ & CHR$(K)
2770 C = C + 1
2780 FOR J = 1 TO 30
2790 NEXT J
2800 GOTO 2720
2810 RETURN
```

We use the input routine in this game to get answers to our questions, so we are calling the input string A$ here. In the Variations section of the last chapter we suggested that you figure out how to let the player delete mistakes. In this game it is even more important to be able to delete, since the player must type in longer strings. Here is one way to do it--add these lines to the above subroutine:

```
2745 IF K = ASC("*") THEN 2775
2772 GOTO 2780
2775 C = C - 1
2776 CALL HCHAR(R,C,32)
```

We are using an asterisk (*) for the delete key. Whatever key you decide to use, you must notify the player what to press to delete. This means that you will have to insert a message and place it somewhere on the screen. Line 2775 subtracts 1 from C to move one step backward, and line 2776 draws a space at that column, thus erasing the last letter.

STORING THE STATES



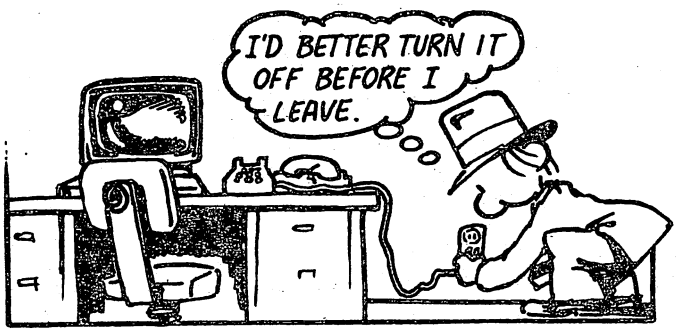
This subroutine is similar to the one in Chapter 1 in which we stored several fortunes into an array. Here we have ten state names in DATA statements. Notice that you can put more than one piece of string data on a DATA line, as long as you separate the strings with commas. We READ the ten states into an array we call S$ (S for state), and before we use the array we DIMension it--that is, we tell the computer to reserve ten spaces in its memory for our array.

```
2000 REM------LOAD STATES INTO ARRAY-------
2010 DATA ''OHIO'',''KANSAS'',
     ''NEW YORK'',''CALIFORNIA''
2020 DATA ''GEORGIA'',''NORTH DAKOTA'',
     ''NEVADA''
2030 DATA ''OREGON'',''FLORIDA'',
     ''NEW MEXICO''
```

```
2040 DIM S$(10)
2050 FOR J = 1 TO 10
2060    READ S$(J)
2070 NEXT J
2080 RETURN
```

The loop in lines 2050 to 2070 READs ten pieces of data (the names of the states) and stores them in the array. The name OHIO is then stored as S$(1), NEW YORK is S$(3), and NEW MEXICO is S$(10). Later on in the program we will use the array in the same way that we used the fortunes array. We'll pick a number at random and use that number to select a cell of the array. Then we'll take the name of the state that is stored in that cell and scramble it.

THE GAME



We are giving the player 6 turns to get to the finish line. Each turn consists of trying to unscramble the name of a state. If the answer is correct, the initial will move to the next plus sign. This means that we have to keep track of several things:

1. Where the initial is
2. If the initial has gotten to the finish line
3. How many turns the player has had

The GAME subroutine is fairly complex. It reads like a miniature program and uses several subroutines within it. Before we look at the BASIC, let's outline the steps we'll have to take in ordinary English:

Check to see if the player has won OR run out of turns.

If either of these conditions is true, then we give the appropriate message.

If neither is true we:

1. Clear the bottom portion of the screen
2. Pick a state name at random
3. Scramble the letters in it
4. Ask the player to unscramble it
5. Evaluate whether the answer is correct.

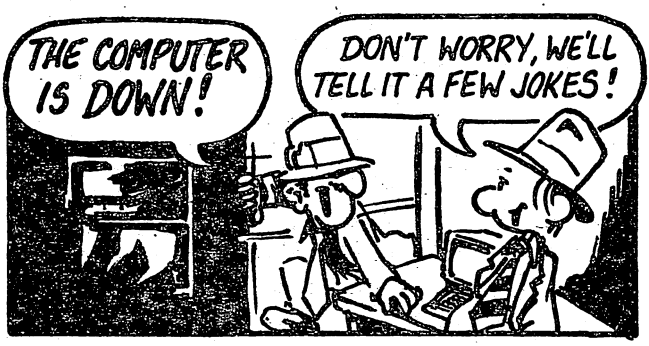
   a. If it IS correct, then we move the initial one step and check whether the player has now won.

b. If it is NOT correct then we tell the player the answer.

c. In EITHER case we add 1 to the number of turns and go back to the first step (Check to see if the player has won...)

Lines 3050 to 3340 perform these steps in BASIC. Lines 3010 to 3040 initialize four variables we need to use. U$(20) is an array we need in the SCRAMBLE section. T stands for turn number, and we set it to 1. P is the position (column) number of the initial, which starts at 5. O$ tells us if the game is over because the player won--we set it initially to "NO".

```
3000 REM --------THE GAME-------
3010 DIM U$(20)
3020 T = 1
3030 P = 5
3040 O$ = "NO"
3050 IF T > 6 THEN 3260
```



```
3055 IF O$ = ''YES'' THEN 3260
3060    GOSUB 2300
3070    GOSUB 3500
3080    R = 18
3090    C = 1
3100    M$ = ''TYPE YOUR ANSWER:''
3110    GOSUB 1500
3120    R = 19
3130    C = 3
3140    GOSUB 2700
3150    R = 21
3160    C = 1
3170    IF A$ = N$ THEN 3210
3180      M$=''SORRY, IT'S '' & N$
3190      GOSUB 1500
3200      GOTO 3240
3210      M$=''RIGHT!''
3220      GOSUB 1500
3230      GOSUB 3800
3240    T = T + 1
3250    GOTO 2300
```

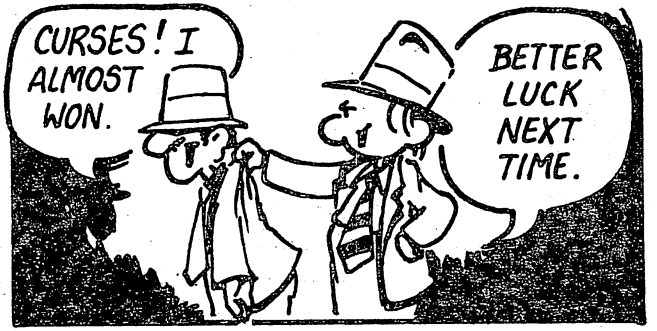
Lines 3050 and 3055 check for two conditions: if the turns are up or the player has won, the program skips past the above section, to line 3260. If the game is not over, we clear the screen (subroutine 2300), scramble the state and ask the question (3500), get the answer (2700) and evaluate it. If A$ (the answer) equals N$ (the name of the state) then the player is right and we output RIGHT!. If wrong, we say SORRY, IT'S (the correct name). Sooner or later the player will either win or

run out of turns, and then the subroutine will be finished:

```
3260 GOSUB 2300
3270 R = 15
3280 C = 1
3290 IF O$ = ''YES'' THEN 3330
3300    M$=''SORRY, YOUR TURNS ARE UP.''
3310    GOSUB 1500
3320    GOTO 3350
3330    M$=''CONGRATULATIONS, YOU WON!''
3340    GOSUB 1500
3350 RETURN
```

We clear the screen, decide if the player has won or lost, and output the appropriate message.

CLEARING THE BOTTOM OF THE SCREEN



This subroutine is virtually identical to the one in Chapter 5 which cleared the bottom of the screen. Here we need rows 15 to 21 erased:

```
2300 REM -----ERASE 15 TO 21-----
2310 FOR R = 15 TO 21
2320    FOR C = 1 TO 32
2330       CALL HCHAR(R,C,32)
2340    NEXT C
2350 NEXT R
2360 RETURN
```

SCRAMBLING THE STATE

We have an array of 10 states stored as S$(10). Whenever it comes time to ask a question, we want to pick one of the states and scramble the letters in it.

The first thing we need to know is the number of letters in the state (which we call N$). Then LEN (or length) function, remember, tells us how many characters are in a string. So LEN(N$) represents the number of characters in the state's name. We say characters and not just letters in the state because sometimes there are spaces also, as in NEW MEXICO.

We call the number of characters in the state L. To scramble them up, we go through a series of steps:

1. Pick a random integer (called X) between 1 and L.
2. Display character X on the screen.
3. Pick another random integer R between 1 and L.
4. Display character X next to the character already on the screen.

Repeat the above steps L times altogether, so there will be L characters displayed on the screen.

There is only one thing wrong with this plan. You know by now that if you tell the computer to pick a random integer between 1 and 7, and do it 7 times in a row, it might pick the same number each time. If your state was FLORIDA (which has 7 letters), and the computer picked the number 4 over and over again, you would get

RRRRRRR

displayed as the scrambled version of FLORIDA. That wouldn't be good.

Therefore, we need a way to tell the computer: keep picking random numbers between 1 and 7, but don't pick the same number twice. In other words, we have to keep track of which numbers were already used. The best way to do this is to create a new array, which we call U$ (for Used). Before we begin scrambling, we put the word "NO" in each cell of U$ to mean that we have not yet used that cell number. As each cell gets used (because the computer picked that number) we change what's stored in it to "YES". (We DIMensioned this array at the beginning of the GAME section.) Let's change our series of steps to show this addition:

1. Pick a number at random between 1 and L.
2. Check if that number has been used yet.
   2a. If it has been used, go back to step 1.
   2b. If it has not been used, then:
   Put "YES" in that cell of U$, and
   Display that character.

Repeat these steps L times.
Here is the BASIC code:

```
3500 REM ---SCRAMBLE A STATE-----
3510 X = INT(10 * RND) + 1
3520 N$ = S$(X)
3530 L = LEN(N$)
3540 FOR J = 1 TO L
3550    U$(J) = ''NO''
3560 NEXT J
3570 R = 15
3580 C = 1
3590 M$ = ''WHAT STATE IS THIS?''
3600 GOSUB 1500
3610 M$ = ''''
3620 FOR J = 1 TO L
3630    X = INT(L * RND) + 1
3640    IF U$(X) = ''YES'' THEN 3630
3650    U$(X) = ''YES''
3660    M$ = M$ & SEG$(N$,X,1)
3670 NEXT J
3680 R = 16
3690 C = 3
3700 GOSUB 1500
3710 RETURN
```

X is a random integer between 1 and 10. Why 10? Because we have 10 states. If X is 1 then S$(X) is OHIO, the first state in the DATA list. To make our code easier to read, we store S$(X) as N$. If N$ is OHIO, then L is 4, the number of characters in OHIO.

The loop in lines 3540 to 3560 initializes the U$ array. It puts the word "NO" in 4 cells of U$. In other words, U$(1), U$(2), U$(3), and U$(4) will all contain the word "NO" to start with. There are 4 letters in OHIO: all of them have NOT been used yet.

After we ask WHAT STATE IS THIS? we get to the part of the subroutine that actually scrambles the letters and displays them. We go through the FOR/NEXT loop in lines 3620 to 3670 L times. In our example, using OHIO, L equals 4 -- 4 letters in OHIO.

Each time through we pick a random number between 1 and 4 (that is, L). Then we check: if that cell of U$ contains "YES" because it's been used already we go back and pick another number. The computer will keep picking until it finds a random number corresponding to a cell of U$ which contains the word "NO". At that point it will skip line 3640 and proceed to do two things. First, put "YES" in that cell number so that the computer can't display the same letter again. Then, add that character to M$, so that when we finsh the loop M$ will be the scrambled state.

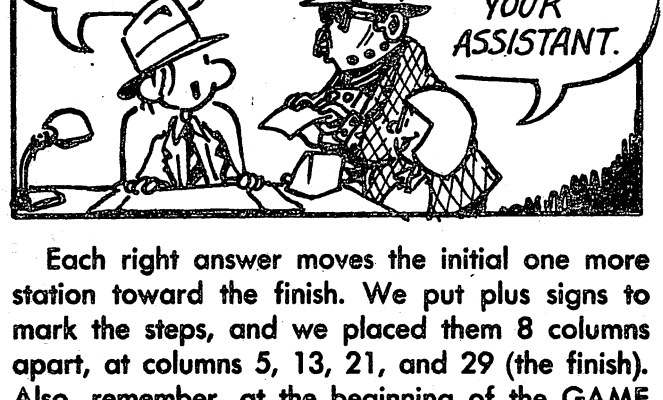
After the loop is finshed, we output M$ (the scrambled state) on row 16, column 3 of the screen.

It would be a good idea at this point for you to go over the code a few times and see how it does what our steps above it do. Convince yourself that every time the program goes to this subroutine it

picks a new state, calls it N$ and calls its length L, then goes around L times picking different random numbers each time and displaying the character of N$ corresponding to that number.

This subroutine does only this job. Refer back to the main GAME subroutine. Right before QUESTION we cleared the bottom of the screen. Right after it we get the answer from the player and evaluate it. If the answer is correct we have to take care of one more piece of business.

MOVING THE INITIAL



Each right answer moves the initial one more station toward the finish. We put plus signs to mark the steps, and we placed them 8 columns apart, at columns 5, 13, 21, and 29 (the finish). Also, remember, at the beginning of the GAME

subroutine we initialized P to 5 to keep track of where we put the initial to begin.

To move the initial we'll go through the same basic steps that we used in the last two games:

1. Erase the initial where it is now.
2. Draw it one column further over.

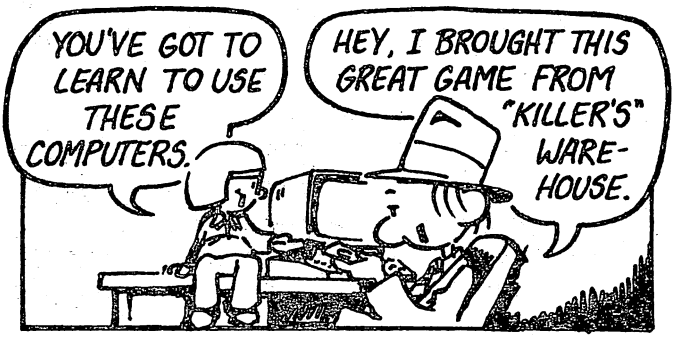
We repeat these steps 8 times to move the initial 8 columns. Then we do two more necessary things. We add 8 to P so that we know where the initial ends up after it's moved, and we check to see if P now equals 29. If it does, this means that the player has won, and we put the word "YES" into O$.

```
3800 REM --MOVE THE INITIAL---
3810 FOR M = P TO P+7
3820    CALL HCHAR(5,M,32)
3830    CALL HCAHR(5,M+1,ASC(I$))
3840    FOR J = 1 TO 10
3850    NEXT J
3860 NEXT M
3870 P = P + 8
3880 IF P = 29 THEN 3900
3890    GOTO 3910
3900    O$ = ''YES''
3910 RETURN
```

The animation here is just the same as that in the last chapter. We draw a space where the intitial is and then draw the intial one column further over, until the ititial has moved 8 columns.

O$ had been set to "NO" at the beginning of the game. Whenever P = 29 it will be set to "YES". Back in the GAME section, line 3055 will now be true, and the program will skip over the loop. When it gets to line 3290, since O$ does equal "YES" it will congratulate the player and end.

FUN AND GAMES

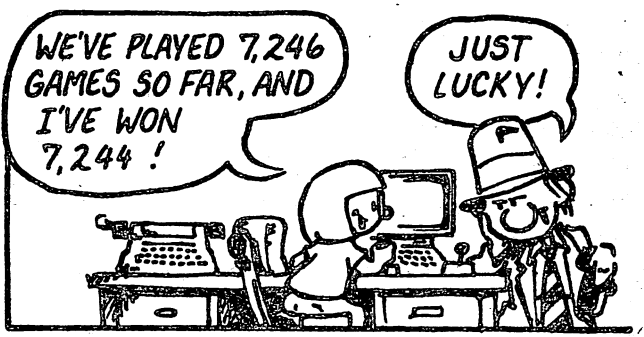


Throughout this book we have covered many aspects of programming. We have used new concepts like the random function and animation. We have explained new commands, functions, and subprograms. Of equal importance, we have tried to introduce good programming practices of structured programming.

In the first four chapters we suggested several possible variations. At this point the variations are all up to you. Almost any of those suggested for

the other games you could also use for the game in this chapter. More important, though, is for you to think up your own variations. Pick any part of any program and you can probably think of ways to improve it, to make the screen format look better or to make the game more fun. Create your own coding routines. Think up more complicated uses for the random function. Do fancier animating. Invent totally new games.

We have covered many of the built-in Texas Instrument's functions and subprograms. There are several more, however, that you might use to enhance your programs. The CALL SOUND subprogram, for instance, can add music to programs, and CALL COLOR can color any character on the screen, or change the entire screen background. Another subprogram, CALL CHAR, lets you define your own characters to look any way you want. This is a useful graphic feature—you can use it to make your own shapes for animation.



Programming is a combination of using what you already know and using your imagination. We have tried here to supply you with some useful concepts and techniques for programming games. However, there is always much more to learn for any programmer, no matter how experienced. If you want to get better at programming, you will keep reading and studying to learn more about this skill. You already know enough, though, to create imaginative games. Hopefully you will go on learning as you program. The most important thing you can do is to have fun programming!

Index

Index to chapter references for new concepts and terms used in this book. Reserved words in BASIC are capitalized.

SEG$ function	returns a specified string from within a given string. Example: SEG$("HAPPY",4,2) returns 2 characters from HAPPY, starting at the 4th character, or PY.	2
STR$ function	makes a string from a specified numerical variable	4
subroutines	program sections or blocks as called by the GOSUB command. Subroutines must end with the RETURN command.	1-5
VAL function	makes a numerical variable from a specified string, if the string is composed of numerals.	5

YOU, SPEED WALKER, AND YOUR TI-99 Series CAN DO SOME PRETTY AMAZING THINGS!

If you already know some beginning BASIC then you're ready to start programming your TI to play terrific games.

With simple, easy-to-follow directions your clue-finding friend SPEED WALKER will take you step-by-step through programs of mystery and adventure. Programs that you can expand and improve with what you learn in this book.

The rest is up to you and your imagination.

GET READY...GET SET...START PROGRAMMING!

ISBN 0-523-42247-4