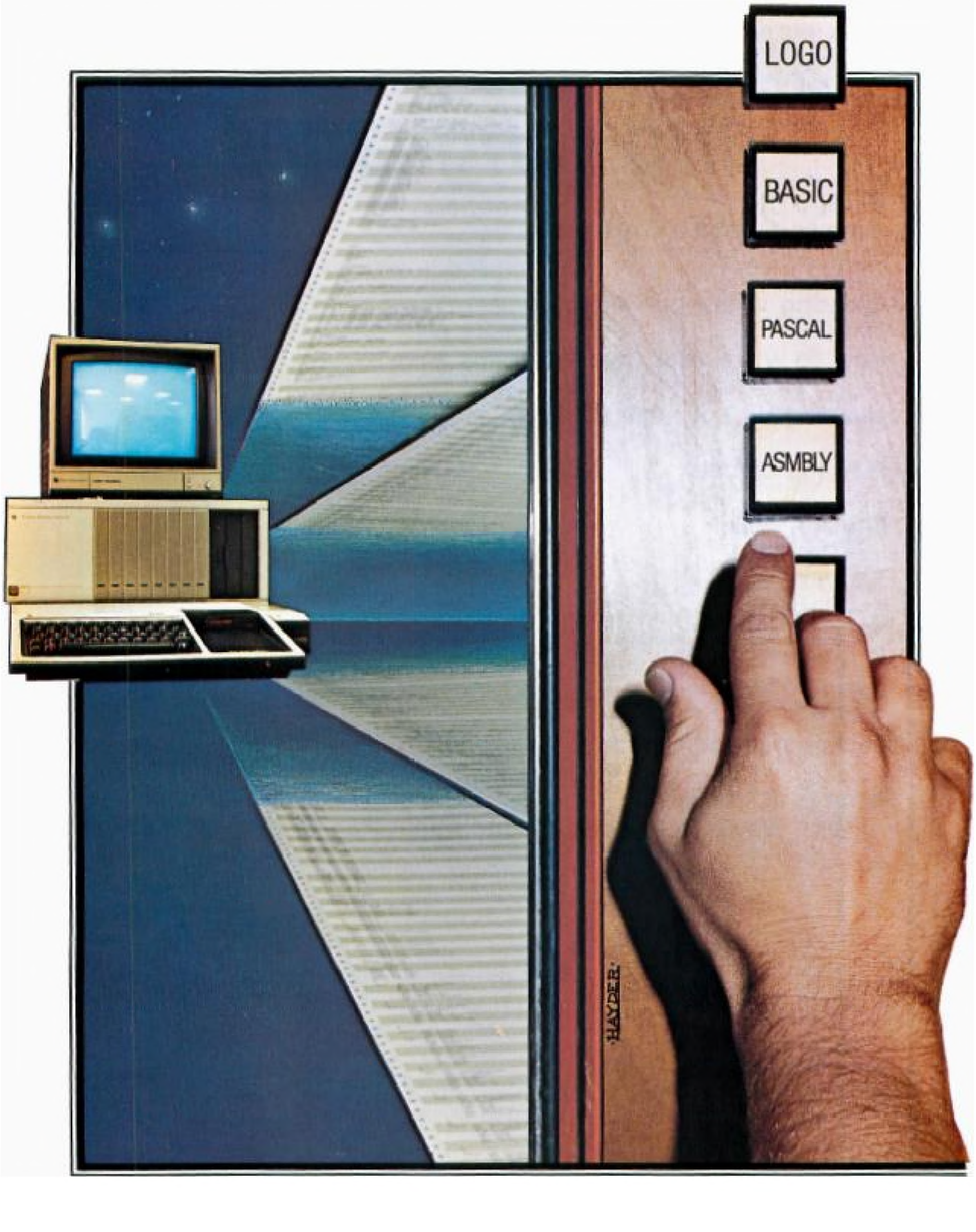
2
Programming Techniques and Languages



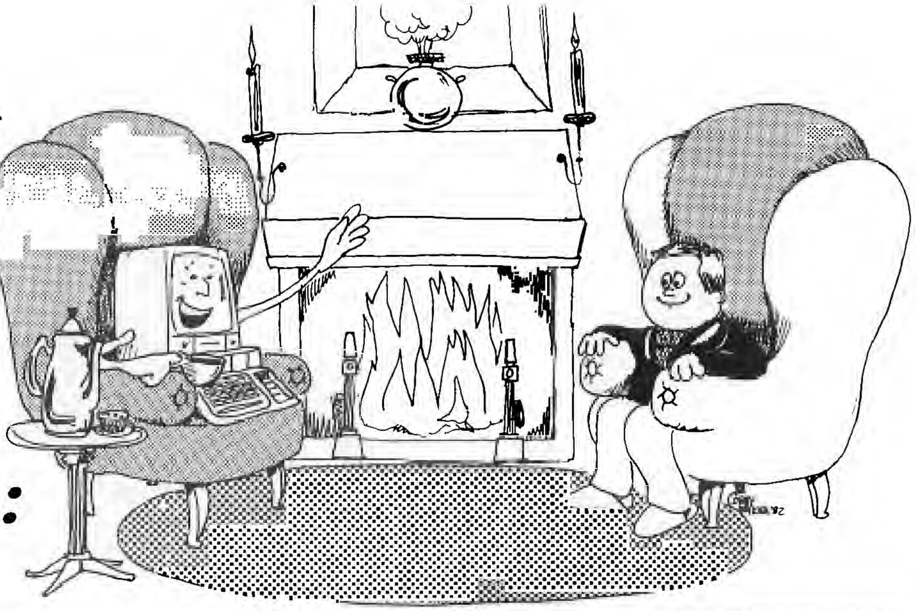
2

Programming Techniques and Languages

How to talk to a computer.

Languages for the Home Computer

Chatting with Your Micro:



Home Computers are indeed wonderful machines. They have been carefully designed to allow beginners to do meaningful tasks, act as educational tools, and provide hours of inexpensive family entertainment.

All of this is made possible by the availability of "user-friendly" software—Command Cartridges, cassette tapes, and floppy disks that have been pre-recorded with programming instructions the computer can understand and carry out.

Users of this software need not concern themselves with how this programming was actually produced—unless, of course, they get smitten with that highly contagious human germ known as "curiosity," and want to understand something about the process.

"Programming" the Home Computer is *not* some mysterious rite that is meant to be practiced by a select few in secrecy. Rather, it is simply a means of communicating with a machine in a language that *both* humans and human-designed electronic circuits can understand—nothing more elaborate than basic, down-to-earth communication.

Languages, whether human-to-human or human-to-machine, differ widely in their complexity. Depending on the language, varying amounts of memorization and practice are required before a "speaker" can communicate effectively. The levels of computer language complexity run the gamut from conversational English phrases, to the switching on and off of electric current that the machine "understands" and transforms into various actions.

Before a user can begin communicating with a computer, however, one of three conditions must be met: (1) The user must be able to communicate in the computer's language; (2) the computer must be able to communicate in the user's language (i.e., English, German, Spanish, etc.); or (3) some common intermediate language must be established, understood, and used by *both* parties. By definition, the closer this intermediate language is to the machine's natural "electrical" language, the *lower* its level. And conversely, the closer to the human's language, the *higher* the level.

Machine Language

First, let's take a look at the lowest level of common intermediate language—referred to as "machine language." Since electricity can either be on or off—one of *two* possible conditions—machine language can only be constructed from *two* "words." This binary language is often expressed by humans with the two digits 1 and 0, with 1 representing the "on" state (presence of electricity), and 0 representing the "off" state (absence of electricity). Absolutely *shocking* in its simplicity, isn't it?

Figure 1.

"MACHINE LANGUAGE SAMPLE"

```
0 0 0 0   0 0 1 0   0 0 0 0   0 0 0 0
0 0 0 0   0 0 0 1   0 0 0 0   0 0 0 0
0 0 0 0   0 0 1 0   0 0 0 0   0 0 0 1
0 0 0 0   0 0 0 0   0 1 0 1   0 1 0 0
0 0 0 0   0 1 0 0   0 0 1 0   0 0 0 0
0 0 0 0   0 0 0 0   1 0 0 0   1 1 0 0
0 0 0 0   0 0 1 0   0 0 0 0   0 0 0 0
0 0 0 0   0 0 0 1   0 0 1 0   0 0 0 0
0 0 0 0   0 0 1 0   0 0 0 0   0 0 0 1
0 0 0 0   0 0 0 0   0 0 1 1   0 1 0 0
0 0 0 0   0 1 0 0   0 0 1 0   0 0 0 0
0 0 0 0   0 0 0 0   1 0 1 0   0 0 0 0
```

Figure 1 represents six machine language "sentences." It's not easy for a human to understand, is it? Yet when communicating this way, more explicit control of the machine is possible, because there can be nothing "lost in the translation."

TMS9900 Assembly Language

Human difficulty in communicating in a binary language led to the next step in the evolution of higher-level languages—an easier-to-remember ("mnemonic") way of expressing these binary "sentences." This was done by assigning combinations of alphabetic letters to represent operations formerly only expressable by binary sequences, and assigning a *full range* of characters (including numbers) to represent the things actually "operated" on.

This easier, alphanumeric way of communicating is called *Assembly Language* because these newly created scores of symbols must eventually be translated back (*assembled*) to their binary equivalents for the machine to carry them out.

Figure 2

```
          REF     VMBW,INPUT
LINE 1    TEXT    'HI, I AM THE TI-99/4A'
LINE 2    TEXT    'HOME COMPUTER '
LINE 3    TEXT    'WHAT'S YOUR NAME?'
BUFFER    BSS     32
LINE 4    TEXT    'NICE TO MEET YOU, '
GREET     LI      R0,0
          LI      R1,LINE1
          LI      R2,32
          BLWP    @VMBW
          LI      R0,64
          LI      R1,LINE2
          BLWP    @VMBW
          LI      R0,128
          LI      R1,LINE3
          BLWP    @VMBW
          LI      R0,BUFFER
          BLWP    @INPUT
          LI      R0,256
          LI      R1,LINE4
          BLWP    @VMBW
          LI      R0,288
          LI      R1,BUFFER
          BLWP    @VMBW
          END     GREET
```

Note: Keep in mind that the use of this and other sample program segments that follow are for comparison purposes *only*, and do *not* indicate the true power of any of the languages. Note also that the reference to a routine called INPUT doesn't imply the existence of that routine (as this is only an example).

In Figure 2 we are showing you part of an Assembly Language program that causes the computer to print several English language messages on the screen, and allows it to accept and acknowledge human response via the keyboard. The screen dialog goes like this:

HI, I AM THE TI-99/4A
HOME COMPUTER
WHAT'S YOUR NAME?

You type in your name

NICE TO MEET YOU,

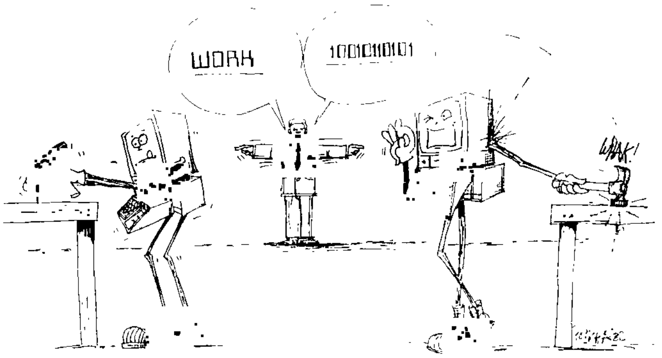
Your name appears here

Observe in Figure 2, the left to right sequence of symbols that must be followed if the program is to be assembled correctly. As an example of the proverbial "before and after"—Assembly Language lines that have been assembled back to binary—take a look at the last seven lines of symbols in Figure 2. The machine language that results from the assembling of these symbols appears as the entire sequence of left-to-right binary sentences shown in Figure 1.

Higher-Level Building Blocks

Although some very important programming is still done at the Assembly Language level, the majority of programs written are in higher-level languages. These languages are closer to human languages such as English than to machine language. To generate these higher-level languages, we must take ordered groups of Assembly Language statements and equate *each group* with a *single word* of the new, higher-level language we are generating. Each word of this new language is much more powerful than any single Assembly Language symbol: With *one* new higher-level word, we can make the computer do *several* things. This is a powerful technique, indeed, and has been the basis for all computer languages that have evolved.

For the computer to understand one of these new English-like languages, the language must first be translated into machine language.



Compiling & Interpreting Down

When translation of *all* the high-level language statements in a program takes place *before* the computer acts on the statements, the language is said to be *compiled*. The binary sequences that result from this compilation are then saved and later used directly any number of times.

On the other hand, the language is said to be *interpreted* when the computer acts on *each* statement immediately after that statement's translation. Therefore, *every* time an interpreted language program is "run" (all statements followed step-by-step to completion), the program *must* be re-translated. Because of this basic difference in translation technique responsible compiled language programs are faster than interpreted ones.

This is not to say that interpreted languages do not have compensating advantages. Ease of use is a case in point:

Additional Terms You'll Want to Know

Command Cartridge—A plug in plastic cartridge from Texas Instruments with integrated circuits that contain a computer program (software).

floppy disk—A mass storage device using a flexible mylar disk to record information. It is a more sophisticated alternative (quick random access) to cassette tape storage (sequential access).

Home Computer—The Texas Instruments TI-99/4A console with either home television or TI Color Monitor.

Integrated circuit(IC)—Integrated circuits have many individual components packed together or integrated in a small area. The circuits of the computer are fabricated on silicon chips. A chip is typically about 1/4 inch on a side. Today's chips are so sophisticated that the basic components of an entire computer can be fabricated on a single chip.

mnemonic—Assisting or intended to assist the memory.

screen—The home television or TI monitor to which the computer outputs information like numbers/letters/graphs, etc.

Speech Synthesizer—A peripheral device built by Texas Instruments for use with the Home Computer and used to reproduce the human voice electronically.

TMS9900—A very sophisticated integrated circuit (called a "microprocessor") containing all the most basic components of an entire computer. Designed and built by Texas Instruments, it is the heart of the Home Computer.

Just as soon as we finish writing the last program statement in an interpreted language, we can *immediately* run the program—without having to go through an additional intermediary step such as compilation. The translation in an interpreted language is therefore invisible or hidden from us.

Furthermore, in many interpreted languages such as the BASIC language that comes built into your Home Computer, statement misuse or errors of spelling in language vocabulary are checked for *right at the time the statements are typed in*. Appropriate error messages (if needed) will appear on the screen; the person doing the programming can then make immediate corrections.

TI BASIC

Because TI BASIC is a high-level and interpreted language, it is easy to learn and use. The sample program segment that follows (Figure 3) will cause the computer to carry on a dialog similar to the one previously shown in Figure 2. Notice how much easier the TI BASIC version is to understand.

Figure 3

```
100   PRINT "HI, I AM THE TI-99/4A"
110   PRINT "HOME COMPUTER"
120   PRINT "WHAT IS YOUR NAME?"
130   INPUT NAME$
140   CALL CLEAR
150   PRINT "NICE TO MEET YOU,"
160   PRINT NAME$
170   END
```

TI Extended BASIC

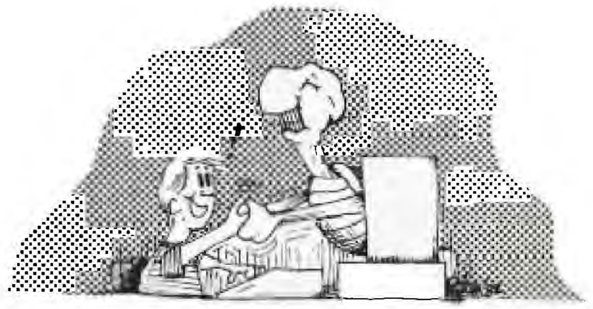
TI Extended BASIC, one of the higher-level languages that you can add to your Home Computer by plugging in the separate Command Cartridge for the language is similar to the regular built-in BASIC. It gives you everything that the regular BASIC does *plus* many special additional features such as arcade-style animated graphics (known as "sprites"), commands to control the Speech Synthesizer, as well as more precise control of on-screen text messages (demonstrated in Figure 4).

Figure 4

```
100    DISPLAY AT(2,1):"HI, I'M THE TI-99/4 HOME
       COMPUTER"
110    DISPLAY AT(6,1):"WHAT IS YOUR NAME?"
120    ACCEPT AT(8,5)VALIDATE
       (UALPHA)SIZE(15):NAME$
130    CALL CLEAR : : PRINT "NICE TO MEET YOU,
        ";NAME$
140    END
```

TI LOGO

Another interpreted language very popular with children and educators is TI's unique implementation of LOGO. It contains the previously mentioned sprites and features "Turtle Graphics." These line drawings generated by a "pen" attached to a simulated "turtle" object (that is moved about the screen with only simple heading and distance commands) are both enchanting and instructive—contributing to the wonder of discovery that children experience with the computer. See Figure 5 for a sample TI LOGO program (known as a *procedure*).



THE "USER-FRIENDLY" LOGO TURTLE

Figure 5

```
TO GREET
CLEARSCREEN
PRINT [HI, I AM THE TI-99/4A]
PRINT [HOME COMPUTER]
PRINT [WHAT IS YOUR NAME?]
CALL READLINE "N
PRINT "HELLO,
PRINT :N
END
```

TI PILOT

Whereas LOGO is a high-level language with great depth—i.e., the built-in vocabulary can be "customized" by the user—TI PILOT, another high-level language, is much more abbreviated. With a fixed vocabulary of only 15 major commands, the interpreted TI PILOT language still allows access to sprites, color graphics, and sound. Each command is represented by one or two letters followed by a colon. The program segment in Figure 6 illustrates a dialog in PILOT.

Figure 6

```
D:  R$(15)
T:  HI, I AM THE TI-99/4A
T:  HOME COMPUTER
T:  WHAT IS YOUR NAME?
A:  R$
T:  HI THERE, $R$
E:
```

With PILOT, you can develop effective educational programs even if you've had little or no programming experience. For this reason, PILOT is favored by educators as a language highly suitable for producing computer-assisted instruction (CAI) courseware.

UCSD Pascal

Currently, the only high-level *compiled* language available for the Home Computer is University of California at San Diego (UCSD) Pascal. This version of Pascal includes functions for accessing all the special Home Computer features. The language is more appropriate for professional programmers or users who wish to delve into more sophisticated programming. Although not as difficult as Assembly Language to master, UCSD Pascal is, nevertheless, much more difficult than other high-level languages on the Home Computer.

This compiled language also happens to be highly *structured*—i.e., it restricts programs to modular organization according to sets of specific construction rules known as *syntax*. Because it is both compiled and structured, programs written in UCSD Pascal are faster and easier to modify than most other high-level languages. This makes it a suitable language for large business and scientific programs. See Figure 7 for a very simple (almost trite!) example of our now-familiar man-machine dialog as written in Pascal.

Figure 7

```
  PROGRAM GREET;
  VAR        NAME: STRING;

BEGIN
  WRITE(OUTPUT,'HI,I AM THE TI-99/4A ');
  WRITELN(OUTPUT,'HOME COMPUTER.');
  WRITELN(OUTPUT,'WHAT IS YOUR NAME?');
  READLN(INPUT,NAME);
  WRITE(OUTPUT,'NICE TO MEET YOU, ');
  WRITELN(OUTPUT,NAME');
END.
```

ASPIC

Early in this article we implied that higher-level languages are constructed from other languages. This means that *you* have the opportunity to design your own personal languages for communicating with your Home Computer. You can do this by defining both the syntax and each word of your *new* language in terms of the statements and commands of an *existing* Home Computer language. The new ASPIC language is a case in point. Constructed from TI BASIC, ASPIC was created to simplify a child's manipulation of color graphics on the Home Computer. Figure 8 shows an example of a typical program segment. [See ASPIC article in this book for a complete discussion of this new language—Ed.]

Figure 8

```
10      CLEAR
20      MAKE +
30      MAKE X
40      COLOR SCREEN RED
50      COLOR + BLACK
60      COLOR X GRAY
70      LET R1 = 5
80      LET C1 = 5
90      LET R2 = 21
100     LET C2 = 13
110     REPEAT 9
120     DRAW + IN ROW#R1 COL#C1
130     DRAW X IN ROW#R2 COL#C2
140     LET C1 = C1 + 1
150     LET R2 = R2 - 2
160     END
```

TI FORTH

If you want to modify a language to fit your own particular needs, TI FORTH may be for you. FORTH is much like LOGO, in that the basic language implementation consists of a small number of built-in primitives (called *definitions*) from which you may construct new definitions. The primitive definitions and the new definitions which you create are entries in FORTH's *dictionary*. Once you've entered your new definitions in the dictionary, however, they're a permanent part of *your* FORTH implementation. Programmers who feel the need to simplify their work with custom modifications—software developers, for instance—will be most interested in FORTH.

Now it's up to *you* . . . Go ahead and strike up a conversation with your new-found electronic friend. Who knows? New respect for and long-lasting TIes with your Home Computer may be the result.

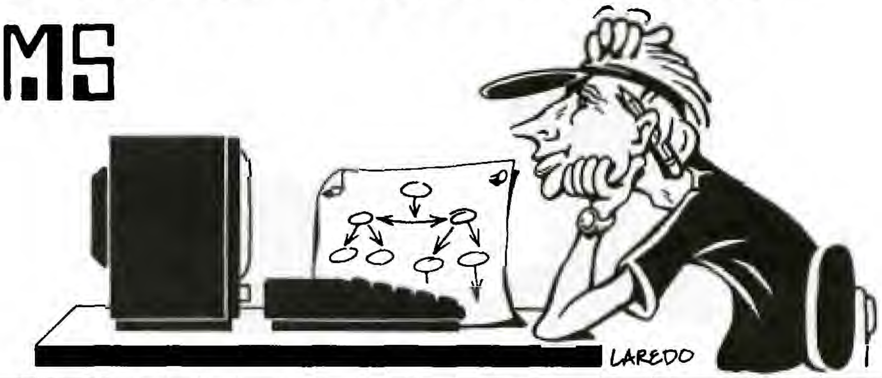
COMPARISON OF LANGUAGES FOR THE TI-99/4A HOME COMPUTER

	Minimum Relative Cost of System Components Needed[1]		Ease of Use (1-10)[4]	Execution Speed (1-10)[4]	Color Graphics Supported	Sprites Supported	High Speed Turtle Graphics Supported	Speech Supported (with additional Synthesizer peripherals)	Music Supported
	To Run Programs	To Write Programs							
ASPIC	1.0	1.0	9	1	yes	no	no	no	no
9900 Assembly Language	1.3 Note 2	4.7 Note 3 1.3 Note 3A	1	10	yes	yes	no[5]	yes	yes
TI BASIC	1.0	1.0	7	3	yes	no	no	no[6]	yes
TI Extended	1.3	1.3	6	4	yes	yes	no	yes	yes
TI LOGO	2.9 Note 8	2.9 Note 8	8	4	yes	yes	yes	yes	yes[7]
UCSD Pascal	3.3 Note 8	7.3	4	7	yes	yes	no[5]	yes	yes[7]
TI PILOT	5.4	7.5	10	5	yes	yes	no	yes	yes[7]
TI FORTH	1.3	4.7	4	7	yes	yes	no	yes	yes

1. Relative cost ratio with price of a TI-99/4A taken as unity.
2. Cost for running assembled program in Mini Memory cartridge.
3. Cost for writing Assembly Language programs using *Editor/Assembler.*
3A. Cost for writing *very* small Assembly Language sub-routines & programs using Mini Memory cartridge and separate *Editor/Asembler Manual*.
4. Number 1 represents worst case; 10 represents best case. Please note these values are subjective and are *not* based on any laboratory test data.
5. With the proper background and experience, the user can use this language to write his own turtle graphics support routines.
6. If the *Speech Editor* cartridge or *Terminal Emulator II* are added to the system, speech can be supported by the language.
7. The new TI LOGO II, TI PILOT, and USCD Pascal have enhanced music commands.
8. Minimum configuration based on cassette storage.

HOW TO WRITE YOUR OWN PROGRAMS

Using Flowcharts To Outline a Solution.



Sitting down in front of your TI-99/4A and running packaged software may stimulate your desire to try some programming of your own. If you have taken courses in computer programming or have had some practical on-the-job training, you can probably type some lines and have the computer do what you want. However, if you haven't had this experience, you may soon find the frustration of not knowing where to start too much to bear. Well, take heart! Here we present some basic information on how you can begin programming on your own.

A Framework for Writing Programs

Computer programming is an exercise in reasoning and logic. Before programmers develop software to do specific jobs, they plan their attack on the individual elements that are inherent to those jobs or problems. It is helpful to have in mind a general framework for solving each problem.

This general framework could take a number of different forms. Most will, however, contain similar steps. These steps can be described as follows.

1. Define the Problem

Initially, it is necessary to have a good understanding of exactly what you want the program to do. If it is possible, try to express the problem in a simple thought or sentence stating the intended outcome of the programming effort. Defining the problem in this manner may not only save you time, but may also help focus your efforts.

2. Outline the Solution

This step is the primary purpose of this article. We'll get back to examine this step in more detail later.

3. Select the Algorithm

Many problems requiring a computer for solution depend on certain mathematical algorithms that are required in the calculation of the desired solution. For those who are puzzled by the word "algorithm," mathematicians and math teachers use this word to refer to the specific method of solving a certain kind of mathematical problem. For example, you may have been taught to subtract whole numbers by placing the larger number on top, the smaller on the bottom, and to borrow when necessary. This is but one possible algorithm for subtraction. In general, you can either locate those algorithms that are necessary from published sources, or design your own. In either instance, the simpler the algorithm, the better.

4. Writing the Program

Many people believe the writing or "coding" of the program is what computer programming is all about. Actually, this is just *one* in a series of steps. Prior planning (as detailed in steps 1-3, above) is absolutely essential before the actual writing of the program can begin. And inherent in the writing of the program must be a reasonable understanding of the computer language you will be using.

5. Debugging

Once you have typed the program into the computer, it is necessary to run it to determine if and what difficulties exist. You will seldom write an error-free program on the first draft. Trying to locate and correct those "bugs" can be frustrating. This is where some of the 99/4A's built-in features help tremendously.

6. Validating the Program

In this step, you intentionally try to locate situations in which the program yields inaccurate or undefined solutions.

7. Documentation

It is a good idea to record the characteristics of the program such as its intent, algorithms, and specifications. Some day, in the future, when you decide to modify this program you will be very happy that this documentation exists. Incidentally, when buying a program, the author's documentation (or lack thereof) can often be a good indication of the quality of the program.

Outlining the Solution

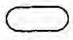
The development of an adequate outline (Step 1 above) is the most critical step in writing a program. Many of us dabblers in the art of programming seem to fail in developing an adequate outline. My intention here is to demonstrate to you some elementary outlining techniques—in the hopes that we, the dabblers, may be able to improve our lot in the somewhat puzzling world of bits, bytes, and bugs.

Flowcharting

There are a number of methods available for outlining a solution to a problem. Of those used in computer programming, flowcharting is one of the simplest and easiest.

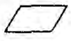
To introduce you to the flowcharting method, let's first look at some of the symbols used.

1. START and END symbol



This symbol is used to indicate both the beginning and the end of a program.

2. INPUT – OUTPUT symbol



The input-output symbol is used to indicate where the user of the program will need to supply a piece of data or where a calculation will be printed out for the user.

3. COMPUTATION or ASSIGNMENT symbol

▭

This symbol is used to indicate where computations or assignments of values to variables will occur.

4. DECISION symbol

◇

This decision symbol is used to indicate where a "yes" or "no" or "true" or "false" decision point is located.

5. STOP symbol

□

This symbol is used in some programs to indicate a termination point if this point is different from the end point of the program.

There are other symbols that can be used according to your needs. Also, remember that no rules exist to stop you from developing your own symbols.

Toward a Workable Technique

The outline strategy that works best for me is to start off simple and then increase the complexity of my outline until it does what I want it to do. My approach includes: (1) writing a sentence that defines the problem I want to solve, (2) preparing an informal outline, (3) developing a more complex flowchart, and then (4) writing the program. To demonstrate how this approach leads you to developing a better program, let's take a look at some examples.

EXAMPLE 1

For our first example, let's write a program that will add two numbers together and print their sum. We will design the program so that we may input two numbers from the console. This is called an *interactive* program, in that the user must input the values to be assigned to the variables. Following the approach presented, we first define the problem:

Step 1. Definition of the Problem:

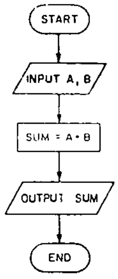
The program will take two numbers being input from the console, add them together and print the sum.

Step 2. Informal Outline:

1. Start
2. Input two numbers, A and B
3. Add A and B
4. Output the sum of the numbers

Step 3. Flowcharting:

Using the flowcharting symbols, the solution is further developed:



Explanation of the flowchart

The "flow" is evident in the continous line running from the initial start symbol to the final end symbol. The input symbol shows that the two values are requested, with the first input value being assigned to the variable A and the second to B. The addition of the two numbers and the assignment of their sum to a variable occurs inside the computation symbol. The value of the sum is then output, and the program ends. The algorithm necessary for the solution is shown.

Now that the problem has been outlined, we proceed to write or code the program.

Step 4. Coding:

```
100 REM **ADDITION PROGRAM**
110 INPUT A,B
120 LET S=A+B
130 PRINT S
140 END
```

Explanation of the program.

The program shows how the original intent is followed.

Line 100 contains a REM statement allowing us a means of identifying the program.

Line 110 allows the user to type in the two numbers to be added.

Line 120 assigns the value of A plus B to the variable S.

Line 130 prints the value of S.

Line 140 ends the program.

Since my primary intent here is to explain how an outline is developed and used, I will not explain the TI BASIC command statements but assume that readers of this article have already read most of the TI *Beginner's BASIC*, the book that came with their computer.

EXAMPLE 2

For a more complex example, let's develop a program that will select and print the larger of two input values.

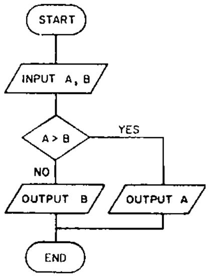
Step 1. Defining the Problem.

Given two numbers, the program will select the larger of the two and print it.

Step 2. Informal Outline:

1. Start.
2. Input two numbers, A and B from the console.
3. Compare number A with B. If A is larger than B, print A. If A is not larger than B, print B.
4. End.

Step 3. Flowcharting:



Explanation of the flowchart.

The flowchart begins with the start symbol. The two values are then input. In the decision box, a comparison of the value A with B takes place. If the statement A>B is true, the computer is instructed to bypass the output B box, and output the value of A. If the statement is false, the computer continues down the chart to output the value assigned to B. The program then ends.

Step 4. Coding:

```
100 REM **PRINTS LARGER OF TWO NUMBERS**
110 INPUT A,B
120 If A>B THEN 150
130 PRINT B
140 GOTO 160
150 PRINT A
160 END
```

Explanation of the program.

Line 100 is a REM statement used to identify the program.

Line 110 allows the user to input the two numbers to be compared.

Line 120 is used to compare the two numbers. If the statement *A is greater (>) than B* is true, the computer is *then* instructed to go to line number 150 to print A. If the statement in line 120 is false the computer continues to the next line.

Line 130 prints the value of B, as it must be the larger.

Line 140 is used to direct the computer to go to line 160. Without this line, the computer would print the value of B, then the value of A. This is, of course, not what we wanted.

One difficulty exists with this program. If A and B are equal, the program will not be able to distinguish the two. (If this arises, B will be printed.) This difficulty could be corrected for this situation by allowing another step where value A and value B could be displayed.

EXAMPLE 3

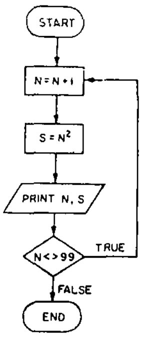
Let's take a look at one more simple example. This time we'll try writing all the squares of the integers between 1 and 99, inclusive of the two boundaries.

Step 1. Defining the Problem:
The program will make a list of all the squares of the integers between 1 and 99, inclusive.

Step 2. Informal Outline:

1. Start.
2. Let N be a variable whose initial value is 1.
3. Compute the value of N^2, and let the result be the value of S.
4. Print N and S on one line of the screen.
5. If the value of N is 99, then end the program. Otherwise go to step 6.
6. Add 1 to the value of N and then go back to step 3.

Step 3. Flowcharting:



Explanation of the flow chart.

After the program starts, the variable N is increased by 1. As the TI BASIC will automatically set the initial value of N to zero, using the statement N=N+1 will set the first value of N to 1. Next, the square is calculated. Both the integer and its square are then printed. The next step checks to see if N is equal to the upper boundary of 99. If N is equal to 99, the computer is instructed to end the program. If N is not equal to 99, the program loops back to add 1 to the value of N and continues.

Step 4. Coding:

```
100 REM **SQUARES**
110 LET N=N+1
120 LET S=N^2
130 PRINT N,S
140 IF N<99 THEN 110
150 END
```

Explanation of the program.

Line 100 is the REM statement.

Line 110 adds 1 to the variable N.

Line 120 computes the square.

Line 130 prints the integer N and its square S.

Line 140 determines if the value of N is 99. If N is equal to 99, the computer goes to line 150 and ends the program. If N is not equal to 99, the computer returns to line 110.

Line 150 ends the program.

Now that we have seen the use of the outlining technique in some rather elementary program examples, let's get serious and try something more challenging.

EXAMPLE 4

Let's try writing a program to test our recall of a series of digits. With each correct matching of a digit, we'll instruct the computer to add another digit to the series.

Step 1. Defining the problem:
The program will display a series of digits of increasing length and ask the user to recall the correct order of the digits.

(It might be helpful to place some limits on the program to further qualify what we want it to do. This can be done in the informal outline.)

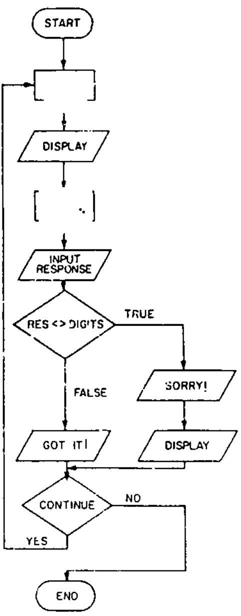
Step 2. Informal Outline:

1. Start.
2. Have the computer select a random digit.
3. Display the series of digits for a short time.
4. Clear the screen.
5. Ask for a response from the user.
6. Compare the response to the series of digits.

7A. If the response is correct, congratulate and ask if the user wants to continue.
7B. If the response is incorrect, show the correct series of digits and ask if the user wants to continue.

8. If the user wants to continue, have the computer select another digit and add it to the end of the previous series.
9. If the user does not want to continue, end the program.

Step 3. Flowcharting:



Explanation of the flowchart.

After the program starts, a random digit is selected. The series of digits is displayed, and the screen is cleared. The user is then asked to respond. If the response is correct, the computer offers congratulations with a GOT IT! message, then asks if the user wants to continue. If the response is incorrect, the computer says SORRY!, then displays the correct response. The user is then asked if he wants to continue. If the answer is yes, the computer loops back to the random digit selection box, tacks on an extra digit to the string, and continues. If the answer is no, the program ends.

Step 4. Coding:

In coding the program, we'll use the RANDOMIZE and RND statments from TI BASIC to get a better selection of digits. Take the number returned by RND, multiply it by 10, then take the integer portion: The algorithm is INT(RND*10). In order to display, compare and add digits to the series, we'll translate them into a string using the STR$ function.

Explanation of the program.

Lines 100-170 are REM statements.

Line 180 is the RANDOMIZE statement.

Line 190 begins the selection of the random digit. With this statement, the computer will display a series of digits starting with a single digit and extending to an upper limit of 25 digits maximum.

Line 200 is the algorithm for selecting the random digit and assigning its value to the variable A.

Line 210 translates the digit selected to a numeric string. The line will also function in adding each digit selected to the end of the previous series of digits.

Line 220 clears the screen.

Lines 230-250 present the series of digits and tell you how much time you are allowed to study the series.

Lines 260-270 time the digits being displayed. Going through the FOR. . .NEXT loop takes about five seconds.

Line 280 clears the screen.

Line 290 directs the computer to jump to line 320. The line is intended to get us out of the FOR I. . .NEXT I loop without disrupting it.

Line 300 continues the FOR I. . .NEXT I loop.

Line 310 ends the program.

Lines 320-330 prompt the user to respond.

Line 340 compares the response to the series of digits. If the response is incorrect, the THEN condition directs the computer to line 380, which is the SORRY! comment. If the response is correct, the computer goes to the next line (line 350).

Line 350 clears the screen.

Line 360 congratulates the user.

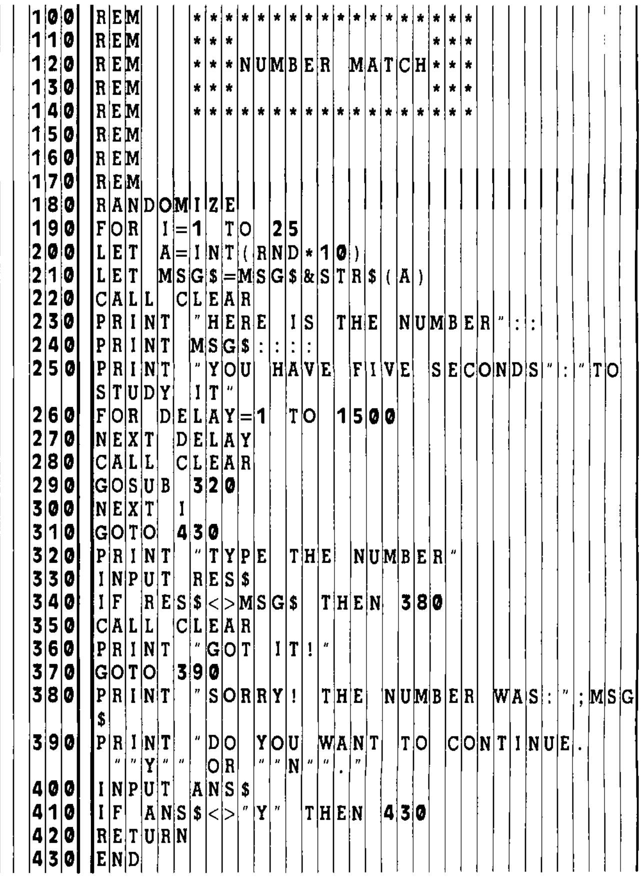
Line 370 directs the computer to go to line 390, bypassing the SORRY! comment.

Line 390 asks if the user wants to continue.

Lines 400-410 check to see if the user is interested in continuing.

Line 420 returns the computer back into the FOR I. . .NEXT I loop.

Line 430 ends the program.



```
100 REM    ******************
110 REM    ***            ***
120 REM    ***NUMBER MATCH***
130 REM    ***            ***
140 REM    ******************
150 REM
160 REM
170 REM
180 RANDOMIZE
190 FOR I=1 TO 25
200 LET A=INT(RND*10)
210 LET MSG$=MSG$&STR$(A)
220 CALL CLEAR
230 PRINT "HERE IS THE NUMBER"::
240 PRINT MSG$::::
250 PRINT "YOU HAVE FIVE SECONDS":"TO
    STUDY IT"
260 FOR DELAY=1 TO 1500
270 NEXT DELAY
280 CALL CLEAR
290 GOSUB 320
300 NEXT I
310 GOTO 430
320 PRINT "TYPE THE NUMBER"
330 INPUT RES$
340 IF RES$<>MSG$ THEN 380
350 CALL CLEAR
360 PRINT "GOT IT!"
370 GOTO 390
380 PRINT "SORRY! THE NUMBER WAS:";MSG
    $
390 PRINT "DO YOU WANT TO CONTINUE.
     ""Y"" OR ""N""."
400 INPUT ANS$
410 IF ANS$<>"Y" THEN 430
420 RETURN
430 END
```

FINAL COMMENTS

Once you've had a chance to use this approach—defining the problem, doing an informal outline, flowcharting, and then coding—in a project of your own, programming your computer will no longer be as forbidding and mysterious as you first thought.

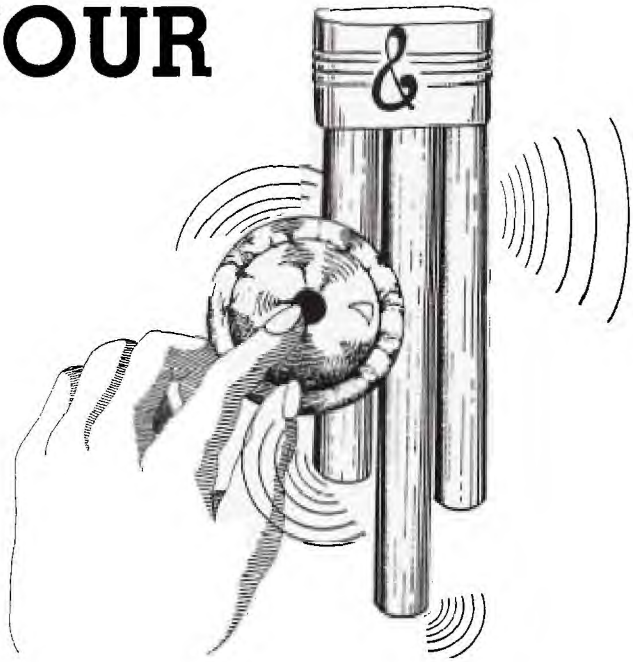
Before attempting programs of your own, you may want to try a little exercise. Add the following features to the previous program:

(1)Allow the user to choose how much time the digits are displayed on the screen.

(2)If the response is correct, play a 3-note chord.

(3)If the response is incorrect, play one note of noise and print a screen message that tells how many digits were contained in the largest number correctly guessed.

LIVENING UP YOUR CALL SOUNDs



The CALL SOUND subprogram in TI BASIC commands an amazing integrated circuit in your TI-99/4A called the SN76489 Sound Generation Controller. On a single chip, TI has squeezed in three programmable frequency dividers, a programmable noise generator, four programmable attenuators (volume controls), and eight registers to hold the data that control the tones, noise, and their volume levels. In effect, the tones and noise are synthesized to your specifications from a frequency of 3.58 megahertz; this is also the frequency that carries the color information from your computer to your color monitor or video modulator.

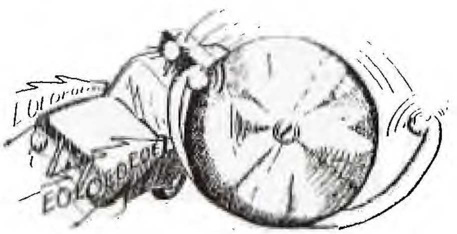
If you have used CALL SOUND only to produce miscellaneous beeps, noise, and music, read on. I'm going to give you some "mini programs" that demonstrate the variety of other sounds your Home Computer is capable of producing.

For the first example, let us try to re-create the sound of a door bell of the type associated with the once popular "Avon Calling" commercial. This is an example of an object that is struck a sharp blow and allowed to vibrate at its resonant frequencies. The following characteristics are needed to recreate this sound: 1) the fundamental frequencies of the two tones, 2) the overtone frequencies, and 3) a gradually decaying volume. Those of you with a sense of absolute pitch would immediately recognize the two fundamental frequencies, but in my case, I actually measured the dimensions of the sounding bars and their points of support and determined with a magnet that the bars were probably steel. From a textbook, *Acoustical Engineering* by Harry F. Olson, I obtained the formula and values of the constants needed to calculate the resonant frequencies of the bars. The calculated frequencies came out to be very close to 698 and 554 cycles per second (F and C# above high C). The book also told me that the two closest overtones were 2.756 and 5.404 times the fundamental frequency. The bars were supported on rubber mounts close to the theoretical nodes (points of minimum vibration) for the fundamental and the first overtones, but were located near points of maximum vibration for the second overtone. I therefore assumed that the second overtone would be dampened out, so I omitted it from the CALL SOUND specification for each tone. The decaying volumes for the tones were obtained by including each CALL SOUND in a FOR-NEXT loop as follows:

```
100 REM DOOR CHIMES
110 FOR A=0 TO 30 STEP 5
120 CALL SOUND(-99,698,A,1924,A)
130 NEXT A
140 FOR A=0 TO 30 STEP 5
150 CALL SOUND(-99,554,A,1527,A)
160 NEXT A
```

If you are wondering about the significance of the 99 for the durations, it is simply an easily keyed number larger than the 50 milliseconds needed to make the steps sound continuous. The minus sign indicates that the sound generator will be updated as soon as the new value for A is determined; the duration specified need only be long enough to cover the time between updates.

Next, let us try a sound in which the frequency varies with time. A siren is an example which can be characterized by a slowly rising and falling frequency. Apparently, this is a sufficient clue to the brain for us to recognize it as a siren. Try varying the frequency range step in the following program to see how far it can be varied and still be recognizable as a siren.



```
170 REM SIREN
180 N=1
190 FOR F=700 TO 900 STEP 5
200 CALL SOUND(-99,F,0)
210 NEXT F
220 FOR F=900 TO 700 STEP -8
230 CALL SOUND(-99,F,0)
```

```
240 NEXT F
250 N=N+1
260 IF N=4 THEN 270 ELSE 190
270 END
```

N=4 on line 260 limits the siren to 3 up-down frequency sweeps.

In the next example, let us vary both the frequency and the volume as a function of time. Imagine a large "killer" bee buzzing around you, with the frequency of the buzz proportional to the rate of the beating wings, and the volume proportional to the closeness of the bee.

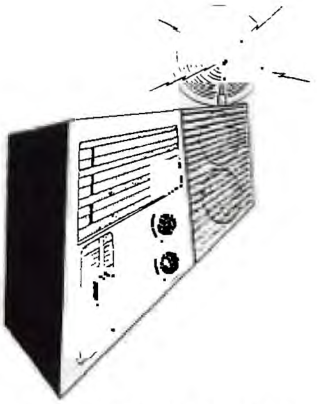
```
280 REM BEE
290 N=1
300 CALL SOUND(-99,RND*8+110,RND*10)
310 N=N+1
320 IF N=75 THEN 330 ELSE 300
330 END
```

Unlike the previous examples, where the variations in frequency and volume were obtained by using a FOR-NEXT loop, the variations in this case were obtained by using the RND statement. It is interesting to note that this routine will not sound the same in TI Extended BASIC—the bee sounds very sluggish. This is one case in which the TI BASIC runs faster than the Extended version.

For the next sound, imagine that you are tuning a shortwave radio receiver. The background static is simulated with the noise type (-8), and the random signal is simulated with frequency #3. The random volume on frequency #3 simulates varying signal levels with the noise volume formulated to be high when the signal level is low and vice versa.

```
340 REM SHORTWAVE RECEIVER
350 N=1
360 F=RND*15000+110
370 A=RND*30
380 CALL SOUND(-99,111,30,111,30,F,A,-8,30-A)
390 N=N+1
400 IF N=100 THEN 410 ELSE 360
410 END
```

Frequencies #1 and #2 are "do nothing frequencies" since their volumes are set to the minimum and are inserted so the program will recognize frequency #3, from which noise type -8 is derived. The 111's therefore were picked for the ease of inputting.



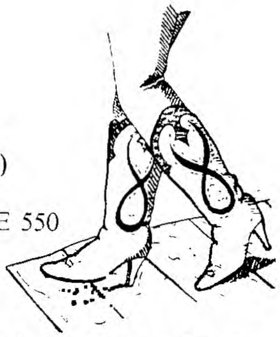
Next, imagine that the radio of the previous example is now tuned to a "pre-ASCII" teleprinter signal which uses an 850 cycle-per-second frequency shift to differentiate between a mark and a space.

```
420 REM RADIO TELEPRINTER
430 N=1
440 CALL SOUND(22,2975,0)
450 FOR D=1 TO 5
460 S=850*INT(RND*2)
470 CALL SOUND(22,2125+S,0)
480 NEXT D
490 CALL SOUND(31,2125,0)
500 N=N+1
510 IF N=30 THEN 520 ELSE 440
520 END
```

One character consists of a 22 millisecond (ms) start pulse, followed by a five-bit code for the character with each bit 22 ms long, and a 31 ms stop pulse. Line 440 generates the start pulse, which is always a space. The FOR-NEXT loop in lines 450-480 randomly generates a mark or space pulse for the five data bits, and line 490 generates the stop pulse, which is always a mark. Line 510 limits the number of characters generated to 29. Like the "bee" sound, this will not come out well in Extended BASIC. In general, data communications signals are easy to imitate because they are well defined by standards.

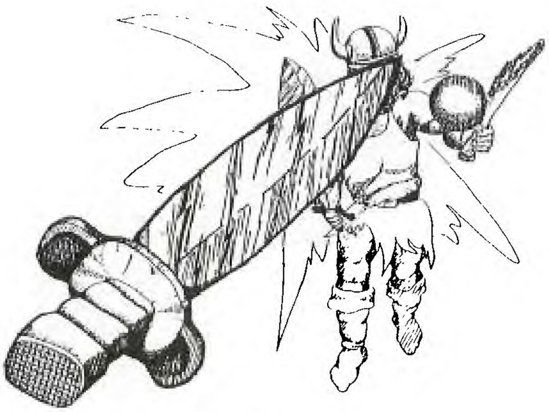
For a change of pace, try the following sound:

```
530 REM FOOTSTEPS
540 N=1
550 X=INT(RND*5)
560 IF X=2 THEN 620
570 CALL SOUND(5,-3,5)
580 CALL SOUND(30,-7,20)
590 CALL SOUND(500,-7,30)
600 N=N+1
610 IF N=30 THEN 640 ELSE 550
620 CALL SOUND(60,-7,20)
630 GOTO 590
640 END
```



The CALL SOUND on line 570 is the heel contacting the floor, followed by the sole contact on line 580. The CALL SOUND on line 590 is the delay between steps. Lines 550, 560, and 620 add a shuffle about once every 4 steps to make the footsteps sound a little more natural. Changing the noise type on line 580 from -7 to -5 will make the shoes squeak.

The sound of a sword fight can be re-created by recognizing that the sword blade is a resonator like the door chimes, except that instead of being essentially free, it is clamped at the handle—thus creating overtones at different ratios than the chime bars. Also, the amplitude decays faster, since the collision of the two blades would have a dampening effect.



```
650 REM SWORD FIGHT
660 N=1
670 FOR A=0 TO 30 STEP 15
680 CALL SOUND(-99,1000,A,3250,A,6750,A)
690 NEXT A
700 FOR D=1 TO RND*200
710 NEXT D
720 N=N+1
730 IF N=30 THEN 740 ELSE 670
740 END
```

Lines 700 and 710 add a random delay between sword clashes.

For the final example, let us try to simulate the sound of an internal combustion engine starting, accelerating, and then decelerating to a stop.

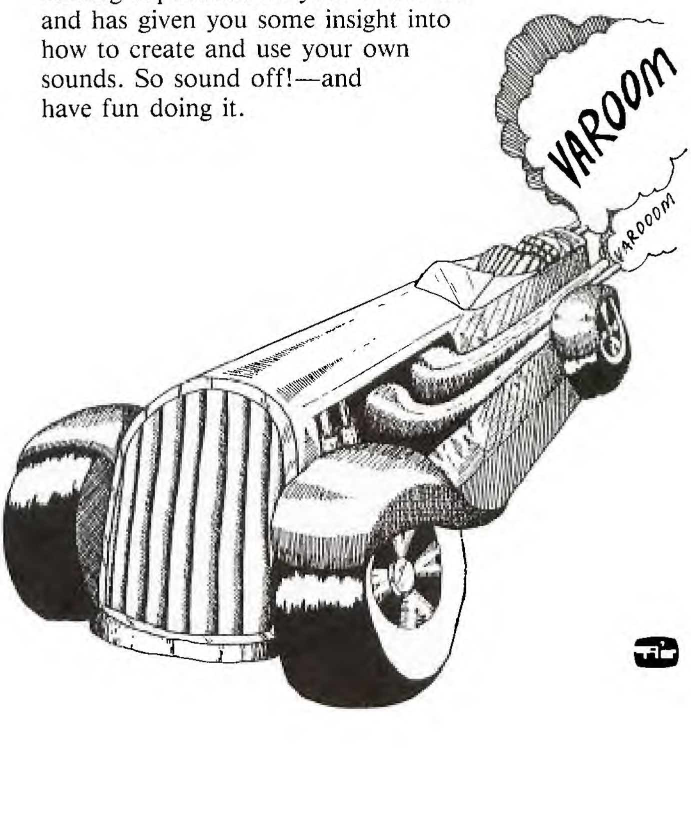
```
750 REM ENGINE
760 FOR N=1 TO 8
770 CALL SOUND(60,220,8,-5,0)
780 CALL SOUND(60,220,8,-5,5)
790 NEXT N
800 CALL SOUND(80,220,8,-5,0)
810 FOR F=1000 TO 5000 STEP 20
820 CALL SOUND(-99,111,30,111,30,F,30,-8,0)
830 NEXT F
840 FOR F=4000 TO 800 STEP -50
850 CALL SOUND(-99,111,30,111,30,F,30,-8,0)
860 NEXT F
870 END
```

Lines 760 through 800 simulate an electric starter motor. The accelerating and decelerating engine sound is made by sweeping noise -8 up and down in a FOR-NEXT loop.

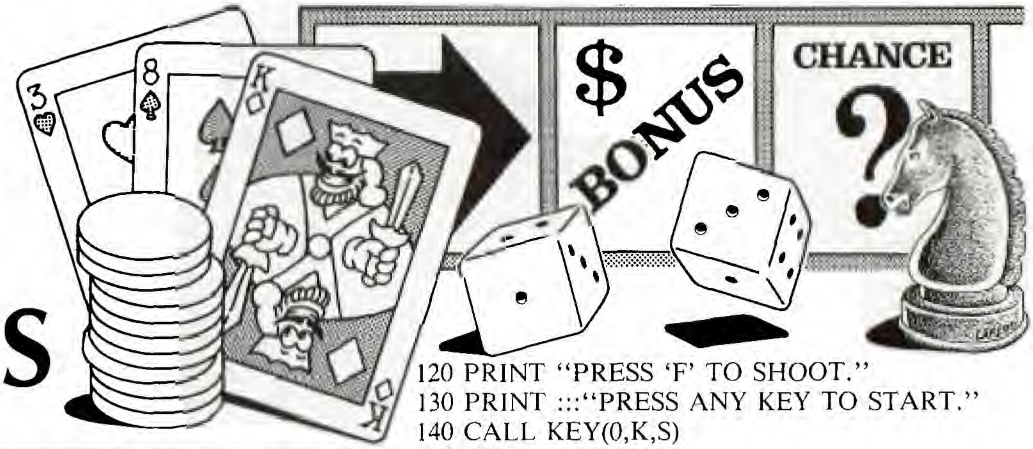
Now that you're convinced that your computer can produce a wide variety of sounds, you are probably wondering how one uses these sounds. If you are an adventure game programmer, suppose that the player is confronted with a door with a knocker and a bell button. Wouldn't it be more interesting if the player heard the bell upon pressing the bell button—before getting the usual textual message? Or if you are dynamically simulating a race car, you could use line 820 in the engine sound example in a CALL KEY loop where the F parameter would depend on the accelerator pedal setting. The duration in the CALL SOUND would have to be increased if you are updating other parameters in the loop for the sound to be continuous.

One nice thing about sounds is that the listener will make up the visual image that fits, which is why the radio programs of years past were so effective. The bee sound, for instance, immediately conveys the situation, whereas a screenful of color graphics would be hard-pressed to evoke the same feeling. Thus, for the programmer of interactive fiction, sound should be a very effective way to make a story come alive. If you could collect enough sounds, you could even write a sound effects program where a given sound could be accessed on cue for stage plays.

Hopefully, this article has opened your ears to the sound-making capabilities of your TI-99/4A and has given you some insight into how to create and use your own sounds. So sound off!—and have fun doing it.



FUN & GAMES



Psst! I've got a little secret for you, gang: *Designing* and *programming* your own game can be just as much fun as *playing* games produced by others. And best of all, it's really not as hard as you might think. . . .

Pick an Idea

You can have a maze, a game using dice, a card game, a memory-type game, a board game, a popular sport, a game involving logic, a game using skills or reaction time, some form of hide-and-seek, an adventure, or a myriad of space and shooting games. Still don't have a game plan? Walk through a video arcade to get some ideas.

Use the Computer

Now this sounds silly, doesn't it, since we're talking about writing computer games. Let me explain. If you write a game of tic-tac-toe or Othello for two players, you're really only utilizing graphics—the game could just as well be played on paper or on the board. But, if you write the game for one person against the computer, then you *are* using the computer to help go through a logic process. Another use of the computer is doing anything with random numbers.

Write Your Program

Of course, you may just sit at the console and begin programming your game and hope you can remember all the logic. Some programmers like to draw a flowchart. On logic games you may like "tree diagrams"—i.e., if the player does one option you branch one way; then depending on the next choice, you branch again and so forth. Other programmers prefer a structured approach—each process of the game is in a subroutine and the main program calls the subroutines in order. This type of program is easy to evaluate and easier for other programmers to follow than a program that has GOTO statements all over the place.

Include Instructions

Many players are anxious to play the game and won't read anything that comes with the game program, so it is wise to include simple instructions within your program. Players who are playing the game a second time, however, won't want instructions, so you must try to satisfy everyone. One method is to print the instructions on one screen with "PRESS ANY KEY TO START" at the bottom of the screen. The player can then look at the screen as long as he wants or immediately press any key to start the game.

```
100 PRINT "PRESS ARROW KEYS TO GO"
110 PRINT "LEFT OR RIGHT."
120 PRINT "PRESS 'F' TO SHOOT."
130 PRINT :::"PRESS ANY KEY TO START."
140 CALL KEY(0,K,S)
150 IF S<1 THEN 140
160 Program continues for game.
```

Another method is to ask the player if he needs instructions:

```
100 PRINT "NEED INSTRUCTIONS? (Y/N)"
110 CALL KEY(0,K,S)
120 IF K=78 THEN 150
130 IF K<>89 THEN 110
140 Program prints instructions.
150 Program continues for game.
```

If the player presses Y, instructions will be printed; if he presses N, the game starts. Any other key pressed is ignored.

Be sure the instructions are as clear and concise as possible. Use enough blank lines to make the instructions easy to read. Make sure words are not divided at the ends of lines, be sure to spell correctly, and use correct grammar.

"Dummy-Proof" Your Game

A nicer way of saying this is make your program "user-friendly." This means consider all possibilities of input. You never know what some other player will try to do. If he has to answer "yes" or "no," can he just press Y or N, or does he need to spell out and ENTER the answer? Pressing one key makes for less chance of error than using INPUT. What if the game asks for a *number*, and a *letter* is pressed? What if the game asks for a choice of 1 through 4, and the number 7 is pressed? If the player needs to use the arrow keys, is there a default value if he hits the wrong key, or is that key ignored—or worse yet, does the program crash?

Check for Speed and Captivation

You don't want the player to fall asleep between moves. If you have moving objects in your game, he wants them to be as fast as possible. The main hints here are to make the moving object just one character and to minimize the logic between moves. Remember, the more objects you have to move, the longer it will take. And if you don't need to worry about scrolling (lines moving up the screen), PRINTing characters is faster than CALL HCHAR or VCHAR.

Look Through your Listing

If you use the same group of lines several times, use a GOSUB and place the subroutine near the beginning of the program. For example, a subroutine to print a message M$ on Row X starting in Column 1 is

```
180 FOR J=1 TO LEN(M$)
190 CALL HCHAR(X, J,ASC(SEG(M$(J,1)))
200 NEXT J
210 RETURN
```

Within the program, the message and row numbers are defined, then the subroutine called:

```
1740 M$="BLUE WINS THIS TIME."
1750 X=22
1760 GOSUB 180
```

Check for unnecessary statements. I have seen a few listings that contain some coding that can never be executed or is superfluous, or a subroutine that is never called. Other cases may occur because of editing. For example:

```
900 GOTO 920
910 X=25
920 GOTO 980
```

Or:

```
900 GOTO 910
910 Z=Z+1
```

Or:

```
900 IF X=A THEN 700 ELSE 910
910 GOTO 980
```

Test Your Game

Again, check all possibilities. If you say your spaceship can move to the right and to the left, be sure to check *both* directions. Make sure positive and negative numbers work correctly in your calculations (you may want to use the ABSolute function). Check the scoring to see if it is adding correctly. Test the possibility of hitting the wrong key. Test moving objects at the edges of the screen.

Specific Game Coding

Random Numbers

Be sure to use the statement RANDOMIZE before using RND so each game played will be different. If random numbers are computed in several different places, consider using RANDOMIZE before each RND to ensure total randomization throughout the game. Sometimes a single RANDOMIZE statement at the beginning of the program does not work.

A simulation of rolling the dice would need a random number between 1 and 6:

```
100 RANDOMIZE
110 D1=INT(RND*6)+1
```

In a space program or skill-type game you may want to place obstacles at random positions. If you have several objects, DEFine a few functions at the beginning of the program, so they can be used more easily in the coding later:

```
100 DEF RX=INT(RND*24)+1
110 DEF RY=INT (RND*29)+2
120 CALL CLEAR
130 RANDOMIZE
140 FOR I=1 TO 5
150 CALL HCHAR(RX,RY,65)
160 NEXT I
170 CALL VCHAR (RX,RY,66)
180 STOP
```

The DEFinition statements must be numbered lower than the statements in which the functions are used. Lines 140-170 place five A's and one B in random X and Y positions for X from 1 to 24 and Y from 2 to 30.

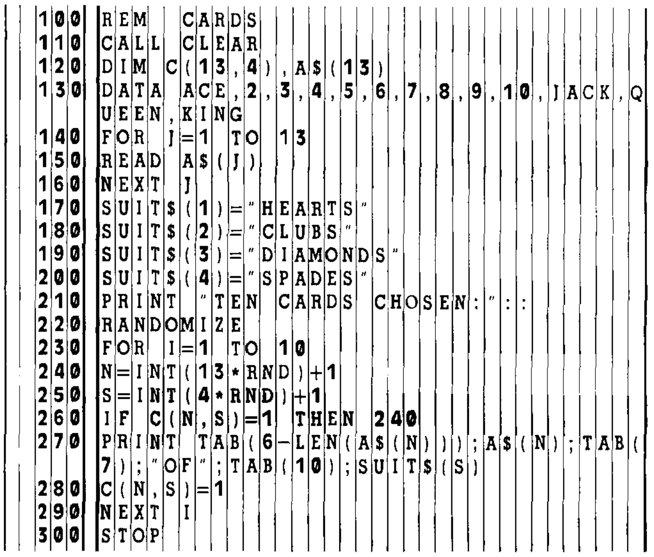
Another use of random numbers is choosing a random message or procedure. For example,

```
500 PRINT A$(INT(RND*9)+1)
```

chooses one of nine messages previously stored in the A$ array. For random subroutines, the coding would be

```
510 ON INT(RND*5)+1 GOSUB 220,250,300,350,400
```

Games using a deck of cards may use an array to keep track of which cards are dealt. You may use C$(52) for the 52 cards or a two-dimensional array C(13,4) where the first parameter is the number chosen and the second is the suit. An example for choosing ten cards follows. The values in the card array are initially zero. As a card is chosen, the corresponding C element is set equal to 1. In the following example I printed the card values, but you really should use the TI graphics to *draw* the cards.



```
100 REM  CARDS
110 CALL CLEAR
120 DIM C(13,4),A$(13)
130 DATA ACE,2,3,4,5,6,7,8,9,10,JACK,Q
    UEEN,KING
140 FOR J=1 TO 13
150 READ A$(J)
160 NEXT J
170 SUIT$(1)="HEARTS"
180 SUIT$(2)="CLUBS"
190 SUIT$(3)="DIAMONDS"
200 SUIT$(4)="SPADES"
210 PRINT "TEN CARDS CHOSEN:"::
220 RANDOMIZE
230 FOR I=1 TO 10
240 N=INT(13*RND)+1
250 S=INT(4*RND)+1
260 IF C(N,S)=1 THEN 240
270 PRINT TAB(6-LEN(A$(N)));A$(N);TAB(
    7);"OF";TAB(10);SUIT$(S)
280 C(N,S)=1
290 NEXT I
300 STOP
```

One more use of RND is for choosing random sounds. The CALL SOUND statement requires a frequency between 110 and 44733. Of course, most people cannot hear frequencies above 15,000; however, your dog may enjoy the higher frequencies. This statement plays a sound frequency between 110 and 2109:

```
300 CALL SOUND (200,INT(RND*2000)+110,0)
```

You may wish to use random sounds while you're placing objects randomly on the screen.

Sound and Noise

A lot of the fun in programming games is choosing the sound effects to fit your game. The following is a program that demonstrates the "noises" available on the TI-99/4A:

```
100 REM NOISE
110 FOR I=-1 to -8 STEP -1
120 CALL SOUND(4000,I,0)
130 CALL CLEAR
140 CALL SCREEN(ABS(I)+2)
150 PRINT "NOISE NUMBER";I
160 NEXT I
170 GOTO 110
180 STOP
```

Listen to these noises and choose what you need for your game. You can make crashing noises, explosions, airplane or car motors, splats, bounces, rocket boosters, missile fire, or whatever you need. The noises may be varied by adding another set of sound frequencies and loudnesses.

Time

Since the 99/4A does not have an accessible real-time clock, time may be simulated by placing a counter in the CALL KEY routine or another loop that is executed regularly. The following example shows a counter as you move the asterisk up and down with the up and down arrows (E and X) keys. After a time of 100, the number of moves you have made is printed. You will notice that if you press a key, the counter moves more slowly than if no key is pressed, so the counter is not as even as a metronome but good enough for games.

```
100 REM  TIMING
110 CALL CLEAR
120 X=12
130 CALL HCHAR(X,15,42)
140 TIME=0
150 CALL KEY(0,K,S)
160 TIME=TIME+1
170 FOR I=1 TO LEN(STR$(TIME))
180 CALL HCHAR(22,I+3,ASC(SEG$(STR$(TI
    ME),I,1)))
190 NEXT I
200 IF TIME=100 THEN 350
210 IF K<>69 THEN 240
220 DX=-1
230 GOTO 260
240 IF K<>88 THEN 150
250 DX=1
260 CALL HCHAR(X,15,32)
270 X=X+DX
280 IF X>0 THEN 300
290 X=24
300 IF X<25 THEN 320
310 X=1
320 CALL HCHAR(X,15,42)
330 MOVES=MOVES+1
340 GOTO 150
350 PRINT "MOVES= ";MOVES
360 STOP
```

Following is another example of a way to time a process—in this case, typing your name.

```
100 REM  SPEED TEST
110 CALL CLEAR
120 TIME=0
130 PRINT "TYPE NAME THEN PRESS ENTER"
    ::::::
140 FOR Y=3 TO 28
150 CALL KEY(0,K,S)
160 TIME=TIME+1
170 IF S<1 THEN 150
180 IF K=13 THEN 210
190 CALL HCHAR(21,Y,K)
200 NEXT Y
210 PRINT "TIME= ";TIME
220 STOP
```

An accurate way to delay for a specific length of time in your program is to use CALL SOUND for the number of milliseconds you need. Use 30 for the volume level and a very high frequency if you don't want to hear anything. While the CALL SOUND statement is being executed you may also be doing graphics of calculations. To end your timing device you will need another sound statement with a duration of 1. The following example illustrates how the CALL SOUND statements may be used for a rocket countdown.

```
100 FOR I=10 TO 1 STEP -1
110 CALL SOUND (1000,44000,30)
120 PRINT I
130 NEXT I
140 CALL SOUND (1,44000,30)
150 Program continues for rocket blastoff.
```

Arrow Keys

In games where you move a character up, down, left, or right, you may wish to have the player press the arrow keys. The arrows are on the keys E, D, X, and S. A CALL KEY statement is used to receive the player's input; then the program branches, depending on which arrow is pressed. Any other key pressed should be ignored so your program doesn't crash with bad values.

The following routine will draw a trail of asterisks as you press the arrow keys. Remember, you must consider the edges of the screen or you will get a "BAD VALUE" message. Lines 270-340 test for the edge values and will keep the asterisk at the edge position.

```
100 REM  MAKE-A-TRAIL
110 CALL CLEAR
120 X=12
130 Y=15
140 CALL HCHAR(12,15,42)
150 CALL KEY(0,K,S)
160 IF K<>69 THEN 190
170 X=X-1
180 GOTO 270
190 IF K<>68 THEN 220
200 Y=Y+1
210 GOTO 270
220 IF K<>88 THEN 250
230 X=X+1
240 GOTO 270
250 IF K<>83 THEN 150
260 Y=Y-1
270 IF X>=1 THEN 290
280 X=1
290 IF X<=24 THEN 310
300 X=24
310 IF Y>=1 THEN 330
320 Y=1
330 IF Y<=32 THEN 350
340 Y=32
350 CALL HCHAR(X,Y,42)
360 GOTO 150
370 STOP
```

Remember that there are many ways of coding to get the same result, and the examples presented here are just that—*examples*. The following routine illustrates another way to use the arrow keys to move a character. This time the previous character is deleted. Also, lines 330-410 will make the asterisk scroll to the other side of the screen instead of staying at the edge.

```
100 REM  MOVE A STAR
110 CALL CLEAR
120 X=12
130 Y=15
140 CALL HCHAR(X,Y,42)
150 CALL KEY(0,K,S)
160 IF K<>69 THEN 200
170 DX=-1
180 DY=0
190 GOTO 310
200 IF K<>68 THEN 240
210 DX=0
220 DY=1
230 GOTO 310
240 IF K<>88 THEN 280
250 DX=1
260 DY=0
270 GOTO 310
280 IF K<>83 THEN 150
290 DX=0
```

```
300 DY=-1
310 CALL HCHAR(X,Y,32)
320 X=X+DX
330 IF X>0 THEN 350
340 X=24
350 IF X<25 THEN 370
360 X=1
370 Y=Y+DY
380 IF Y>0 THEN 400
390 Y=32
400 IF Y<33 THEN 420
410 Y=1
420 CALL HCHAR(X,Y,42)
430 GOTO 150
440 STOP
```

A more compact approach to automatic scrolling is to replace lines 330-360 and 380-410 with these two lines:

```
330 X=INT(24*((X-1)/24-INT((X-1)/24)))+1
380 Y=INT(32*((Y-1)/32-INT((Y-1)/32)))+1
```

Split Keyboard

A split keyboard is used when two competing players or teams are interacting with moving objects on the screen. Instead of CALL KEY(0, KEY, STATUS), you will need to recieve input with CALL KEY (1, KEY1, STATUS1) and CALL KEY(2, KEY2, STATUS2). You may wish to use a *Video Games 1* Command Cartridge overlay for the arrow keys. You'll notice the arrow keys for the right side of the keyboard are keys I, J, K, and M. The key codes returned in CALL KEY are 5 for up, 2 for left, 3 for right, and 0 for down for both sides of the split keyboard. *Note:* There is a slight problem in testing for zero on the 99/4A console, so use logic such as IF KEY2 + 1< >1 instead of IF KEY2 < >0. It also seems wise to avoid using SHIFT, ENTER, G, B, slash, semi-colon, comma, periods, and the space bar for key input (such as firing a missile) because the key codes for these keys are different on the 99/4 and 99/4A. You will want your game to work on both consoles so you can share with others.

An example of the logic for two players and a split keyboard is shown in lines 910-1510 from the game *Maze Race* in the section "Computer Gaming."

Joysticks

Enter the sample programs that come with your TI Wired Remote Controllers to get an idea how to program movement with one or two joysticks. Keep in mind that CALL JOYST (KU, X, Y) returns X and Y values of 0 and plus or minus 4, depending on the position of the lever. By the way, don't get these X and Y values confused with X- and Y-coordinate values for HCHAR and VCHAR.

Following is a sample program that allows the player to move the asterisk with either the arrow keys or a joystick. Line 150 is a CALL KEY statement. If no key on the keyboard is pressed, all the arrow key logic is skipped and CALL JOYST (line 330) is executed. If a key has been pressed, then the joystick logic statements (lines 330-350) are skipped. (Remember: ALPHA LOCK up for joysticks, down for arrow keys.)

```
100 REM    JOYSTICKS
110 CALL CLEAR
120 X=12
130 Y=15
140 CALL HCHAR(X,Y,42)
150 CALL KEY(0,K,S)
160 IF S=0 THEN 330
170 IF K<>69 THEN 210
180 DX=-1
190 DY=0
200 GOTO 360
210 IF K<>68 THEN 250
220 DX=0
230 DY=1
240 GOTO 360
250 IF K<>88 THEN 290
260 DX=1
270 DY=0
280 GOTO 360
290 IF K<>83 THEN 150
300 DX=0
310 DY=-1
320 GOTO 360
330 CALL JOYST(1,JA,JB)
340 DX=-JB/4
350 DY=JA/4
360 CALL HCHAR(X,Y,32)
370 X=X+DX
380 Y=Y+DY
390 X=INT(24*((X-1)/24-INT((X-1)/24)))
    +1
400 Y=INT(32*((Y-1)/32-INT((Y-1)/32)))
    +1
410 CALL HCHAR(X,Y,42)
420 GOTO 150
430 STOP
```

Detecting a Crash

Probably the most common way of determining if your moving object hit some obstacle in position X, Y is by using CALL GCHAR(X ,Y ,C). The C value returned is the character number occupying positon X, Y on the screen. For example, you may then test if C = 32 (space); if so, the program could continue. But if C = 96 (one type of object), the program would branch one way, and if C = 99 (another object) the program would branch another way—with the appropriate sounds and graphics.

Another method of determining the character in a certain position is to have the screen positions in an array and have each array element contain information about the character in that position. For example, you may have an array A(24,32) for the 24 rows and 32 columns of the screen. Each element of A could be zero for a space and 1 for a block in a maze. Your testing statement would look like

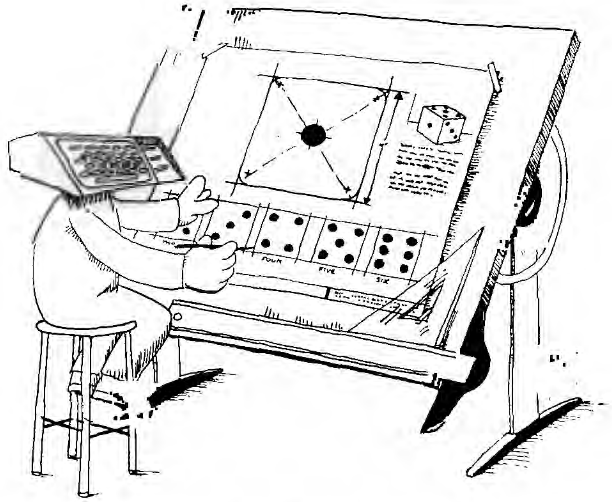
```
200 IF A(X,Y)=1 THEN 240
```

This means if the position X ,Y is a block, then branch to line 240 where a crashing noise is made and appropriate action takes place. Note that by using OPTION BASE 1, you will eliminate Row 0 and Column 0 and save memory space.

Do It!

I've presented some fundamental hints and ideas for programming; now it's your turn to put on your thinking cap, turn on the computer, and have fun writing your *own* games!

CHUCK -A- LUCK



HOW PROS PROGRAM

PART 1: "A bad beginning makes a bad ending"

Have you ever LISTed a program that you bought, and after looking at the listing thought, "That's not so hard. . ."?

Actually, you're quite correct in *believing* that writing a good program is not really so difficult. But when beginning programmers sit down to translate this belief into a finished program, many wind up confused and frustrated. This can most often be attributed to their accompanying belief that writing a program is just a matter of sitting down at a keyboard and banging away at it—a procedure that is destined to fail.

To explain an alternate approach—one that all experienced programmers use—I'll list the sequence of events that I go through whenever I want to write a program.

First, I sit down and decide what the program is going to do. If it's a game, I write down all the rules (even if I'm making the game up). If it's a business-oriented program, I decide what features it has to have—i.e., sorting, saving data, or printer output. Without this initial planning, I wouldn't have a goal in mind when I reached subsequent stages.

Second, I design the program. A design is a plan showing the functions (the "whats") that a program contains. For example, a program that plays the game of *Chuck-A-Luck* would contain the following functions: (1) explaining the rules, (2) rolling dice, (3) accepting bets, (4) paying off (or collecting) money, and (5) checking for the final win/loss condition. (See Figure 1 for the rules of the *Chuck-A-Luck* game that I'll be using as an example.) I don't figure out *how* I'll do these things at this time; I just figure out *what* the program has to do.

Third, I group together any "whats" that I feel are different parts of the same top-level module or function. For example, giving the rules, generating the dice characters, and getting player names are all part of initialization; so at first I put them together under the top-level function name of START-UP. Now, I'll write these functions down in a list. For this simple game of *Chuck-A-Luck,* my list of top-level modules looks like this: START-UP, DICE-ROLLS, and END-GAME.

Next, I look closely at each function (or module) and list everything I need to do in each of these modules. For example, the START-UP function will also have to include things like DIMensioning data, asking if rules are needed, asking the number of players, and initializing data fields. The DICE-ROLLS module will have to take bets for each roll, roll the dice (and display them), decide the winners and losers, and recompute new cash balances for each player. The END-GAME routine will have to print an appropriate message after all players go broke or a winner is determined, ask if any player wants to try again, and restart the game.

Notice that all I have done so far is write down the "whats" of the program. I haven't looked at the "hows" yet. The technique I have been using is called *top-down design* and consists of breaking a problem or program into its component modules. These new modules are themselves broken down into even smaller ones until you finally arrive at reasonably sized, easily codable low-level modules.

Sometimes, as you break modules into smaller and smaller routines, you may find at the lowest levels that some modules are duplicated. That means that the same module can belong to (or be used by) more than one higher level module. This kind of routine is called a *subroutine.* A good example of a subroutine that you would code in TI BASIC would be a routine to display messages on the screen using CALL HCHAR. It would

Figure 1. **CHUCK-A-LUCK Rules**

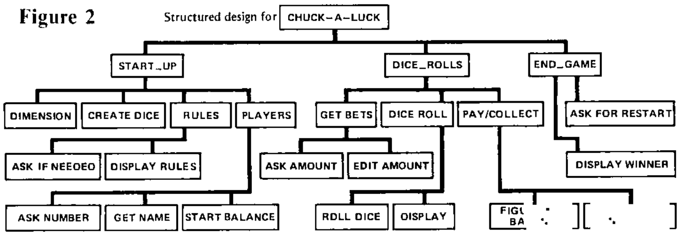
1. Each player starts with $500.
2. Each player bets an amount of money from $10 to $50 on a dice value from one to six.
3. Three dice are rolled.
4. If no die has a value equal to the value selected by a player, he loses his bet.
5. If one die has a value equal to the value selected by a player, that player receives an amount equal to the amount that he bet.
6. If two dice have that value, the player receives twice the amount he bet.
7. If three dice have that value, the player receives three times the amount he bet.
8. A player who goes bankrupt is out of the game.
9. The game ends when only one player remains. The remaining player is the winner.
10. If all the remianing players go bankrupt at the same time, there is no winner.

be called by other routines in your program (using the GOSUB statement) whenever they wanted a message displayed on the screen.

But when do we stop *designing* and actually start *writing* the program? There's a different answer for each program and programmer. The idea here is to stop at a point where you feel that you can picture in your mind what the code should look like. For advanced programmers, this may mean that there are fewer modules in a design, and each module will have a lot of lines of code in it. For beginners, I would recommend stopping when each module is self-explanatory to you—usually this requires about 10-20 lines of BASIC code.

As I am doing my design, I keep track of the modules by drawing a structured design chart which shows which lower-level modules belong to (are to be part of) each top-level module. After going over all of my top-level modules, my structured design looks like Figure 2, below.

Figure 2



Take a look at the START-UP module. It includes a low-level function called GET NAME. Now look at DICE-ROLLS. It includes a low-level module called EDIT AMOUNT. These routines demonstrate two rules I always follow when I design programs: First, whenever possible, I try to make the program "user-friendly." This includes things like displaying understandable error messages (instead of a cryptic "NOT POSSIBLE"), using player names (instead of numbers), and giving prompts that explain what action is required. Too many people write programs that call you PLAYER #1 and tell you to do something by saying things like "CODE?". It takes only a little longer to write a program that says "OK, MIKE, HOW MUCH WILL YOU BET THIS TIME?" And the results are well worth the effort.

Of course, in a business-oriented program, you don't usually ask for people's names. But such a program can become user-friendly just by judicious use of self-explanatory prompts and error messages. Of course, being user-friendly makes for longer and larger programs. I personally don't worry about how much extra memory it requires at first. After all, I can always remove those wonderful messages and replace them with a "NOT POSSIBLE" message if I have to!

By the way, if you stop to think about it, the TI BASIC and Extended BASIC that you work with is very user-friendly. It does things like prompt you for cassette tape I/O and give you meaningful error messages when you are in EDIT or COMMAND mode.

The second rule that I always follow is that any data that is input into a program must be fully edited—i.e., it must be checked to make sure it is the proper type and in the proper form. Always! Always! Always! I said it three times because this is one of the major differences between a professional program (which can be used by anyone without "blowing up"—especially when encountering some strange input from an unfamiliar user) and a program which is usable only by the person who wrote it.

Some of the rules that I always like to follow include:
1. Make sure that numeric data really is numeric (of course this is something the TI BASIC does for you automatically).
2. Make sure that integers really are integers (and not decimals or scientific notation).
3. Make sure that the data itself is realistic (always test for maximums and minimums—e.g., making sure nobody bets more money than he has!).

I'm now finished with my design as far as *what* modules are needed. The fifth step in creating a program is to decide what information I need to communicate between these modules. The information that is passed from one module to another is called a *variable*. And deciding on what variables are needed before you sit down to write program code is just as important as deciding what modules you need. If you make a mistake in your design variables, the last phase of programming (called debugging) will take twice as long as it needs to be. This is because whenever you realize that you need a new variable, you have to make coding changes in modules (that have already been coded) in order to handle them. And changing code is what destroys well-written programs!

Programs will also need variables that are used only within a module (i.e., things like loop counters), but you don't have to worry about them during your design. As long as a variable that is only used within a module has a unique name (not used again in another module), then no problems should arise when debugging. Of course, if the variable name will be used again in another module (which is a bad idea unless memory is, tight), then it is just as important as a regular variable's.

The variables that I need to communicate between my *Chuck-A-Luck* modules are
The number of players
The player names
The cash each player has on hand
The amount bet by each player
The dice value on which each player bets
The value of each die

Choosing the names for these variables is equally important. A poorly chosen name is asking for trouble when you get down to writing and debugging your code. A good variable name has the following three attributes:
It is long enough to say what it is and what it's for.
It is short enough so as not to slow down the program.
It does not look too similar to any other variable.

For the *Chuck-A-Luck* game, I'll use NO__PLAYERS, PLAYER__NAME, PLAYER__CASH, PLAYER__BET, PLAYER__DICE, and DICE__VALUE as my variable names. And, I won't re-use variables that are used within a module.

Now my design is finally complete, and I'm ready to start coding. I have done everything that I could to insure that the program will do its job and am ready for the sixth step in creating a good program—planning the code. As we have just learned, the first rule of good programming is PLAN, PLAN, PLAN!

CHUCK-A-LUCK PART 2: "Make no little plans. . . ."

Building a good program is a lot like building a house. First, you need a good design. Then, you need good tools, good materials, and good work habits to use them all properly. We have discussed a way to develop a good design by using a techinque commmonly called *structured design* or *top-down design.* Now we'll talk a little about how to get the necessary tools and materials and cultivate the habits that we need.

After completing our design effort, you might expect the next step to be coding the program. But this, in fact, is not the case. Just having a good design doesn't mean that the code in your program will be correct or that you will write the best code for the job. In every task, there are two things to remember: The first is that you want to do the right *job.* The second is that you want to do the job *right.* To do the right job means that your code has to follow the design that you came up with. To do the job right, you have to create the best code for the job. And like anything else, these both require planning. That's right! We still have some planning to do. Only this time, we must plan our code.

The first thing to do is refresh your memory on the design we came up with to play *Chuck-a-Luck.* Notice how we developed the modules that tell us *what* to do, but not *how* to do it. The purpose of planning our code is to figure out *how* we want to do it in the best possible way. At the same time, we want the "hows" that we develop to be easily coded and debugged, to execute quickly, and to be easily modifiable so that we can make future improvements.

Starting UP with START-UP

Let's start with the module called START-UP. One of its top-level components was DIMENSION. That module is needed to set up the dimensioning of any arrays needed in the program. Although it is not absolutely necessary to code the DIMension statement at the very beginning of your program, I have found that it is always best to put it right up front. So, when I plan my code, the DIMENSION module will be my very first line of code. Another good coding habit to get into is to start your programs at line 100, which leaves you room in the front of your program in case you have to add an extra statement to start off your program. I will reserve lines 100-140 for any dimensioning of data that I will need. But before I go on with the remaining design of the code, I think that we had better take time out to talk about the DIM statement and what it is used for.

When I was doing the design, I knew from my original plan that the program was going to have to handle 4 players. That meant that every time I did something concerning a player, I would have to know which player I was dealing with. For example, if each player was going to make a bet and win or lose, the program would keep track of these things (called variables) for each player. There are two ways of doing this. The first way is to give a different name to each one of these variables for each player. That is, I could keep track of each player's bet by having one variable called BET_1 and another variable called BET_2, and so on. This way, I would know at a glance what was contained inside the variable. The only problem with this way of doing things is that the program needs separate code for each player. This means that you would have to key in more lines of code. It means more chances for data entry errors. It also means the possibility that you could accidentally write the code for each player a little bit differently, which in turn means that you would need to debug your code for each player.

Suppose, however, that you did not need to give each player a different variable name. Suppose that you could just call the variable by the single name of BET. Then the code for each player would be the same. As a matter of fact, you would have to code the logic only once, because it could be re-used for each player. As you can see, this would be a great improvement. You still have more than one player so you would have to be able to say which player's BET you wanted to deal with. Well, the way that the BASIC language handles this is to allow you to set up an array called BET. This array has only one name but contains multiple slots. Imagine an apartment building called BET containing only one floor with a lot of rooms in it. The room numbers start with 0 and increment by one. The computer can put the betting information for player 1 into room number 1, the information for player number 2 into room number 2, and so on. Now, in order to look at the bet of player number 3, all we have to do is tell the computer to look at room number 3 of BET. We do this by saying BET(3). The value 3 is called a *subscript* of the array called BET.

This is an improvement over saying BET_3 but not much. But if the computer can be told which subscript (room number) to use via another variable, then you can realize a great improvement. Suppose all you had to do was tell the computer to look at something called X, and that X had the value of the subscript in it. Now you just put the room number inside X and tell the computer to look at BET(X). How do you put the room number into X? The same way you put any number inside any variable. You can say things like X=3 or X=A+B or set X to a range of values in a FOR-NEXT loop. The important thing is that you do not have to know in advance what is in X before you execute the code. By the way, I used the name X just as an example of my subscript name. We could have called it PLAYER_NUMBER, or I, or any other legal variable name. Also, just because a variable is being used as a subscript in one part of your program, it doesn't mean that the variable can only be used as a subscript. Any variable can be used as a subscript. It is also possible for two (or more) variables to be used as subscripts for the very same array, depending on what you are trying to do.

"Roomy" Arrays in TI BASIC

Now, two questions should be running through your head. The first question should be, "How many rooms can TI BASIC build for a given array?" The answer is that it depends on what (if anything) you tell it to do. If you don't tell it anything, it will automatically set aside 8 rooms (slots) for any array it may meet in your code. It will do this the first time it sees the array. If you need more than 8, you may not want to waste space on unused slots. In either case—less than 8 or more than 8 with no waste space—you tell it how many slots you need by using the DIM statement.

The second question should be, "What about room 0?" In my game, it is always empty. In some programs, however, room 0 may very well be used. If room 0 is not going to be used at all, you can tell this to TI BASIC so that it won't waste computer memory with a room 0. This is done by putting a statement with OPTION BASE 1 in front of the DIM statement.

Of course, just as an apartment building can have rooms on more than one floor, an array can have more than one level of slots. But since we don't need multiple levels in our program, we'll leave a discussion of this to a later time. For now, let's look at our variable list and see what variables are going to be arrays so that we will put them in our DIM statement. We will need to keep track of information for 1 to 4 players. In addition, there will be 3 dice and each die will have a value. Look at line 100, of the TI BASIC listing that starts on page 59, to see how I coded the DIM statement for these arrays. Notice that you cannot use a "-" as part of a name in TI BASIC. You can make variable names with several words in them by using the underline character ("_") to connect the words. For instance, I coded the array as DICE_VALUE in this program.

Leaving Out the Difficult is Easier

We are not ready to begin planning the code for the rest of the START-UP module. Because (by definition) this code will only be used once for each game, I like to keep it up far away from the main logic of my program. For this reason, I usually begin coding these one-shot modules at line number 20000 to give myself a lot of leeway in case I leave out a line of code or have to add another line during debugging. I always increment my statement numbers by 10 or more. In addition, I also make sure that there are plenty of unused statement numbers between the end of one module and the start of another.

The first module to be coded in START-UP is responsible for creating the graphics for the dice. Naturally, you now expect me to give you the code. But I won't! You see, it's not really important that I do this right away; I can always create the dice later after I am sure that the rest of the program is working correctly. This is one of the important advantanges of designing and planning your program.

When you plan your code, don't rush right into figuring out how the code in all your modules will look. First, decide what modules or parts of modules can be left out without affecting the program logic. For example, the code to display instructions can be added as the very last part of your programming effort. A program usually will contain whole modules requiring complex code that can be replaced by easy code the first time through. After you are sure that the program as a whole is working correctly, you can gradually replace the easy code with the complex code. Why? Because it's easier to find your mistakes in an easy program! So an important "rule of thumb" is to always start out with an easy version of your program. Then, as you add the difficult pieces, you at least know where to look if you hit a snag in your debugging.

So, if I leave out all the graphics for now, what can I substitute in their place? I can simply display the number of each die instead of graphically showing the dice themselves. After the program is running, I will go back and add the graphics as well as any sound routines. Look at what I am trying to accomplish this way:

1. By leaving out unnecessary code, I can get the program up and running faster. This means that I can begin debugging my program earlier. This in turn means *easier* debugging because there is less code to go through.

2. By using easy code in place of complex code in some modules, I make it easier to debug the "guts" of my program. After knowing that the program runs correctly, I can begin replacing the easy code with the hard parts a little at a time. Then I will test only one or two new parts at a time. This means easier debugging because any problems will probably be due to the new code.

3. After ensuring that the main portions are running correctly, I can "fool around" with the hard portions without worrying that I will hurt the program's logic. For example, after I know that the program is running correctly by displaying the dice numbers, I can now experiment with how I want the dice themselves to be displayed. I can even come up with two versions—one for TI BASIC and a different one for Extended BASIC using sprites! I won't have to worry that adding different versions of this code will destroy my program.

4. By getting a version up early, I can see if my program is worth continuing. After seeing it in action, I may decide that it just isn't worth the effort to continue with the coding.

So for right now, I won't code the CREATE DICE routine but I will set aside lines 20000-20500 for the code later. The next module is called RULES and will be responsible for giving the rules when asked. One part asks if the rules are wanted; another displays them. Like the CREATE DICE module, the entire code for this module isn't needed now. But if I do code in the part asking if the rules are wanted, I can test this part of the logic. If the program you are writing is large enough, you may decide to leave both of these parts out on your very first try.

Since I have decided to code part of this module, I will lay it out in lines 21000-22000. The first thing I want to do is clear the screen. This will attract the players' attention and remove any "clutter" that may be on the screen from any previous program. It's always a good idea to start out your program with a CALL CLEAR statement. Notice that in my code in lines 21010-21050, I am asking the players for information *and telling them in what form I expect the answer to be!* Too often, a programmer will code his program so that he is expecting a particular answer, but never tell the person using his program what form the answer should be in. There is nothing more frustrating to a user than trying to figure out what the person who coded the program means when the program displays a message like CODE?, and what the valid values of the input are. You should try to develop the habit of explaining what data you are looking for and what legal values the program will allow as part of your code for an INPUT statement.

The next thing the module does is make sure that only the first character is going to be looked at. This is done by using the SEG$ function to strip off the rest of A$. One of my programming "rules of thumb" is to minimize the chance of a program user entering bad data. If I am only expecting a Y or N, I want to look at only the *first* character of the input. If the wrong answer is given, an appropriate message is displayed and the original question is asked again. The code to display the message will be eventually located in lines 21000-21990, but I'll just put in a REM statement to show where the code will be added later.

The next module (called PLAYERS in our design) is very important and easy to code, so I will code it in full the first time out. This is done in lines 22000-22330. Notice that it prompts the player for the required data in each case, and edits the input to insure that only valid data gets in. One of the main differences between a well written program and a poorly written one is the amount of editing done on input. The "hows" of an edit for an alphanumeric field should always include a test for an empty field (called a *null string*). TI BASIC allows a null string to be entered in response to an input statement. This kind of string data can cause a number of problems in your program, expecially if you want to display the data on the screen. I always test for an empty field whenever I INPUT a string variable. That is what I am doing in line 22140.

It may also be necessary to limit the size of an alphanumeric field depending on how you want to display it on the screen. For example, you may want to limit the size of a player's name so that it fits on the same line as his cash balance. The best way to handle this is to check its length (using the LEN function) as part of your edit. If the player enters a name that is too long, you can tell him so, and ask that he enter a shortened version of his name. There is also, however, another way: You may shorten the field yourself by using the SEG$ function—as I do in line 22250.

When numeric fields are entered into your program, there are always four things you must edit for. First, you have to make sure that numeric data was entered. Luckily, TI BASIC will do this for you automatically so you don't have to write any code to test for this. You should get in the habit of immediately testing your input as follows: (1) check to make sure it isn't too large, (2) check to make sure it isn't too small, and (3) especially check to make sure it is a whole number (if that's what you are expecting). Look at my code in statements 22020-22030 to see what I mean. Also note that if the answer is illegal, I ask for the item to be re-entered. If you don't make it obvious that you want the data entered again, it is possible that the person using your program may not even know that he or she made a mistake and get confused on what to enter next.

The main portion of our design is called DICE-ROLLS; it is responsible for actually playing the game. First, it gets the bets from each player. Then it rolls the dice. Finally, it makes the payments to or collects the losings from each player who is still in the game. Since this code is executed often, I will place it in front of my program. The three main components are called GET BETS, DICE ROLL and PAY/COLLECT. The first two components will be coded as subroutines called from DICE-ROLLS. Line 210 calls GET BETS and line 230 calls DICE ROLL. The third module, PAY/COLLECT, will be coded as part of the DICE-ROLLS module.

Save the Unimportant for Mañana.

Why did I set the modules up this way? The answer to this question requires a little background in the style of coding that I have adopted. As you know by now, I have a number of set methods that I follow. One of these rules of thumb is that if I get a module that I will be expanding or replacing later, I set it up as a subroutine to be coded later. I just code in a GOSUB statement and keep going. If it is a module that has to be coded fully the first time around, I usually code it right then—unless it looks like something that is hard to code. In that case, I code in the GOSUB statement and hold off coding it in until I have to. I write my programs this way because I never want to tackle any code that will destroy my train of thought. After all, one of the reasons we did a design in the first place was to make sure that nothing important would be left out. So if I keep coding, I won't get sidetracked into worrying about the hard parts until I absolutely have to.

Lines 530-560 are used to figure out how many "hits" a player has after the dice are rolled. Notice that this is done using two FOR-NEXT loops, one inside the other. The inside loop in lines 530-560 checks to see if a player bet any of the dice numbers that came up. The outer loop from statements 250-760 controls which player we are looking at. For now, I won't code the full CHECK FOR BANKRUPTCY module. I will instead code a short module (statements 740-750) to check for bankruptcy and STOP the program if there's a loser. Notice how the use of arrays has made this code simple to write. Try to imagine what it would look like if I had to name each variable separately!

The module called END-GAME is also not very important to the main logic of the program, so I'll ignore it for now. This means that the only modules I haven't looked at are GET BETS and DICE ROLL. I coded them in lines 1200-1900 and 2000-2990 respectively. I am leaving a lot of room in the DICE ROLL routine because I still don't know exactly how I am going to do all of it. Oh, I know how to roll the dice, but I haven't gotten around to figuring out what the graphic display of the dice and the design of the screen will look like. . .and until I do, I can't really figure out all of the "hows" of this module. For now, I will code the DISPLAY routine to just show what the dice are.

In order to simulate rolling the dice, I will have to create three random numbers between 1 and 6. This is done using the RND function in statement 2110. Remember that RND is really random only when you start your program with a RANDOMIZE statement. We will eventually put this in statement 140. But until I have fully debugged my program, I will leave the RANDOMIZE statement out. Without it, the dice rolls will not be truly random. They will always follow the same pattern from the start of the program. This allows me to replay a game exactly the same way each time, so that if I find a bug and have to correct my code, I can test the corrected code under the same conditions that caused the bug in the first place. With the RANDOMIZE statement in my program, I may never hit the same conditions that caused the bug and won't be sure that I made the right correction.

After coding in these statements, you can find the result in Listing 1. Let's briefly review just what this program can and cannot do. First, it *does* play the game according to the rules of *Chuck-a-Luck*. It will handle the bets of up to 4 players. It will keep track of cash held by each player and declare a loser. Once I have this program debugged, I then have to plan what pieces I want to add next. The program is missing three important features: First, it stops as soon as one player goes bankrupt and it cannot be restarted without rerunning the program. Second, it cannot display the rules. Third, it is boring because it doesn't have any of the graphics and sound features that the TI-99/4 can add to a program to make it interesting.

Once I have written enough code to run at least a "stripped-down" version of the program, I should turn my efforts to debugging it. Only after I was reasonably sure that this version of the program was working properly, would I begin to add more code. I then would add one module at a time and retest. And that will be the subject of *Chuck-a-Luck*, Part 3.

CHUCK-A-LUCK PART 3: "I never make misteaks"

Don't laugh. All too often you find programs with errors as glaring as that in my first sentence. So let's correct it: I never make mistakes! Now, doesn't that sound egotistical? Nobody would have the nerve to say it out loud. But some people who write programs act like they never make a mistake while programming! The best programmers that I have met not only admit that they make coding errors, but they have also developed quick and efficient ways to find these inevitable mistakes—called *bugs* by programmers. As with everything else, we need a good plan—a "debugging" plan—to catch them.

In the last section, we wrote a large percentage of the code required to play the full game. As a matter of fact, the only important module not coded was the graphics routine. So obviously, it's time to bring on the bugs! WHOA! First we have to figure out how to test for the various bugs I KNOW are in there. Before we do this, let's stop and talk about the different types of bugs found infesting even the best programs.

The first bug that must be eradicated is the "Baddus Plannus." This bug hits programs that do everything (according to the design) correctly but don't achieve the desired result or implement all the rules that you originally laid out. For example, as soon as I began testing my original code for *Chuck-A-Luck*, I hit a situation that I had not planned for and which was outside the scope of the rules of the game. In my original list of rules, I said that a player's bet could be from $10 to $50. As soon as I began debugging my program, however, I immediately saw a flaw in the whole idea! If a player bets in anything other than $10 units, he may eventually wind up with less than $10 in his bankroll. In that case, he can't make a minimum $10 bet and yet he isn't bankrupt. When that happens, the player is in limbo and the whole idea of the game falls apart. A major disaster? No, not necessarily. You see, when you have a good design, these kinds of problems can usually be overcome. I could have changed the logic to allow a player to bet only multiples of $10; instead, I just changed the rules so that bets of less than $10 are allowed. You may have noticed that this change is already in the code found in the last section.

Note that I am not ashamed to admit this error. Indeed I expect something of the sort to happen whenever I write a program. So when I set up my debugging plan, the first few items on my list are tests of the rules. These items don't have to be the first things actually tested, but they must be tested by the time we finish debugging.

The second bug that creeps inside programs is the very evil "Baddus Designus." This guy shows up when the code *almost* does what you want. A sure sign that your program has this problem is that it doesn't do everything that you wanted it to. It may mean that you left out some modules needed to get the program running correctly. It could also mean that a piece of code needs more information (or variables) to do its job. In other words, you forgot (or missed) some facts when you were designing your code. This kind of bug is uncovered by making sure that each routine is thoroughly tested and also by ensuring that each routine is tested using different values in the variables.

The third bug is "Baddus Codus." This means that a piece of code doesn't work even though it has all the information it needs. There are a number of reasons for this kind of bug, but they all boil down to three major ones:

1. You didn't write code that TI BASIC or Extended BASIC understands (for example, you typed in misspelled keywords).

2. You don't really know how a particular feature of BASIC works. You expect it to do something that it just won't do. This can hit your code unless you are prepared to check the reference manual for the usage of BASIC statements that you are not thoroughly familiar with.

3. You wrote code that doesn't do the job. The code may be in the wrong sequence (i.e., you are zeroing out a number just before printing it out on the screen), or a piece of code line is missing, or you typed in the wrong variable name, or even keyed in the wrong variable letter. It all boils down to normal human error.

Bug Catching

If you are lucky, TI BASIC or Extended BASIC will catch some of your errors for you. But don't rely on it. The only good way to check for a case of "Baddus Codus" is to look over your code before running it and then carefully watch how your program behaves when you run it.

Since a test plan for each program depends on the particular code and therefore is unique, the best that I can do for you is list some rules to follow when making up your test plan and debugging your programs.

A. List the program and visually check the code. Review your code for incorrect spelling of variables, miscoded statements (i.e., missing double colons between statements in Extended BASIC), and incorrect CALL names. Fix any errors you find immediately. After you have done this, do it again. Then save this copy of your program to disk or tape before you run the program. This will protect your hard-earned code if your computer decides to "eat" your program on the very first test. Label this Version 1.

B. Write down the function of each major module. Under each module, list the range of valid variables. This should be done so that when you begin debugging, you can set up your tests using both the largest and smallest values possible for each module.

C. Set up a test for each major module. Write down what values you will input and what you expect the output values to be. If you don't write it down before you begin your test, you won't really know if a module is working correctly while you are debugging.

D. Decide whether or not you can use the BREAK command to test the module. In many cases, a routine or module can be tested locally. By that, I mean that the module uses only a few variables and that you can set some values for these variables at the start of the module and BREAK at the end. Then you can check to make sure that the results are correct by PRINTing them on the screen when the computer stops at the BREAK point. For example, suppose a routine starts at line 1000 and uses the variables X and Y as input. The routine is supposed to use these values to calculate the variable Z using some formula. You can test this routine locally by adding the following code in the front and back:

1000 BREAK (replaces the REM statement at the start of the routine)

```
        routine
        code is
        here
          .
          .
          .
1100 BREAK
```

Now RUN your program and make X and Y whatever values you want them to be when the program initially stops at line 1000. When you type in CON, the machine will execute your routine and and stop at the second BREAK statement at line 1100. When your program stops, type in PRINT Z so you can look at Z's value. In fact, you may want to add the following program statement after the second BREAK: 1110 GOTO 1000. In this way, the routine will continually repeat so that you can test your code using a number of input variables without the trouble of having to execute the rest of the program each time. That's why I call it a *local* test. Just make sure you remove that extra GOTO statement as soon as you finish testing that module!

Of course this technique isn't possible with all routines, and in some cases, it's not worth the effort. Just keep in mind that it's one debugging tool that you can use. It also shows a good reason to get into the habit of writing very straightforward code. In a routine, you should try to minimize GOTO statements which take the program outside the routine. If the routine above had GOTO statements that jumped outside the routine, it would be almost impossible to test the routine locally, because you could never be sure that your program would reach statement, 1100. Although program size limitations may force you to reuse code, write all your routines with only one entry point and one exit point if possible.

E. Begin your tests. Carefully note any time that a routine does not give a correct result. Don't stop the program (using the Shift C or FCTN4 keys) each time you notice a problem. Just note the nature of the problem and what the program was doing at the time. For example, if you notice a problem in a routine only when the second player is betting, or if the dice roll is a 6, this is very important information and you should make sure that you write it down. Wait until you have uncovered a number of problems or until the computer stops with a BASIC error message.

F. Check each routine where an error occurred. Mentally "walk" through the code by doing each instruction or calculation on a piece of paper. Usually, you will find your errors this way quite easily. When you locate the error, write down the line number and the solution *but don't key it in!* This is because as soon as you change any of the code in a program statement, BASIC will reset all of the variables to 0 (for numbers) or empty (for strings). This may make it impossible to debug some other routine during the same test run. If you cannot find the bug by walking through the code, look at any intermediate results that may be available by PRINTing any intermediate variables. You may be able to find your mistake this way. This works especially well in complex code with a lot of intermediate totals.

G. If you get to a very difficult spot where the code looks OK, but you are sure it contains an error, don't panic! Use the BREAK *xxx* command, where *xxx* is an actual line number. This allows you to stop the program every two or three lines. At every BREAK, PRINT the important variables, and write them down along the line number of the BREAK. Then type in another BREAK *xxx* command, using a line number two or three lines further along. Type in CON and wait for the program to stop again. You can usually narrow the problem down to a single line this way. If you can't find a misspelling or other typographical error, re-enter the program line very carefully when you have finished this round of debugging. This will likely fix the error (as long as the code you are entering is good code).

H. When you have gone as far as you can in this test, fix all the bugs that you have discovered. Check off any of the tests that have successfully been concluded.

I. Save this new version of your program to disk or tape. I usually have a version number in a REM statement in the front of my programs. I increase this version number every time I change my code. This allows me to know what version of the program I have read into the machine when I begin my tests the next day. If you are saving to cassette tape, make sure you label the tape with the new version number. If you are using a disk, you may want to add the version number as part of the program name (i.e., SAVE DSK1.CHUCKV3). Making the version number part of your SAVE routine can save you some agonizing problems. There is nothing worse than realizing that you are debugging the same code that you fixed the day before.

J. As your program runs, review its actions against the rules and requirements that you originally set up when you began your plan. See if the results are what you expected. If they aren't, immediately stop testing and try to figure out why. You may have to change the rules. You may even have to redesign part of your program. It isn't worth testing any more until you fix this kind of problem.

K. If you get an idea to improve your program, write it down. Don't stop testing to make minor improvements. You may overlook a major flaw while adding a small feature. Add all of these improvements at one time, and revise your test plan to retest the old code as well as test the changes.

After my initial debugging, I began to add some of the modules that I left out the first time. The first routine I added checked to see if the game was over. This feature was added in lines 750, 770-890, and 5000-5400. I do this by checking each player's cash balance. If a player has a balance greater than zero, I increase a counter which tells me how many players are still in the game. I also save that player's number. That way, if only one player is left at the end of a round, I know who it is. If the game is over, I check to see if a replay is wanted. I also added the code at 21100-21500 which displayed the rules. I then retested the program to check both that the new modules worked and that they did not cause any damage to the old code in the rest of the program.

In the next section I'll explain how I added the graphics for both the TI BASIC and Extended BASIC versions. For now, you can study and type in the complete TI BASIC game listing that follows.

```
  60 REM **    CHUCK-A-LUCK      **
  70 REM *         TI BASIC      *
  80 REM
  90 REM
 100 DIM DICE_VALUE(3),PLAYER_NAME$(4),
     PLAYER_CASH(4),PLAYER_BET(4),PLAYE
     R_DICE(4)
 110 DIM DICE_PIP(9,9),LOC_X(27),LOC_Y(
     27)
 140 RANDOMIZE
 160 GOSUB 20000
 170 REM BETTING LOOP
 200 REM GET BET
 210 GOSUB 1200
 220 REM THROW DICE
 230 GOSUB 2000
 240 REM UPDATE CASH BALANCE
 250 FOR I=1 TO PLAYERS
 260 IF PLAYER_CASH(I)=0 THEN 760
 280 PRINT "":PLAYER_NAME$(I);", YOU BE
     T ON";PLAYER_DICE(I);"FOR";PLAYER_
     BET(I);"DOLLAR";
 290 IF PLYER_BET(I)<2 THEN 310
 300 PRINT "S";
 310 PRINT "."
 520 WIN=0
 530 FOR J=1 TO 3
 540 IF PLAYER_DICE(I)<>DICE_VALUE(J)TH
     EN 560
 550 WIN=WIN+1
 560 NEXT J
 570 IF WIN=0 THEN 690
 580 WIN=WIN*PLAYER_BET(I)
 590 PRINT "YOU ";"WIN";WIN;"DOLLAR";
 600 IF WIN<2 THEN 620
 610 PRINT "S";
 620 PRINT "."
 630 PLAYER_CASH(I)=PLAYER_CASH(I)+WIN
 640 PRINT "YOU NOW HAVE";PLAYER_CASH(I
     );"DOLLAR";
 650 IF PLAYER_CASH(I)<2 THEN 670
 660 PRINT "S";
 670 PRINT "."
 680 GOTO 760
 690 PRINT "YOU LOST";PLAYER_BET(I);"DO
     LLAR";
 700 IF PLAYER_BET(I)<2 THEN 720
 710 PRINT "S";
 720 PRINT "."
 730 PLAYER_CASH(I)=PLAYER_CASH(I)-PLAY
     ER_BET(I)
 740 IF PLAYER_CASH(I)>0 THEN 640
 750 PRINT "YOU ARE BANKRUPT!"
 760 NEXT I
 770 REM CHECK FOR END OF GAME
 780 GOSUB 5000
 790 IF NO_LEFT>1 THEN 970
 800 INPUT "WANT TO PLAY AGAIN (Y/N)?":
     A$
 810 A$=SEG$(A$,1,1)
 820 IF A$<>"Y" THEN 850
 830 GOSUB 22000
 840 GOTO 200
 850 IF A$<>"N" THEN 880
 860 PRINT "THANK YOU FOR PLAYING.":"":
     ""
 870 STOP
 880 PRINT PL$
 890 GOTO 800
 970 FOR I=1 TO 600
 980 NEXT I
 990 GOTO 200
1200 CALL CLEAR
1210 FOR I=1 TO PLAYERS
1220 IF PLAYER_CASH(I)=0 THEN 1500
1230 ON INT(RND*4+1)GOTO 1240,1260,1280
     ,1300
1240 PRINT "NOW, ";
1250 GOTO 1350
```

```
1260 PRINT "OK, ";
1270 GOTO 1350
1280 PRINT "ALRIGHT, ";
1290 GOTO 1350
1300 PRINT "YOUR TURN, ":
1350 PRINT PLAYER_NAME$(I);","
1360 PRINT "YOU HAVE ";PLAYER_CASH(I);"
     DOLLAR";
1370 IF PLAYER_CASH(I)<2 THEN 1390
1380 PRINT "S";
1390 PRINT ".":"WHAT'S YOUR BET? "
1400 INPUT PLAYER_BET(I)
1410 IF PLAYER_BET(I)<1 THEN 1450
1420 IF PLAYER_BET(I)>PLAYER_CASH(I)THE
     N 1450
1430 IF PLAYER_BET(I)>50 THEN 1450
1440 IF INT(PLAYER_BET(I))=PLAYER_BET(I
     )THEN 1470
1450 PRINT "THAT'S NOT POSSIBLE."
1460 GOTO 1230
1470 PRINT "WHAT NUMBER WILL YOU BET ON
     ?"
1480 INPUT PLAYER_DICE(I)
1490 IF INT(PLAYER_DICE(I))<>PLAYER_DIC
     E(I)THEN 1520
1500 IF PLAYER_DICE(I)<1 THEN 1520
1510 IF PLAYER_DICE(I)<7 THEN 1540
1520 PRINT "TRY AGAIN."
1530 GOTO 1470
1540 NEXT I
1550 RETURN
2000 REM
2010 CALL CLEAR
2020 CALL SCREEN(10)
2030 FOR I=1 TO PLAYERS
2032 IF PLAYER_CASH(I)=0 THEN 2370
2035 GOSUB 28000
2040 ROW=(I-1)*5+1
2050 COL=15
2060 MSG$=PLAYER_NAME$(I)
2070 GOSUB 4900
2100 ROW=ROW+1
2110 COL=15
2120 MSG$="BET"
2130 GOSUB 4900
2150 COL=20
2160 MSG$="$"&STR$(PLAYER_BET(I))
2170 GOSUB 4900
2200 ROW=ROW+1
2210 COL=15
2220 MSG$="CASH"
2230 GOSUB 4900
2250 COL=20
2260 MSG$="$"&STR$(PLAYER_CASH(I))
2270 GOSUB 4900
2300 ROW=ROW+1
2310 COL=15
2320 MSG$="DIE-"
2330 GOSUB 4900
2340 COL=21
2350 MSG$=STR$(PLAYER_DICE(I))
2360 GOSUB 4900
2370 NEXT I
2500 FOR I=1 TO 3
2510 DICE_VALUE(I)=INT(RND*6)+1
2520 NEXT I
2600 REM DISPLAY DICE
2610 FOR I=1 TO 3
2620 CHAR_NO=DICE_VALUE(I)
2630 IF CHAR_NO=1 THEN 2740
2640 IF CHAR_NO=4 THEN 2740
2650 IF CHAR_NO=5 THEN 2740
2660 IF RND<.5 THEN 2740
2670 IF CHAR_NO<>2 THEN 2700
2680 CHAR_NO=7
2690 GOTO 2740
2700 IF CHAR_NO=6 THEN 2730
2710 CHAR_NO=8
2720 GOTO 2740
```

```
 2730 CHAR_NO=9
 2740 REM DISPLAY A DIE
 2750 FOR J=1 TO 9
 2760 K=(I-1)*9+J
 2780 CALL HCHAR(LOC_X(K),LOC_Y(K),96+DI
      CE_PIP(CHAR_NO,J))
 2790 NEXT J
 2800 CALL SOUND(20,1111,0,1166,0,1221,1
      )
 2990 NEXT I
 3800 FOR I=1 TO 400
 3810 NEXT I
 3900 CALL SCREEN(4)
 3910 CALL CLEAR
 3920 RETURN
 4900 FOR Z=1 TO LEN(MSG$)
 4910 CALL HCHAR(ROW,COL+Z+1,ASC(SEG$(MS
      G$,Z,1)))
 4920 NEXT Z
 4930 RETURN
 4990 REM CHECK FOR A WINNER
 5000 NO_LEFT=0
 5010 FOR I=1 TO PLAYERS
 5020 IF PLAYER_CASH(I)=0 THEN 5050
 5030 NO_LEFT=NO_LEFT+1
 5040 LAST_PLAYER=I
 5050 NEXT I
 5060 IF NO_LEFT>0 THEN 5200
 5110 PRINT "NO ONE IS LEFT.":"THE GAME
      ENDS IN A TIE."
 5120 GOTO 5400
 5200 IF NO_LEFT>1 THEN 5400
 5300 PRINT PLAYER_NAME$(LAST_PLAYER);"
      WINS!"
 5400 RETURN
20000 PL$="PLEASE ANSWER THE QUESTION"
20010 CALL CHAR(96,"0000000000000000")
20020 CALL CHAR(97,"0000001818000000")
20030 CALL COLOR(9,2,16)
20050 CALL CLEAR
20090 ROW=12
20100 FOR I=1 TO 9
20110 FOR J=1 TO 9
20120 READ DICE_PIP(I,J)
20130 NEXT J
20140 IF INT(I/2)=I/2 THEN 20170
20150 MSG$="CHUCK-A-LUCK"
20160 GOTO 20180
20170 MSG$="           "
20180 COL=10
20190 GOSUB 4900
20200 NEXT I
20300 CNT=0
20310 FOR I=1 TO 3
20320 FOR J=1 TO 3
20330 FOR K=1 TO 3
20340 CNT=CNT+1
20350 LOC_Y(CNT)=J+I*4
20370 LOC_X(CNT)=K+2
20400 NEXT K
20410 NEXT J
20420 NEXT I
21000 CALL CLEAR
21010 INPUT "NEED INSTRUCTIONS (Y/N)? ":
      A$
21020 A$=SEG$(A$,1,1)
21030 IF A$="Y" THEN 21100
21040 IF A$="N" THEN 22000
21050 PRINT PL$
21060 GOTO 21010
21100 PRINT :" ":" ":"WELCOME TO THE GAME
      OF":"   CHUCK-A-LUCK!:"
21110 PRINT "THIS GAME CAN BE PLAYED BY"
      :"2 TO 4 PLAYERS. EACH PLAYER STAR
      TS OUT WITH $500. FOR"
21120 PRINT "EVERY TURN, EACH PLAYER BET
      SFROM $1 TO $50 ON A DICE    VALUE
       FROM 1 TO 6. THREE"
21130 PRINT "DICE ARE THEN ROLLED. EACH
       PLAYER WILL THEN RECEIVE AN AMOUN
      T EQUAL TO THIS BET"
21140 PRINT "MULTIPLIED BY THE NUMBER OF
       TIMES THE VALUE HE SELECTED CAME
      UP. IF NO DIE HAS THE"
21150 PRINT "VALUE SELECTED, THE PLAYER
       LOSES HIS BET. A PLAYER WHO GOES
      BANKRUPT IS OUT OF THE"
21160 PRINT "GAME. THE GAME IS OVER WHEN
       ONLY 1 PLAYER REMAINS. IF NOONE R
      EMAINS, THERE IS NO"
21170 PRINT "WINNER. ":" "
21500 FOR I=1 TO 1000
21510 NEXT I
22000 INPUT "HOW MANY PLAYERS (2-4)? ":P
      LAYERS
22010 IF PLAYERS<2 THEN 22060
22020 IF PLAYERS>4 THEN 22060
22030 IF INT(PLAYERS)=PLAYERS THEN 22100
22060 PRINT PL$
22070 GOTO 22000
22100 FOR I=1 TO PLAYERS
22110 PRINT "PLAYER NUMBER";I;"ENTER YOU
      R"
22120 INPUT "NAME-":PLAYER_NAME$(I)
22140 IF PLAYER_NAME$(I)<>"" THEN 22250
22170 PRINT PL$
22180 GOTO 22110
22250 PLAYER_NAME$(I)=SEG$(PLAYER_NAMES(
      I),1,10)
22310 PLAYER_CASH(I)=500
22320 NEXT I
22330 RETURN
25000 DATA 0,0,0,0,1,0,0,0,0
25010 DATA 1,0,0,0,0,0,0,0,1
25020 DATA 1,0,0,0,1,0,0,0,1
25030 DATA 1,0,1,0,0,0,1,0,1
25040 DATA 1,0,1,0,1,0,1,0,1
25050 DATA 1,1,1,0,0,0,1,1,1
25060 DATA 0,0,1,0,0,0,1,0,0
25070 DATA 0,0,1,0,1,0,1,0,0
25080 DATA 1,0,1,1,0,1,1,0,1
25090 DATA X
28000 T2=700
28005 T=120
28010 CALL SOUND(T,392,1)
28020 CALL SOUND(T,523,1)
28030 CALL SOUND(T,659,1)
28040 CALL SOUND(T,784,1)
28050 CALL SOUND(T,784,1)
28060 CALL SOUND(T,784,1)
28070 CALL SOUND(T,659,1)
28080 CALL SOUND(T,659,1)
28090 CALL SOUND(T,659,1)
28100 CALL SOUND(T,523,1)
28110 CALL SOUND(T,659,1)
28120 CALL SOUND(T,523,1)
28130 CALL SOUND(T2,392,1)
28140 CALL SOUND(1,39999,30)
28150 CALL SOUND(T2,392,1)
28160 CALL SOUND(T,523,1)
28170 CALL SOUND(T,659,1)
28180 CALL SOUND(T,784,1)
28190 CALL SOUND(T,784,1)
28200 CALL SOUND(T,784,1)
28210 CALL SOUND(T,659,1)
28220 CALL SOUND(T,659,1)
28230 CALL SOUND(T,659,1)
28240 CALL SOUND(T,392,1)
28250 CALL SOUND(T,392,1)
28260 CALL SOUND(T,392,1)
28270 CALL SOUND(T2,523,1)
28280 RETURN
```

CHUCK-A-LUCK PART 4: "The Die is Cast"

Before me was the task I had been putting off from the beginning: to plan the graphics for the DICE-ROLL routine.

Because the program had been coded in TI BASIC all along, I coded this routine using HCHAR and VCHAR graphics. It occurred to me however, that the Extended BASIC graphics ability (i.e., sprites) would add a lot to this program. Then I saw that it could be done both ways.

The only problem was that I am not very graphics-oriented. Oh, I do all right, but I am no world-beater at eye-boggling displays. That left me one option: I called for HELP!! and turned to my "Guru of Graphics," Ron Binkowski. You may have seen his name on some fine programs he has written for *99'er Magazine*.

I asked Ron to develop a graphics routine to display dice rolling inside a Chuck-A-Luck wheel. About two weeks later, he called me back with the bad news, "No dice." (Pardon the pun—I just couldn't resist it). Rolling the dice was just too complicated for this program, but Ron did come up with an idea to move them graphically.

Starting to Roll

I reworked Ron's routine so that it could support both SPRITE graphics for inclusion in the Extended BASIC version and HCHAR graphics for TI BASIC. The design indicated that DICE-ROLL needed an initializing routine (to set up some variables) as well as the actual graphics roll itself. I added another module to display each player's name, cash balance, amount bet, and dice value bet.

The DICE-ROLL routine needed three new arrays. Each die can be thought of as a formation 3 pieces high and 3 pieces wide. Each character can have either a dot (*pip*) or be blank. Since there are three dice and each needs 9 characters, we will have to keep track of the locations of 27 characters. The 9 characters for the first die will be numbered 1-9; for the 2nd die, 10-18; and 19-27 for the third die. The array called LOC_X keeps track of the x-coordinates (the horizontal rows) of these characters while LOC_Y keeps track of the y-coordinates (the columns). This means that both of these arrays must be DIMensioned with 27 entries.

The array called DICE_PIP tells whether characters are blank or have a pip for each possible value of the dice. Since there are six possible dice values, each to contain information on nine characters, we will need a two-dimensional array composed of 9 entries for each of 6 possible dice-values.

Arrays are Like Buildings

Remember our discussion about arrays? I said that you can envision an array as a building with a number of rooms on each floor. Well, in a two-dimensional array, the first variable can be thought of as the floor number. The second number is the room on the floor. For example, you can think of DICE_PIP(2,4) as the value located in the 4th room of the second floor of a building called DICE_PIP. For our program it will contain the information about the 4th character (middle row on the left) needed to display a dice roll of 2.

To make the display more interesting, Ron added 3 more dice values. He realized that, depending on how a die fell, the values of 2, 3, and 6 could be portrayed two different ways. The three extra "floors" in DICE_PIP are alternate displays for the values of 2, 3, and 6. This meant the DICE_PIP had to be DIMensioned as (9,9). I added this at line 110. In addition, for the SPRITE version of the routine in Extended BASIC, Ron needed an array to keep track of particular pieces of the die, to determine if they were in position. He called this array LOC, and since there are 27 different pieces, I DIMensioned it at 27 in line 120.

I then added the code in lines 20010-20420 (see Extended BASIC listing starting on page 63) to fill in the data needed for the new arrays. Lines 20100-20200 are used to read in the data for DICE_PIP. Each DATA line (in 25000-25080) describes whether a character of a dice value is supposed to be blank (= 0) or have a pip (= 1). Each line gives the information needed for the 9 characters making up the dice value. Line 25090 is an extra DATA line. TI BASIC usually slows down when it reads the last DATA line in a program, but with an extra DATA statement, it never reads the last line, and never slows down.

In order to simulate the DISPLAY AT function, available only in Extended BASIC, I added a routine to the TI BASIC version in statements 4900-4930 to print whatever was in MSG$ beginning at COL on the row contained in ROW. It runs much faster than the code given in TI's *Programming Aids I* software package because it is restricted to a single row and does no preliminary editing of the message area. In lines 20300-20420, I added the codes to show where each of the 9 characters for each die are to be displayed. In the TI BASIC version, these are actual row and column numbers. In the Extended BASIC version, these contain the dot row and dot column values needed for sprites.

I then coded lines 2000-2370 to display the information about each player on the screen. The new code in 2600-3920 displays the three dice values graphically. Lines 2630-2740 give a 50-50 chance that a dice value of 2, 3, or 6 will be displayed in its alternate format. The 9 characters making up the die are then displayed in the loop in lines 2750-2990 in the TI BASIC version, and lines 2750-3020 in the Extended BASIC version. For TI BASIC, this consists of a simple loop which displays at LOC_X and LOC_Y the appropriate DICE_PIP for each of the nine pieces. After the character on the last die is displayed, I wait a little while and then leave the routine.

Notice that in order to highlight the dice roll routine, I changed the color of the screen and added a little music. My "music jar" of melody listings borrowed from other programs gave up only one piece that remotely matched up with gambling, the "Call to the Post" tune played at the track just before a race. Perhaps you have a more fitting musical phrase.

The sprite version of the display routine is more complex than the HCHAR version. I will go through it very carefully because Ron has some great ideas about controlling sprites. Note that this routine was written with multiple statements on each line. This has to be done to make your BASIC code run as fast as possible when handling sprites. Slow code at this point could make it very difficult to handle them smoothly.

Graphic Routines

If you have an interest in designing Extended BASIC programs with sprites, tracing through the following program will put you well on your way toward your own creative endeavors. All line numbers from this point on will refer to the Extended BASIC version only.

2750-2820 This code figures out the sprite number for each character of the die being displayed and starts it out as a sprite with a random motion. Note that this motion can be either positive or negative so that we get them flying in all directions. We also set the LOC value for that character to zero, to show that we haven't yet moved the character back to its final location.

2840-2920 This routine uses a variable called CNT to keep track of the number of characters moved back to their starting locations. If this number is low enough, we will randomly choose the character we work on. If CNT is 21 or greater, however, we won't choose the character randomly. We'll just look through the LOC array sequentially to find the first character that we haven't yet moved back to its location (i.e., its LOC has a zero in it).

Why is Ron going through the trouble of doing it this way? The answer requires a little thought. Suppose we just randomly kept choosing a figure. By the time 20 or so characters have been reset to the final location, the odds on randomly selecting a good character will then be 7/27 or 26%. The odds on the next selection being a good character will then be 6/27 and they keep getting smaller and smaller. With one character left, the odds on hitting it randomly are 1/27 or less than 4%. As you can see, it is very unlikely that you will hit a good character when only a few are left. To prevent a long wait until the computer randomly locates a good character, Ron set up his code so that the last 7 or so sprites will not be randomly chosen. Of course, he is also checking CNT to see if he has finished with all 27 characters.

2930-2980 This part of the routine takes the selected character, changes its color to black (to highlight it on the screen while we play with it), and freezes it momentarily. That is what the CALL MOTION(#1,0,0) is for. The major problem in sprite handling is that they keep moving at a pretty high speed, while BASIC keeps plodding along with old data. Ron prevents this problem by freezing the sprite before finding its location. This means that he gets *accurate* data via the CALL POSITION code.

After locating the sprite, he computes the velocities needed to move it back to its original (and final) location. The Extended BASIC reference manual talks about row velocities and column velocities, but it doesn't explicitly tell you that you can control the direction of the sprite. For example, if you want to move a sprite at a 45 degree angle, both the row and column velocities must be equal. To move at a 30 degree angle, just make the column velocity equal to twice the row velocity. Ron is using this fact in statement 2960 to figure out how far the sprite is, vertically and horizontally, from where it is supposed to go. He calculates this in MY and MX respectively. He then adds the two to get a value called TOT. The distances can be positive or negative depending on the sprite's location relative to its final position—left or right, above or below.

In order to get a good value of TOT, we have to ignore the signs of the distances. In other words, we don't care if the number is positive or negative, as long as we know its *absolute* value. We find it with the ABS function. By making the row and column velocities a function of both the distance the sprite it has to go (MY or MX) and the TOT value, we can direct the sprite to travel in the right direction. Take a look at the last statement in line 2960. It uses the MAX function available in Extended BASIC. TOT must be a reasonably-sized number because we will divide MY and MX by TOT to get our velocities. Since it is possible for the sprite to be right where it should be, TOT can be zero. If you divide by 0, however, your program will stop with an error. To make sure that TOT has a value of at least one, you would normally code in something like this:

```
 xxxxx TOT=ABS(MX)+ABS(MY)
 yyyyy IF TOT<1 THEN TOT=1
 zzzzz
```

This can be done just as easily with the MAX function, which gives the larger of the two alternatives. In this case, if 1 is greater than the result of the addition, it will return 1. On the other hand, if the result of the addition is greater than 1, it will return that number. Using the MAX function eliminates the need for an IF statement right in the middle of my code. MAX (along with its cousin, the MIN function) is a handy feature of Extended BASIC that can save you a lot of coding trouble. We now use the values that we just computed to set the sprite moving again using a CALL MOTION.

2990-3010 I have also set a new variable (my, we are collecting a whole slew of them now!) called CHK to be equal to zero. This counter will be used to make sure that we don't try the next lines of code more than 10 times before we give up and refigure a new MOTION command. If we haven't tried it more than 10 times, we do a CALL COINC to see if the sprite has reached its goal. If not (HIT=0) we go back and do it again. If the sprite has reached its final location, Ron stops it with a CALL MOTION, and does a CALL LOCATE to make sure it is being stopped exactly where he wants it. This is necessary because a sprite that keeps moving between the CALL COINC and the final CALL MOTION may no longer be in the right spot. He changes the color back to white.

3020-3920 This code checks to see if we finished all the characters and restarts the process if we haven't. It then changes the screen back to green. It also issues a CALL DELSPRITE which clears the sprite characters from the screen.

Protection and Improvement

We have now finished the Extended BASIC version of the code. Our game gets a final debugging and is ready to go! The next step is just some administrative work to make sure that your effort will not be in vain. First, change the REM statement at the beginning of the program so that it says FINAL VERSION as well as the version number. Next, save it on cassette tape or disk. Label the tape or disk with the name of the program, the date, and the version number along with the words FINAL VERSION. Make two copies. If you are saving on tape, make one copy on each side and verify both. Then make another copy on a backup tape. You should always have a backup tape kept separately from your original master copies. Remove the tabs in the back of the tape to prevent accidental erasures. For disks, add the write-protect tab. Make a backup disk. Keep it separate from your regular disks. Then enjoy the fruits of your labor!

```
  60 REM   *** CHUCK-A-LUCK  ***
  70 REM   *    EXTENDED BASIC   *
  80 REM
  90 REM
 100 DIM DICE_VALUE(3),PLAYER_NAME$(4),
     PLAYER_CASH(4),PLAYER_BET(4),PLAYE
     R_DICE(4)
 110 DIM DICE_PIP(9,9),LOC_X(27),LOC_Y(
     27)
 120 DIM LOC(27)
 130 RANDOMIZE
 160 GOSUB 20000
 170 REM BETTING LOOP
 200 REM GET BET
 210 GOSUB 1200
 220 REM THROW DICE
 230 GOSUB 2000
 240 REM UPDATE CASH BALANCE
 250 FOR I=1 TO PLAYERS
 260 IF PLAYER_CASH(I)=0 THEN 760
 280 PRINT "":PLAYER_NAME$(I);", YOU BE
     T ON";PLAYER_DICE(I);"FOR";PLAYER_
     BET(I);"DOLLAR";
 290 IF PLAYER_BET(I)<2 THEN 310
 300 PRINT "S";
 310 PRINT "."
 520 WIN=0
 530 FOR J=1 TO 3
 540 IF PLAYER_DICE(I)<>DICE_VALUE(J)TH
     EN 560
 550 WIN=WIN+1
 560 NEXT J
 570 IF WIN=0 THEN 690
 580 WIN=WIN*PLAYER_BET(I)
 590 PRINT "YOU ";"WIN";WIN;"DOLLAR";
 600 IF WIN<2 THEN 620
 610 PRINT "S";
 620 PRINT "."
 630 PLAYER_CASH(I)=PLAYER_CASH(I)+WIN
 640 PRINT "YOU NOW HAVE";PLAYER_CASH(I
     );"DOLLAR";
 650 IF PLAYER_CASH(I)<2 THEN 670
 660 PRINT "S";
 670 PRINT "."
 680 GOTO 760
 690 PRINT "YOU LOST";PLAYER_BET(I);"DO
     LLAR";
 700 IF PLAYER_BET(I)<2 THEN 720
 710 PRINT "S";
 720 PRINT "."
 730 PLAYER_CASH(I)=PLAYER_CASH(I)-PLAY
     ER_BET(I)
 740 IF PLAYER_CASH(I)>0 THEN 640
 750 PRINT "YOU ARE BANKRUPT!"
 760 NEXT I
 770 REM CHECK FOR END OF GAME
 780 GOSUB 5000
 790 IF NO_LEFT>1 THEN 970
 800 INPUT "WANT TO PLAY AGAIN (Y/N)?":
     A$
 810 A$=SEG$(A$,1,1)
 820 IF A$<>"Y" THEN 850
 830 GOSUB 22000
 840 GOTO 200
 850 IF A$<>"N" THEN 880
 860 PRINT "THANK YOU FOR PLAYING.":"":
     ""
 870 STOP
 880 PRINT PL$
 890 GOTO 800
 970 FOR I=1 TO 600
 980 NEXT I
 990 GOTO 200
1200 CALL CLEAR
1210 FOR I=1 TO PLAYERS
1220 IF PLAYER_CASH(I)=0 THEN 1500
1230 ON INT(RND*4+1)GOTO 1240,1260,1280
     ,1300
1240 PRINT "NOW, ";
1250 GOTO 1350
1260 PRINT "OK, ";
1270 GOTO 1350
1280 PRINT "ALRIGHT, ";
1290 GOTO 1350
1300 PRINT "YOUR TURN, ";
1350 PRINT PLAYER_NAME$(I);","
1360 PRINT "YOU HAVE";PLAYER_CASH(I);"D
     OLLAR";
1370 IF PLAYER_CASH(I)<2 THEN 1390
1380 PRINT "S";
1390 PRINT ".":"WHAT'S YOUR BET? "
1400 INPUT PLAYER_BET(I)
1410 IF PLAYER_BET(I)<1 THEN 1450
1420 IF PLAYER_BET(I)>PLAYER_CASH(I)THE
     N 1450
1430 IF PLAYER_BET(I)>50 THEN 1450
1440 IF INT(PLAYER_BET(I))=PLAYER_BET(I
     )THEN 1470
1450 PRINT "THAT'S NOT POSSIBLE."
1460 GOTO 1230
1470 PRINT "WHAT NUMBER WILL YOU BET ON
     ?"
1480 INPUT PLAYER_DICE(I)
1490 IF INT(PLAYER_DICE(I))<>PLAYER_DIC
     E(I)THEN 1520
1500 IF PLAYER_DICE(I)<1 THEN 1520
1510 IF PLAYER_DICE(I)<7 THEN 1540
1520 PRINT "TRY AGAIN."
1530 GOTO 1470
1540 NEXT I
1550 RETURN
2000 REM
2010 CALL CLEAR
2020 CALL SCREEN(10)
2030 FOR I=1 TO PLAYERS
2035 GOSUB 28000
2040 ROW=(I-1)*5+1
2060 DISPLAY AT(ROW,15):PLAYER_NAME$(I)
2160 DISPLAY AT(ROW+1,15):"BET  $";STR$
     (PLAYER_BET(I))
2260 DISPLAY AT(ROW+2,15):"CASH $";STR$
     (PLAYER_CASH(I))
2350 DISPLAY AT(ROW+3,15):"DIE-  ";STR$
     (PLAYER_DICE(I))
2370 NEXT I
2500 FOR I=1 TO 3
2510 DICE_VALUE(I)=INT(RND*6)+1
2520 NEXT I
2600 REM DISPLAY DICE
2610 FOR I=1 TO 3
2620 CHAR_NO=DICE_VALUE(I)
2630 IF CHAR_NO=1 THEN 2740
2640 IF CHAR_NO=4 THEN 2740
2650 IF CHAR_NO=5 THEN 2740
2660 IF RND<.5 THEN 2740
2670 IF CHAR_NO<>2 THEN 2700
2680 CHAR_NO=7
2690 GOTO 2740
2700 IF CHAR_NO=6 THEN 2730
2710 CHAR_NO=8
2720 GOTO 2740
2730 CHAR_NO=9
2740 REM DISPLAY A DIE
2750 FOR J=1 TO 9
2760 K=(I-1)*9+J
2780 CALL SPRITE(#K,96+DICE_PIP(CHAR_NO
     ,J),16,LOC_X(K),LOC_Y(K),RND*120-6
     0,RND*120-60)
2800 LOC(K)=0
2820 NEXT J
2830 NEXT I
2840 CNT=0
2850 IF CNT<21 THEN 2900
2860 FOR I=1 TO 27 :: IF LOC(I)=0 THEN
     2920
2870 NEXT I :: GOTO 3800
2900 I=INT(RND*27)+1
2910 IF LOC(I)=1 THEN 2900
```

```
 2920 LOC(I)=1 :: CNT=CNT+1
 2930 CALL COLOR(#I,2):: CALL SOUND(-1,1
      10,0,165,1,220,2)
 2940 CALL MOTION(#I,0,0):: CHK=0
 2950 CALL POSITION(#I,Y,X)
 2960 MY=LOC_Y(I)-Y :: MX=LOC_X(I)-X ::
      TOT=MAX(1,ABS(MY)+ABS(MX))
 2980 CALL MOTION(#I,MY*50/TOT,MX*50/TOT
      )
 2990 CHK=CHK+1 :: IF CHK>10 THEN 2940
 3000 CALL COINC(#I,LOC_Y(I),LOC_X(I),20
      ,HIT):: IF HIT=0 THEN 2990
 3010 CALL MOTION(#I,0,0):: CALL LOCATE(
      #I,LOC_Y(I),LOC_X(I)):: CALL COLOR
      (#I,16):: CALL SOUND(-1,1111,0)
 3020 IF CNT<27 THEN 2850
 3800 FOR I=1 TO 400
 3810 NEXT I
 3900 CALL SCREEN(4)
 3910 CALL DELSPRITE(ALL):: CALL CLEAR
 3920 RETURN
 4990 REM CHECK FOR A WINNER
 5000 NO_LEFT=0
 5010 FOR I=1 TO PLAYERS
 5020 IF PLAYER_CASH(I)=0 THEN 5050
 5030 NO_LEFT=NO_LEFT+1
 5040 LAST_PLAYER=I
 5050 NEXT I
 5060 IF NO_LEFT>0 THEN 5200
 5100 PRINT "NO ONE IS LEFT.":"THE GAME
      ENDS IN A TIE."
 5110 GOTO 5400
 5200 IF NO_LEFT>1 THEN 5400
 5300 PRINT PLAYER_NAME$(LAST_PLAYER);"
      WINS!"
 5400 RETURN
20000 PL$="PLEASE ANSWER THE QUESTION"
20010 CALL CHAR(96,"FFFFFFFFFFFFFFFF")
20020 CALL CHAR(97,"FFFFFFE7E7FFFFFF")
20030 CALL COLOR(9,2,16)
20050 CALL CLEAR
20090 ROW=12
20100 FOR I=1 TO 9
20110 FOR J=1 TO 9
20120 READ DICE_PIP(I,J)
20130 NEXT J
20140 IF INT(I/2)=I/2 THEN 20170
20150 MSG$="CHUCK-A-LUCK"
20160 GOTO 20180
20170 MSG$="            "
20180 DISPLAY AT(ROW,14):MSG$
20200 NEXT I
20300 CNT=0
20310 FOR I=1 TO 3
20320 FOR J=1 TO 3
20330 FOR K=1 TO 3
20340 CNT=CNT+1
20350 LOC_Y(CNT)=(J+I*4)*8
20370 LOC_X(CNT)=(K+2)*8
20400 NEXT K
20410 NEXT J
20420 NEXT I
21000 CALL CLEAR
21010 INPUT "NEED INSTRUCTIONS (Y/N)? ":
      A$
21020 A$=SEG$(A$,1,1)
21030 IF A$="Y" THEN 21100
21040 IF A$="N" THEN 22000
21050 PRINT PL$
21060 GOTO 21010
21100 PRINT :"":"":"WELCOME TO THE GAME
      OF":"   CHUCK-A-LUCK!":""
21110 PRINT "THIS GAME CAN BE PLAYED BY"
      :"2 TO 4 PLAYERS. EACH PLAYER STAR
      TS OUT WITH $500. FOR"
21120 PRINT "EVERY TURN, EACH PLAYER BET
      SFROM $1 TO $50 ON A DICE    VALUE
       FROM 1 TO 6. THREE"
21130 PRINT "DICE ARE THEN ROLLED. EACH
       PLAYER WILL THEN RECEIVE AN AMOUN
      T EQUAL TO HIS BET"
21140 PRINT "MULTIPLIED BY THE NUMBER OF
       TIMES THE VALUE HE SELECTED CAME
      UP. IF NO DIE HAS THE"
21150 PRINT "VALUE SELECTED, THE PLAYER
       LOSES HIS BET. A PLAYER WHO GOES
      BANKRUPT IS OUT OF THE"
21160 PRINT "GAME. THE GAME IS OVER WHEN
       ONLY 1 PLAYER REMAINS. IF NOONE R
      EMAINS, THERE IS NO"
21170 PRINT "WINNER. ":""
21500 FOR I=1 TO 1000
21510 NEXT I
22000 INPUT "HOW MANY PLAYERS (2-4)? ":P
      LAYERS
22010 IF PLAYERS<2 THEN 22060
22020 IF PLAYERS>4 THEN 22060
22030 IF INT(PLAYERS)=PLAYERS THEN 22100
22060 PRINT PL$
22070 GOTO 22000
22100 FOR I=1 TO PLAYERS
22110 PRINT "PLAYER NUMBER";I;"ENTER YOU
      R"
22120 INPUT "NAME-":PLAYER_NAME$(I)
22140 IF PLAYER_NAME$(I)<>"" THEN 22250
22170 PRINT PL$
22180 GOTO 22110
22250 PLAYER_NAME$(I)=SEG$(PLAYER_NAME$(
      I),1,10)
22310 PLAYER_CASH(I)=500
22320 NEXT I
22330 RETURN
25000 DATA 0,0,0,0,1,0,0,0,0
25010 DATA 1,0,0,0,0,0,0,0,1
25020 DATA 1,0,0,0,1,0,0,0,1
25030 DATA 1,0,1,0,0,0,1,0,1
25040 DATA 1,0,1,0,1,0,1,0,1
25050 DATA 1,1,1,0,0,0,1,1,1
25060 DATA 0,0,1,0,0,0,1,0,0
25070 DATA 0,0,1,0,1,0,1,0,0
25080 DATA 1,0,1,1,0,1,1,0,1
25090 DATA X
28000 T2=700 :: T=120
28010 CALL SOUND(T,392,1)
28020 CALL SOUND(T,523,1)
28030 CALL SOUND(T,659,1)
28040 CALL SOUND(T,784,1)
28050 CALL SOUND(T,784,1)
28060 CALL SOUND(T,784,1)
28070 CALL SOUND(T,659,1)
28080 CALL SOUND(T,659,1)
28090 CALL SOUND(T,659,1)
28100 CALL SOUND(T,523,1)
28110 CALL SOUND(T,659,1)
28120 CALL SOUND(T,523,1)
28130 CALL SOUND(T2,392,1)
28140 CALL SOUND(1,39999,30)
28150 CALL SOUND(T2,392,1)
28160 CALL SOUND(T,523,1)
28170 CALL SOUND(T,659,1)
28180 CALL SOUND(T,784,1)
28190 CALL SOUND(T,784,1)
28200 CALL SOUND(T,784,1)
28210 CALL SOUND(T,659,1)
28220 CALL SOUND(T,659,1)
28230 CALL SOUND(T,659,1)
28240 CALL SOUND(T,392,1)
28250 CALL SOUND(T,392,1)
28260 CALL SOUND(T,392,1)
28270 CALL SOUND(T2,523,1)
28280 RETURN
```

SPELLING FLASH

TI
BASIC

Spelling Flash will help students review their spelling periodically. This program does not use the Texas Instruments Speech Synthesizer.

Its design incorporates one of the simplest, yet most elemental programming structures: the *loop*. One of the most valuable features of computers is their ability to repeat any task many times over. *Spelling Flash* uses a GOTO statement to form the loop. The program begins, reads a word from its data, presents it to the student, accepts the response, prints a message to the student and then repeats the process. Line 330, GOTO 200, simply sends the program back to line 200, where the process begins again. In this case, the loop (and in *Spelling Flash*, the program) ends when it reads the non-word "ZZZ." Line 210 checks for this *flag*; if the spelling "word" is "ZZZ," it ends the program.

In order to use this program, the spelling words have to be typed into the program as DATA statements. The accompanying listing has a selection of spelling words, starting in line 380, but you can put in words of your choice, of course. If you use more words than are in the listing shown, and in the process generate more DATA statements with more line numbers, you will have to alter the value after THEN in line 210 to reflect the new line number of the END statement.

The words will be read as string variables. They may be entered with separate statements for each word, or several words may be listed in each statement, as long as they are separated by commas. "Words" in this context may, of course, also consist of phrases or names with embedded spaces or other special characters. Such phrases *must* be enclosed in quotes. ZZZ must be the last word in the list of words; if it isn't, the computer will return a data error when it tries to read data that's not there.

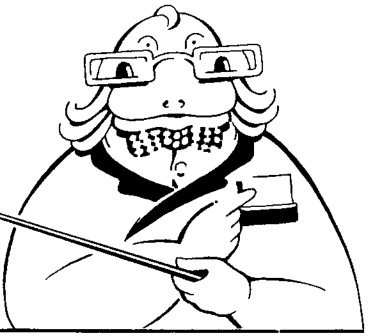
When the program runs, the screen is first cleared and a spelling word is flashed on the screen. After a short delay, the word is cleared and the student is asked to type in the spelling word. The subroutine in lines 340-370 cause the delay; if it seems too long or too short, the value in line 340 can be changed. The student signals that he's finished spelling the word by pressing the ENTER key. The program gives some positive reinforcement with some sounds and the message, "YOU SPELLED IT RIGHT!!" If the word is incorrectly spelled, the student must try again until it is correct.

This section of the program is also a loop. An incorrect spelling sends the program from line 280 back to line 220 until the student gives the correct spelling. After the student has spelled the word correctly, the screen is cleared again and the next word flashes on the screen.

```
100 REM    ****************
110 REM    *              *
120 REM    *SPELLING FLASH*
130 REM    *              *
140 REM    ****************
150 REM
160 REM
170 REM
180 CALL CLEAR
190 PRINT "SPELLING FLASH"::::
200 READ WORD$
210 IF WORD$="ZZZ" THEN 410
220 PRINT "SPELL ";WORD$
230 GOSUB 340
240 PRINT "SPELLING WORD:"
250 INPUT W$
260 IF WORD$=W$ THEN 290
270 PRINT ::"SORRY-PLEASE TRY AGAIN"::
280 GOTO 220
290 CALL SOUND(500,-1,2)
300 CALL SOUND(500,-2,2)
310 PRINT ::"YOU SPELLED IT RIGHT!!"
320 GOSUB 340
330 GOTO 200
340 FOR DELAY=1 TO 800
350 NEXT DELAY
360 CALL CLEAR
370 RETURN
380 DATA TIGER,BEHAVE,BEGIN,"JOHN ADAM
    S"
390 DATA TOMORROW,OZONE,SPRIG,GOOSE
400 DATA CREAM,COVER,AROMA,ABATE,ZZZ
410 END
```

This program may be saved on cassette tape for the students' daily use. Each week, you can alter the list of spelling words by changing the DATA statements.

Pocket Typing Trainer



Pocket Typing Trainer

Here is a pocket-sized program for the TI-99/4A—small enough to fit on a 3 × 5 card—that is not only quick to key in, but is also educational, illustrates a powerful technique with random numbers and is fun for all ages. The *Pocket Typing Trainer* asks which characters the user would like to practice, and then plays back an endless series of random five-character groups for him to copy. Two tones rising at the end of the typist's response say "Correct;" two tones descending here mean "Oops." Try it! If you are a beginning typist, start with characters ASDF, the home keys of the left hand. Stop the program with a FCTN 4 (or SHIFT C on the 99/4) keystroke when you can type those four consistently without looking at them, and RUN the program again with ASDFJKL, and so forth . . . . If you are already a typist and you want to practice some of the unusual features of the TI-99/4A keyboard, as well as some of the characters important in BASIC (but not usually part of the typist's repertoire), try the characters "$( )* + -.

You are unlikely to notice it, but the *Pocket Typing Trainer* tends to focus on the characters which the typist is getting wrong—a remarkably sophisticated feature to find in a pocket-sized program—and one which brings me to my next point.

```
100 REM  POCKET TYPING TRAINER
110 REM
120 REM
130 DISPLAY "TYPE IN THE LETTERS YOU W
    ANT TO PRACTICE TODAY"
140 INPUT LETTER$
150 LENGTH=LEN(LETTER$)
160 OUT$=NUL$
170 FOR I=1 TO 5
180 OUT$=OUT$&SEG$(LETTER$,INT(LENGTH*
    RND^3+1),1)
190 NEXT I
200 DISPLAY "      ";OUT$
210 INPUT "      ":IN$
220 IF OUT$=IN$ THEN 320
230 FOR I=1 TO 5
240 L$=SEG$(OUT$,I,1)
250 IF L$=SEG$(IN$,I,1)THEN 280
260 N=POS(LETTER$,L$,1)
270 LETTER$=L$&SEG$(LETTER$,1,N-1)&SEG
    $(LETTER$,N+1,LENGTH-N)
280 NEXT I
290 CALL SOUND(100,131,3)
300 CALL SOUND(100,110,3)
310 GOTO 160
320 CALL SOUND(100,110,3)
330 CALL SOUND(100,262,2)
340 GOTO 160
```

Skewing the Distribution

Line 180 is where OUT$, the random character string, is manufactured a character at a time. It might have been written without the^3, in which case equal segments of the interval from zero to one would be assigned to the characters given by the typist. (Since the 99/4's built-in random generator, RND, generates "uniform random" numbers, every character would have the same chance of being chosen.) With^3, the random numbers are cubed before a character is chosen. Since the numbers are less than 1, they get smaller as they are cubed; this results in many more RND's corresponding to characters at the left end of the LETTER$ string. For example, suppose that LETTER$, the string of characters which the typist wants to practice, has four characters. If RND turns out to be .50001, then the character a bit more than half way down LETTER$ (i.e., the third character) would be the one chosen. But if we cube RND, the result is .12500, which is well within the first quarter of the range from 0 to 1; and the first character is chosen. Perhaps Diagrams 1 and 2 would help to illustrate this more clearly. The *Pocket Typing Trainer* takes advantage of this by moving missed letters to the beginning of LETTERS$ (Line 270).

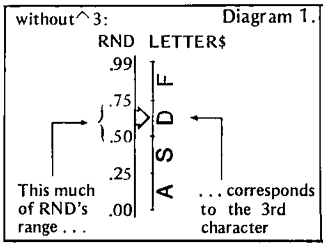


Diagram 1.

without^3:
RND
LETTER$
.99
.75
.50
.25
.00
A
S
D
F
This much of RND's range . . .
. . . corresponds to the 3rd character

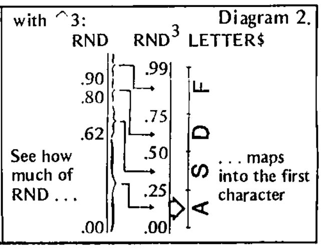


Diagram 2.

with ^3:
RND
RND³
LETTER$
.90
.80
.62
.00
.99
.75
.50
.25
.00
A
S
D
F
See how much of RND . . .
. . . maps into the first character

The lesson here is that uniform random numbers like those provided by RND are a perfectly satisfactory foundation for any sort of randomness one could desire. This includes the statisticians' favorites: Gaussian, binomial, gamma, and so on. One simply needs to apply the proper transformations.

Homework

Tailoring and embellishing programs to suit users' personalities is at least half the fun of computing. The *Pocket Typing Trainer* can be extended in many directions. Here are some of the options:

Problem #1 (simple): Modify the program to allow the typist to choose how many random characters he'd like on the line.

Problem #2 (moderate): Change the program to heighten the emphasis on characters which the typist is getting wrong whenever the error rate is high.

Problem #3 (sound and graphics practice): Keep score, and periodically (say every 25 lines) treat the typist to a colorful and melodic display, one whose elaborateness is greatest for a perfect score.

WHAT IS UCSD PASCAL

And Why Is Everybody Talking About It?

You can hardly pick up a computer publication, or attend a computer conference or fair these days without being inundated with discussion of UCSD Pascal. To understand what all the fuss is about, you must first know something of what is meant by *portability* and understand the concept of *pseudocode* and its relation to the *pseudomachine.*

Portability, Pseudocode, and P-machine

Let's start by assuming that you already know that Pascal is a structured, high-level, compiled language (just as TI BASIC is a high-level interpreted language). In this article we won't go into the theory of compilers, interpreters, or the structure of Pascal as a computer language; we'll save that for a future time. For right now, let's imagine that your friend has written a really great Pascal compiler and operating system in his native 6502 Assembly Language for his Apple computer. You'd like to move it to your fully-configured TI-99/4 which has a TMS9900 microprocessor. What are your options? Sure, you could always recode the Pascal system for your TMS9900 (assuming you had a TMS9900 assembler), but it would probably be almost as much work as starting from scratch. How about first writing a 6502 simulation program for your TMS9900 and letting it re-write all the 6502 code? But even if you do this, the extra layer in between will result in a loss of speed and a greater memory overhead. This is what the microcomputer community has been up against—virtually *no portability* in moving languages or applications software from one system to another without a major re-working of the code.

Now let's design a hypothetical processor to provide a convenient "home" for Pascal. We'll give it built-in instructions for doing the type of things that the Pascal language likes to do. Let's call this pseudomachine a *p-machine* for short, and configure it to be a simple, idealized stack computer that uses pseudocode, or *p-code*—the native language or machine code for the p-machine.

Great, but where do we go from here? What's the use of a p-machine, and how does it contribute to software portability? Must we throw out all existing hardware and software and start over by giving everyone p-machines? Obviously not. Rather, consider what would happen if we could eliminate the differences between the instruction repertoires of specific microprocessors, so that they all execute an identical p-code. If a p-machine emulator for each CPU were written (in its native assembly language), one of the largest obstacles to portability would be overcome: Software could be written on different computers in a high-level language such as Pascal, then compiled to p-code, and finally "interpreted" for each specific CPU. Since the p-code would be universal, in theory a program written on, say, an Apple could be run without modification on a TI-99/4, if the program consisted entirely of p-code. Score one for portability!

This is, in effect, what has been done in the UCSD Software System. All high-level languages in the system—only one of which is Pascal—are compiled into p-code. One way of looking at it is that the *system software* is not portable at all, because it is always executed on a p-machine. The portability is provided by a p-machine emulator for each host. So when you think of a TMS9900-based system running Pascal, it is really running a simulation of a computer which is running Pascal object programs.

Speed vs. Space: A Tradeoff

What price do we pay for the benefit of portability? The detour through a p-machine often produces slower execution than would native code. But raw execution speed is often overshadowed because p-code is considerably smaller than the corresponding native code—allowing the available memory to store a more capable program. If a program can be represented with p-code that fits entirely into available memory, and using native code requires extensive overlaying, then the p-code version will actually run faster!

For best performance, it is desirable to optimize some portions of a program for space and others for speed. Since the UCSD Pascal System provides communication between an assembly language routine and a Pascal host program, it is possible (with some reduction in portability) to code time-critical routines (usually less then 10% of a program) directly in assembly language. The low-level assembly routine can request access to host program global variables and constants, and can also allocate its own global storage space.

A project is underway at SofTech Microsystems (the firm responsible for the licensing and maintenance of the UCSD Pascal System) to alleviate many of the performance drawbacks of p-code (e.g., speed) without sacrificing portability. Code generators will translate time-critical procedures into native code through an optional step in the compilation process. A code generator will take a complete p-code program as input, and produce, as output, a mixture of unmodified p-code and translated native code procedures. Programs can then be written and maintained entirely in Pascal, with the p-code object version still completely portable. A prototype code generator for the TMS9900 demonstrated that improvement in execution performance compared to interpretive execution has been around a factor of 15! And if we take into account that translated native code for the TMS9900 is about 50% larger then the corresponding p-code, the improvement is indeed significant.

The Operating System

UCSD Pascal is not only a language compiler, but a complete operating system with utilities and libraries. In addition to the Compiler, you have a screen-oriented Editor and a File Manager (or *Filer*). The design philosophy behind UCSD Pascal was to keep users continually informed about the state of the system and the options available in that state. This is done with a prompt line that allows users to select options by typing single-character commands.

The screen orientation of the Editor means that you'll be doing lots of paging instead of scrolling. The editor positions a cursor into the text file being edited and surrounds it with a "window" into that area of the file. When you look at the display screen, you are peering into this window. To modify text, you simply move the cursor to the place where the change is desired and indicate the change. Commands are provided for moving cursor, finding and replacing patterns of text, making insertions and deletions, and copying text from elsewhere and moving it to any position indicated by the cursor. In addition to the powerful text editing commands, special facilities are provided for processing documents—e.g., user-specified left and right margins and auto-indenting to encourage the writing of structured programs. In microcomputer systems without an 80-column display, horizontal scrolling allows users to move the text window left and right to view the entire Pascal page.

When you enter the Filer, you have access to another complete set of commands: (1) *housekeeping commands* such as listing directories, compressing files on a disk, and testing disks for bad sectors; plus (2) *program execution and file manipulation commands* for executing named object programs, invoking (with shortcuts) important system programs, designating files for removal, and renaming or transferring among on-line devices.

The Pascal Compiler translates Pascal programs from a humanly readable text form (source code saved on disk by the Editor) into p-code form (object code) which is saved on disks for future execution. The Compiler is designed to translate the entire contents of a text file in one pass. But unlike the Editor and Filer, it has hardly any interactive commands. You can, however, change certain controls (*directives*) which govern the way in which the Compiler does its work.

Error Handling

A big difference between an interpreted language (such as BASIC) and a compiled language (such as Pascal) is demonstrated in the way syntax and run-time errors are handled: If the Compiler finds a syntax error, it halts and displays an error message (if you've set it to return to the Editor automatically), or prints on the screen a progress display containing copies of the line (and previous line) where the program error was found, as well as the coded number of the syntax error. You can fix the error by returning to the Editor or attempt to compile the rest of the program. In some less drastic conditions, the program will, in fact, compile all the way to the end without the Compiler losing its way.

Run-time (execution) errors also cause all the action to stop. A three-line error message tells you the type of error, the segment and block where it occurred, and how far it is from the beginning of the block (which you convert to the actual line of code). In simple cases, this will be all the help that's needed to pinpoint the error; in more complex cases, you'll have to insert WRITELN statements (the equivalent of PRINT) to determine the values of variables before the program blew up. (There's no convenient BREAK statement as in TI BASIC.)

Additional Language Support

The UCSD Pascal System does, in fact, support additional compiled languages. At present, the FORTRAN-77 and BASIC Compiler are supported directly by SofTech Microsystems (MicroFocus CIS COBOL is also presently running under the UCSD p-System). SofTech also has a Cross-Assemblers Package (a complete set of cross-assemblers generating native code for the Z80, 8080, Z8, PDP-11/LSI-11, 6502, 6800, 6809, and 9900 microprocessors) that allows programming on the host machine of your choice, for the object machine of your choice. Think of the possibilities. . . .

UCSD Pascal and the TI-99/4 Community

Texas Instruments has implemenfed UCSD Pascal in a P-Code Card for the TI Peripheral Expansion System. The P-Code Card contains an operating system called the UCSD p-System and allows access to a variety of languages in addition to Pascal. Besides being a powerful tool for software *developers*, UCSD Pascal in TI's version is also of great importance to software *users:* Users won't have to buy all the software and hardware that software developers need in order to write and debug programs. The simplest configuration for software users requires the TI Home Computer, a monitor or TV set, the TI Peripheral Expansion System, the Memory Expansion Card, the P-Code Card and a cassette drive; software developers will need the Disk Memory System (the Disk Drive Controller and up to three disk drives) as well. Under this two-tier system, a TI-99/4A user will be able to run some very sophisticated and powerful *applications* software with only a minimal investment in the *system* hardware and software.