ENRICH GOOD IDEA BOOKS

TRS-80™
TRS-80
TRS-80
TRS-80
TRS-80
FOR THE BEGINNING BEGINNER

ANOTHER GOOD IDEA BOOK FROM ENRICH THE GOOD IDEA PEOPLE
INSTRUCTIONAL MATERIALS DIVISION OF OHAUS®

STATEMENTS

CLOAD-Loads BASIC program file from cassette.

CLS-Clears the display.

CSAVE-Stores resident program on cassette tape.

EDIT-Puts computer into edit mode for specified line.

END-Ends program execution.

FOR . . . NEXT-Opens program loop.

GOSUB-Transfers program control to the specified subroutine.

GOTO-Transfers program control to specified line.

IF . . . THEN-Tests conditional expression.

INPUT-Inputs data from keyboard.

LET-Assigns value to variable (optional).

LIST-Lists program lines to the video display.

NEW-Erases program from memory.

PRINT-Prints an item or list of items on the display at current cursor position.

PRINT @ n-Prints beginning at n, n = 0-1023.

RANDOM-Reseeds random number generator.

REM-Remarks; instructs computer to ignore rest of line.

RETURN-Returns from subroutine to next statement after GOSUB.

RUN-Executes resident program or portion of it.

This book's unique binding enables the user to prop it up by the computer when in use. Note illustration.

TRS-80
FOR THE
BEGINNING BEGINNER

An easy and helpful introduction to computers and programming

Margaret Steimer

ENRICH/OHAUS
San Jose, California

Graphic Design by Kaye Graphics
Illustrations by Bill Eral
Cover Design by Kaye Quinn

Editorial Direction by Contemporary Ideas, Inc.
Edited by Jim Haugaard, Jana Morris, and Matt Foley

Typography by KGN Graphics

Published by
ENRICH/OHAUS
2325 Paragon Drive
San Jose, CA 95131

For information on rights and distribution outside the U.S.A., please write ENRICH/OHAUS at the above address.

ISBN: 0-86582-121-6
Catalog No. EN79224
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ABOUT THE AUTHOR

Margaret Steimer is a computer professional whose work focuses on enhancing the field of education with the many media advantages that microcomputers offer. Currently with DesignWare, Inc., Ms. Steimer designs and implements courseware for topics ranging from pre-school computer literacy to college level data processing.

She has co-authored *Discover BASIC* (Science Research Associates, Inc., 1982), a high school course in the BASIC programming language. The course consists of a student text/ teacher guide and companion computer programs. It is currently available on several disk-based microcomputers, including the TRS-80 Model III.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the students and staff of Magic Mountain School in Berkeley, for field testing the material, and DesignWare, Inc., for the use of their word processing facilities.

The publisher wishes to acknowledge the following companies for their generosity in furnishing us with photographs for use in this book.

RADIO SHACK, A DIVISION OF TANDY CORP.-*44*
INTEL CORPORATION-*42*
MEMOREX-*47*

TABLE OF CONTENTS

INTRODUCTION

Welcome to the TRS-80 computer! You are probably wondering which key to press first, right? By the time you finish working through the exercises and games in this book, you will know that and much more. The important thing to remember is this: The way to learn about a computer is to get your hands on it! The rest is easy.

Since you are probably a novice, like most, you will need to start at the absolute beginning. This book does just that, starts at the beginning.

Don't be afraid to experiment with the computer; it won't yell at you. Computer programmers have developed an easy method for communicating with the computer. This book will help you learn this method. You will be in complete control and make the computer do amazing things.

This book will not make you an "expert", but it will give you enough information about computers to make you "computer literate". This means that the next time a group of people start talking about computers, YOU will be able to join in.

This book also gives you enough "hands on" experiences so that you are able to write and run your own programs. With this knowledge, we know that you will want to go on and learn about the TRS-80.

Remember—Don't be afraid to experiment. You can't hurt the computer by pressing the keys. This book is meant to guide you as you discover what this computer can really do. Each new step will be given with easy to follow directions. Your effort and curiosity will pay rich rewards. Prop this book up by your computer and let's get started.

HOW TO GET STARTED

Level II BASIC and Disk BASIC

Below are instructions for using Level II BASIC (the kind that's built into ROM) and for Disk BASIC. Within each category, you will find a separate set of directions for the Model I and Model III TRS-80. Find the directions for the kind of computer you have and follow them carefully. You will then be ready to start speaking BASIC with the TRS-80.

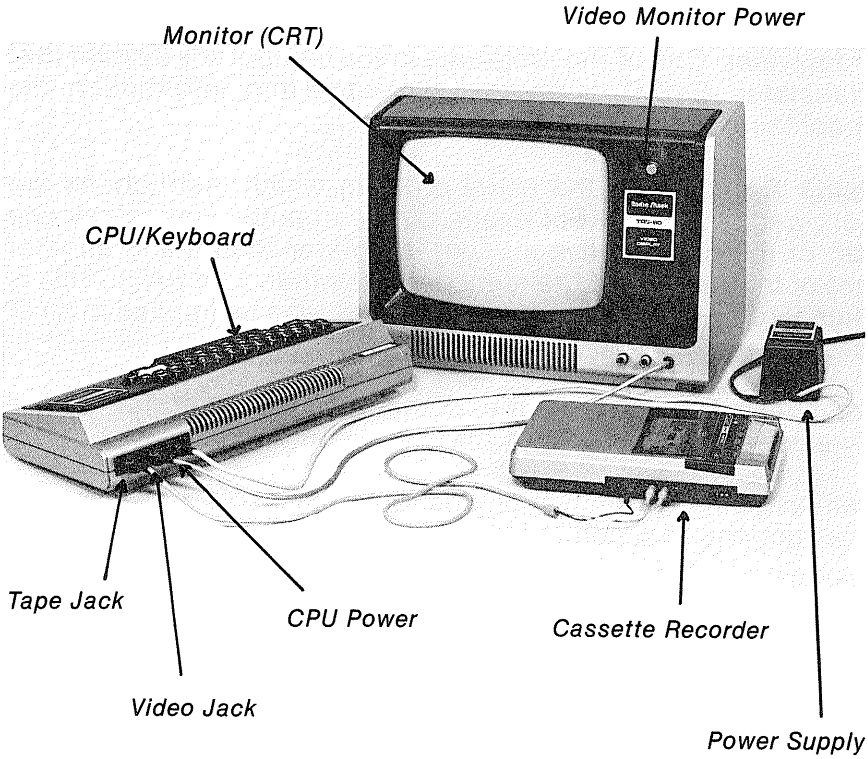
Disk BASIC is used with disk drive systems. It has commands that help you send programs and data back and forth between the computer and the flexible disks you'll use for storage. If you don't have a disk system, you can use a cassette tape recorder to store programs and data. You don't *have* to have a cassette, however, just to use the BASIC that comes with the machine.

After you get to the section on Level II BASIC or Disk BASIC, find the directions for Model I or Model III, whichever you have. There are differences in how the parts of each machine are connected. But after the computer is connected and the power is on, the two models are much the same in operation. What matters more is which kind of BASIC you are using. Whether you're using Disk BASIC or Level II BASIC lets you know which special commands you may use.

Model I-Level II BASIC

Below is a diagram that shows how your Model I TRS-80 should be connected.

MODEL I



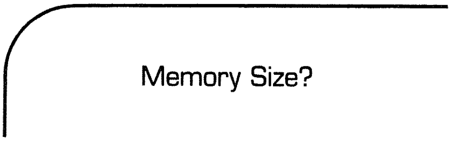
Although you don't need a cassette recorder to use BASIC, the instructions here will include its use. You will eventually want to store, or record, some of the programs you write, and you will need a recorder in order to do so.

If you intend to use a cassette recorder, follow these directions.

1. Make sure the computer is turned off.
2. Locate the cassette recorder connection cable (five-pin connector on one end and three mini-plugs on the other end).
3. Connect the five-pin connector to the cassette jack on the back of the keyboard.
4. Connect the three plug end to the cassette recorder, i.e., black mini-plug to EAR, grey mini-plug to AUX, and grey submini-plug to REM.
5. Connect the AC power cord to an AC outlet.

Whether or not you use the cassette recorder, continue with these directions.

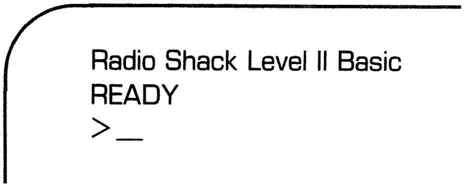
1. Connect the computer AC power cord to an AC outlet.
2. Press the POWER button on the right of the video screen (E in the diagram).
3. Press the ON button on the back of the keyboard (C in the diagram).
4. After a few seconds, the message



```
Memory Size?
```

will appear on the screen. Toward the right on your keyboard is a key labeled ENTER. You press the ENTER key whenever you are finished with an instruction. For now, just press the ENTER key to indicate that you want to use all the computer's Random Access Memory (RAM) for BASIC programs. (Later, you might wish to save part of your RAM for special uses. Then you would need to make a special entry.)

Once you press ENTER, you will see

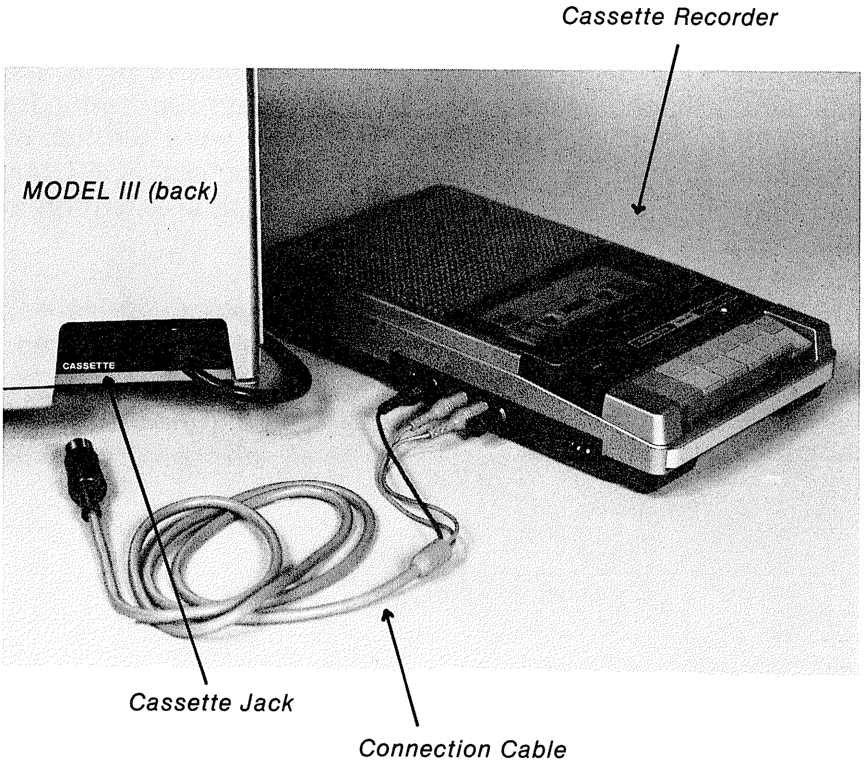


```
Radio Shack Level II Basic
READY
>_
```

The message means that Level II BASIC is now ready for your use. Turn to page 23 to begin.

Model III-Level II BASIC

Below is a diagram that shows where the cassette jack is that connects the Model III TRS-80 to a cassette recorder. The section on Disk BASIC for the Model III shows where the power switch is, along with a sketch of the machine.



Although you don't need a cassette recorder to use BASIC, the instructions here will include its use. You will eventually want to store, or record, some of the programs you write, and you will need a recorder in order to do so.

If you intend to use a cassette recorder, follow these directions.

1. Make sure the computer is turned off.
2. Locate the cassette recorder connection cable (five-pin connector on one end and three mini-plugs on the other end).
3. Connect the five-pin connector to the cassette jack on the back of the TRS-80.
4. Connect the three plug end to the cassette recorder, i.e., black mini-plug to EAR, grey mini-plug to AUX, and grey submini-plug to REM.
5. Connect the AC power cord to an AC outlet.

Whether or not you use the cassette recorder, continue with these directions.

1. Connect the computer AC power cord to an AC outlet.
2. Turn on the computer. The power switch is under the keyboard on the right side.
3. After a few seconds, the message

   ```
   Cass?
   ```

   will appear. This allows you to control how fast the programs and other data are to be transferred to and from the cassette recorder. Type L for slow and H for fast. (If you press ENTER , H will automatically be chosen.)
4. Next the message

   ```
   Memory Size?
   ```

   will appear on the screen. Toward the right of you keyboard is a key labeled ENTER . You press the ENTER key whenever you are finished with an instruction. For now, just press the ENTER key to indicate that you want to use all the computer's Random Access Memory (RAM) for BASIC programs. (Later, you might wish to save part of your RAM for special uses. Then you would need to make a special entry.)

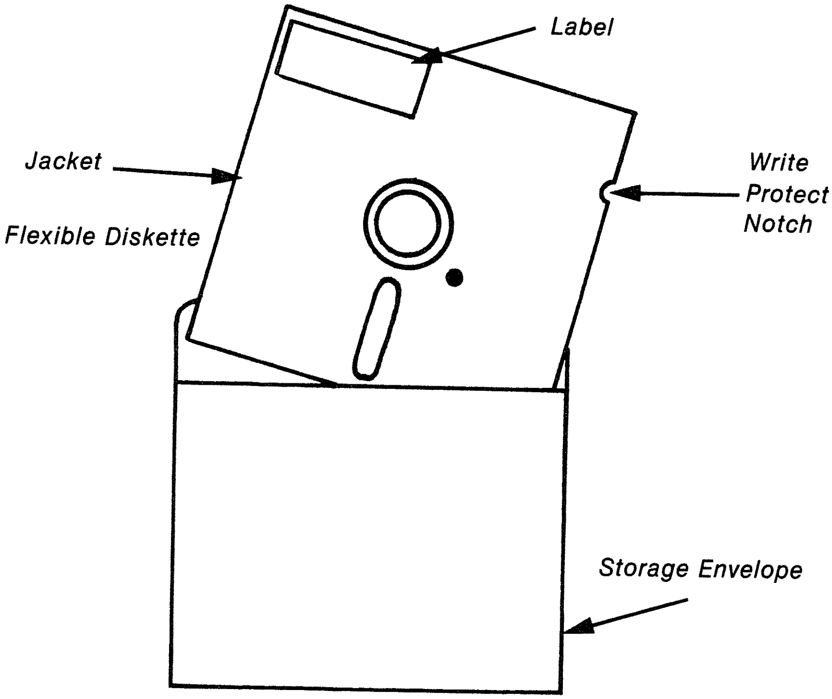
Once you press ENTER, you will see

```
Radio Shack Model III Basic
(c) '80 Tandy
 READY
 > ■
```

The message means that Level II BASIC is now ready for your use. Turn to page 23 to begin.

Disk BASIC

If you have a disk system, then the BASIC you will use is different than the BASIC that comes built in. Disk BASIC is actually an add-on to Level II BASIC. Thus the *disk operating system,* TRSDOS, has to move, or "load," Disk BASIC into memory from a master diskette (at your request, of course).



You use the disk operating system by placing a flexible diskette in the disk drive. TRSDOS is a set of control programs that "keep house" for your computer. The *Disk Operating System/Disk BASIC* manual contains all the commands for using TRSDOS, such as printing a list of all the programs on a diskette or loading certain programs (such as Disk BASIC) into memory. These commands are different from the BASIC instructions you give the computer.

You must see the message

```
DOS READY
```

or

```
TRSDOS READY
```

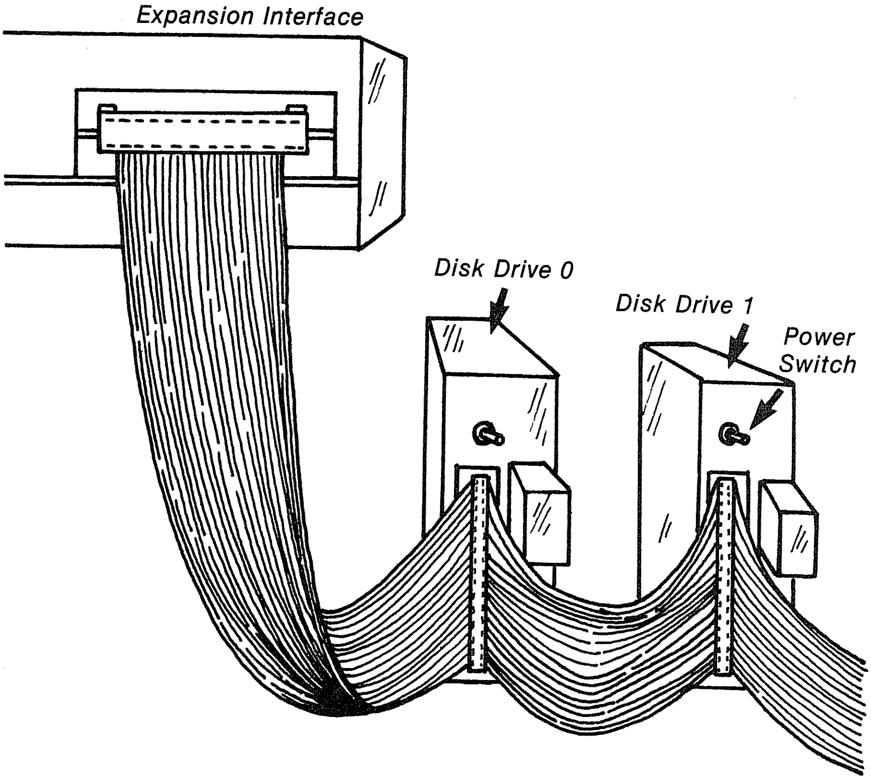
(depending on whether your computer is a Model I or Model III) in order to give a TRSDOS command. When you have Disk BASIC loaded and see the message

```
READY
>■
```

you may give the computer instructions in BASIC.

Model I-Disk BASIC

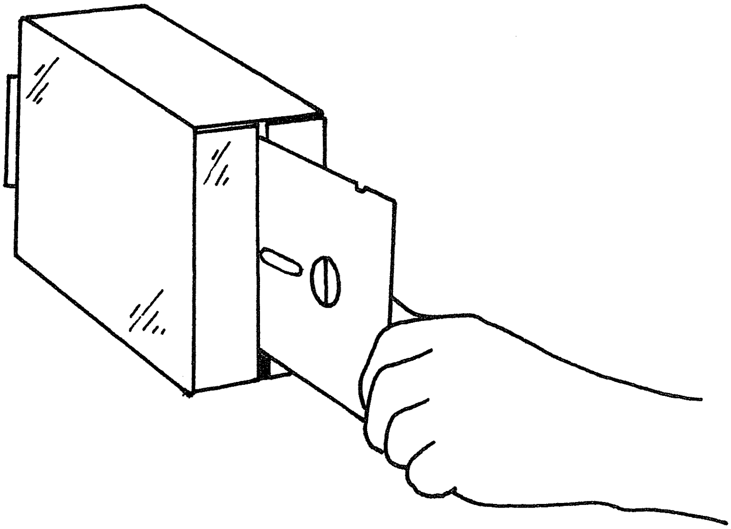
Below is a diagram that shows how the Model I TRS-80 disk drive(s) should be connected.



Follow these steps in order to begin giving BASIC instructions to the computer.

1. Be sure everything is off and there is no diskette in either disk drive.
2. Push the power button, which is in the front center of the Expansion Interface.
3. Turn on each disk drive, starting with Drive 0. The switch is on the back of the disk drive. A red light comes on when the disk drive starts operating.

4. After the red light goes out on the disk drive, insert the System Master diskette into Drive 0, as shown below.



Here are a few things to remember when you handle diskettes.

- Touch only the plastic carrier—never the diskette itself.
- Never let the diskette stay in the sun or other heat source or get cold.
- Always store the diskette in its paper envelope.
- Always wait until the red light is out before you insert or remove a diskette. When the light is on, the diskette is spinning. Moving the diskette while it spins can damage the information recorded on it.

5. Turn on the TRS-80 CPU/keyboard. The power button is in back toward the right.

6. The disk drive light will come on, indicating that the computer is loading the TRSDOS control programs into memory. Once the loading is complete, you will see the message

   ```
   TRSDOS-DISK OPERATING SYSTEM-VER 2.1
   DOS Ready
   ```

   You tell TRSDOS that you want to load Disk BASIC from the system master diskette by simply typing

   ```
   BASIC
   ```

7. You will need to answer two more questions before you are ready to begin. The first is

   ```
   How Many Files?
   ```

   Later, you might want to have a number of data files open, in which case you would type in a number. For now, just press ENTER.

8. You will see

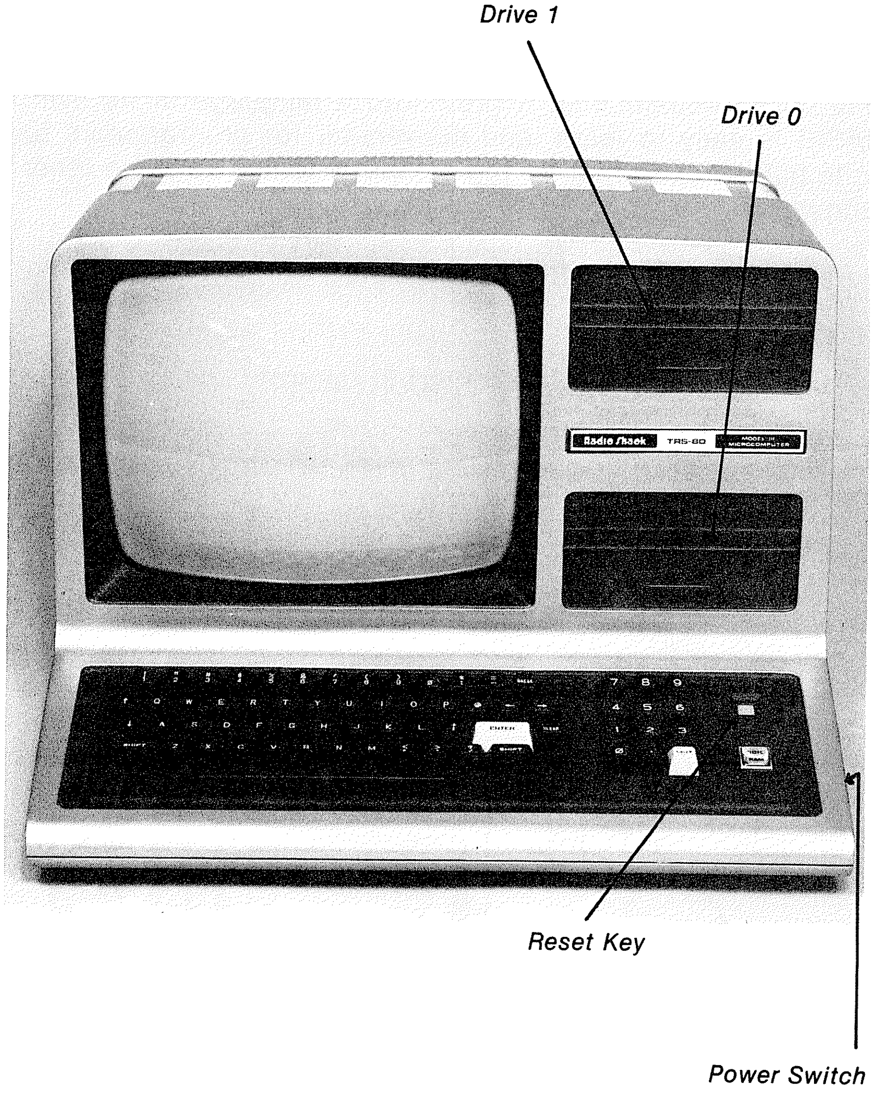
   ```
   Memory Size?
   ```

   Again, just press ENTER in order to tell the computer that you want to use all the random-access memory for BASIC programs. (Later on, you might wish to save part of your RAM for special uses. Then you would need to make a special entry.)

   The "READY" message should appear, indicating that you are now ready to use Model I Disk BASIC. Turn to page 23 to begin.

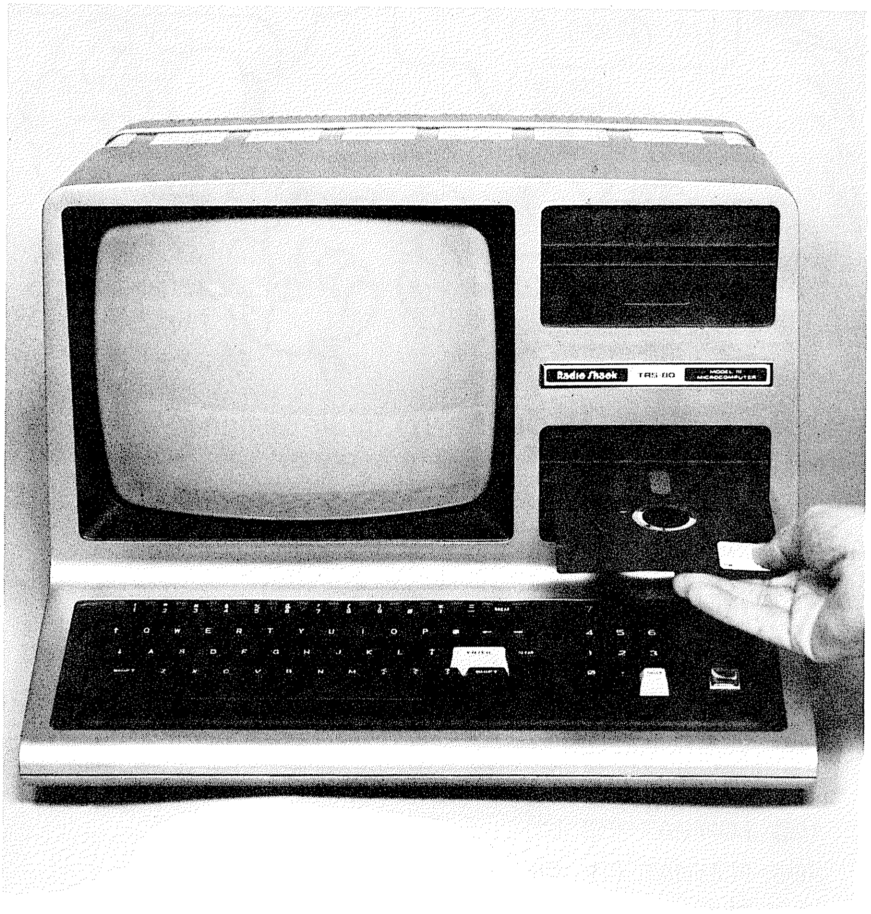
Model III-Disk BASIC

Below is a diagram that shows what's what on your Model III TRS-80.



Follow these steps in order to begin giving BASIC instructions to the computer.

1. Be sure everything is off and there is no diskette in either disk drive. Turn on the computer by flipping the power switch just under the right side of the keyboard. A red light will come on indicating that disk Drive 0 is operating.
2. As soon as the red light goes off, open the door of Drive 0 (the *lower* one, if you have two disk drives) by flipping it up.
3. Insert a System Master diskette, as shown below.



Here are a few things to remember when you handle diskettes.

- Touch only the plastic carrier—never the diskette itself.
- Never let the diskette stay in the sun or other heat source or get cold.
- Always store the diskette in its paper envelope.
- Always wait until the red light is out before you insert or remove a diskette. When the light is on, the diskette is spinning. Moving the diskette while it spins can damage the information recorded on it.

4. Close the disk drive door. Now press the RESET key. The disk drive light will come on again, indicating that the computer is loading control programs into its memory.

5. The computer will ask you for the date and time.

```
Enter date (MM/DD/YY)?
```

Suppose it's March 15, 1984. You would enter the date by typing

```
03/15/84 ENTER
```

After pressing ENTER , this message will appear

```
Enter Time (HH:MM:SS)?
```

Then, if it is 2:30 P.M., type

```
14:30:00 ENTER
```

If you make a mistake, the computer will print a message to try again.

6. The following message will appear on the screen.

```
TRSDOS Ready
```

You tell TRSDOS that you want to load BASIC from the system master diskette by typing

```
BASIC ENTER
```

7. You will need to answer two more questions before you are ready to begin. The first is

```
How Many Files?█
```

Later, you might want to have a number of data files open, in which case you would type in a number. For now, just press ENTER .

8. You will next see

```
Memory Size?█
```

Again, just press ENTER in order to tell the computer that you want to use all the random-access memory for BASIC programs. (Later on, you might wish to save part of your RAM for special uses. Then you would need to make a special entry.)

The "READY" message should appear, indicating that you are now ready to use Model III Disk BASIC. Turn to page 23 to begin.

Let's Begin

Now you are ready to see some results! If you are familiar with a typewriter keyboard, you won't have any problem using the TRS-80's keyboard. They are very similar. If you haven't used a typewriter before, you will want to play around with the letters and numbers. That way, you will quickly learn where to find them. In addition to the letters and numbers, there are keys that are specially for use on the computer. The best way to learn about each of these is to use them as the need arises. As you proceed through this book, you will always find the double asterisk (**) marking computer exercises.

Clearing the Screen

A very important key is the CLEAR key. Whenever the screen becomes filled with characters, simply press the CLEAR key and the screen will be erased. This is very useful for people who like to start with a "clean slate".

Beginning to Type

** You always wanted to see your name in lights, right? Go ahead,type your name! What happens if you press ENTER (on the right side of the keyboard) after your name? You'll most likely see

```
NAME
?SN Error
READY
>■
```

That's a fine way for the computer to greet you! Don't worry, you will soon learn a way to get the computer to remember your name. Notice the flashing square? That is called a cursor.

It lets you know where the next character will be printed on the screen. When you type messages on the keyboard, you will notice that the cursor keeps moving along as each character appears. By pressing ENTER, you tell the computer that you've finished your message. Then, whenever you see the message (called a "prompt")

```
READY
>■
```

you will know that the computer has carried out your orders and is waiting for more.

Typing the Alphabet and Numerals

** First, type the entire alphabet a few times, so that you get used to where each letter is on the keyboard. Second, type the numerals, from 0 to 9. Next hold down the key marked SHIFT while you type the numerals and see what appears on the screen. Notice the characters above each numeral on the keyboard. That's where the strange characters came from. There aren't any characters above the letters of the alphabet. Why are all the letters capitals, instead of lower case? The BASIC interpreter wants its orders only in capital letters. The computer is programmed to create capital letters automatically—without any effort on your part.

Typing Lower Case Letters

** If you want to use some lower case letters, try the following: Hold down the SHIFT key and type Ø (zero,not O)[1]. You didn't see anything print on the screen when you did. But *now* type your name. The computer will now behave just like a

[1] You should first check your reference manual to make sure that your computer can make lower case letters. Some models can't.

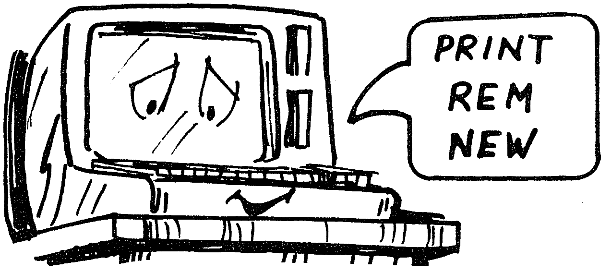
regular typewriter. If you want a capital letter, you must hold down the SHIFT key while you type it. Remember, though, you must use capital letters when you give instructions to the computer. Any time you want to return to all capitals, just press SHIFT and Ø again. It will go back and forth in this way from capitals to lower case. (In computer jargon, the computer "toggles" when you switch back and forth.)

Exploring Further

** Notice what happens if you hold down a letter. How many lines will print on the screen before the computer refuses to print any more? Next, type any letter and then press the key marked → . Repeat this a few times. The → key spaces out, or *tabs,* printed characters across the screen. After you get a few characters on the screen, press the key marked ← ; this is the backspace key. The backspace key allows you to erase mistakes and retype your lines. Even if *you* never make mistakes, you may "change your mind" occasionally.

If you want to change a whole line without backspacing all the way to the beginning, press the BREAK key in the upper right hand corner of the keyboard. BREAK tells the computer, "Forget about the current line. I want to start another one."

Talking BASIC



This is all very interesting, you might think, but the computer *still* doesn't understand what appears on the screen! Every time you press ENTER the same polite ?SN Error appears. That's because you still aren't talking BASIC to the computer.

Try this; suppose your name is Kathy.
** Type this:

```
PRINT "I ONLY TAKE ORDERS FROM KATHY."
```

Congratulations! You typed your first BASIC command. The BASIC word "PRINT" tells the computer to print something on the screen. That "something" is simply whatever you enclose in quotation marks. Now press ENTER and see what happens. Your TRS-80 proudly announces,

```
I ONLY TAKE ORDERS FROM KATHY.
```

But you already saw "I ONLY TAKE ORDERS FROM KATHY" when you typed it in, so why is this special? For this reason: You can combine characters in quotes with information that you store in the computer to make PRINT commands. You store information by using variables. A variable is simply a name that stands for different values at different times.

String Variables

** Tell the computer to store your name in the variable N$. This is how you do it. Type:

```
N$ = "KATHY" ENTER
```

This is called an *assignment* statement, since you are assigning the string variable N$ a value. N$ is called a *string* variable because its value is a "string" of characters. BASIC doesn't care what it finds in a string—numbers, spaces, letters, and punctuation are all acceptable.

** Now you have another way to print your first message. Type the following instruction.

```
PRINT "I ONLY TAKE ORDERS FROM "; N$
```

The semi-colon tells the computer not to start a new line between different things to print. You get the original message as soon as you press ENTER, right? What happens, though, if you assign a new value to the variable N$? Suppose you type:

```
N$ = "NOON UNTIL DINNER TIME."
```

Type the same PRINT message now:

```
PRINT "I ONLY TAKE ORDERS FROM ";N$
```

Most varieties of the BASIC language allow a wide range of variable names. You might use NAME$, ME$, or N2$, for instance, for the variable that holds your name. Just be sure that your string variables begin with a letter and end with a "$".

A Special PRINT Instruction

** You can do more than just print things. You can also tell the computer where to print, anywhere on the screen. To do this, you use a slightly different form of the PRINT command. Type the following:

```
PRINT@775, "HERE I AM!"
```

This instruction tells the computer, "Print the message starting at location 775 on the screen." There are 1,024 places on the screen to print characters. Location 0 is in the upper left-hand corner and location 1023 is in the lower right hand corner. There are 64 characters per row, and 16 rows. Don't worry about where spaces are in this or other commands—BASIC doesn't. The only time BASIC really notices spaces is when they are inside quotes, such as the string above,

```
"HERE I AM!"
```

** Try the following activity. Assign your name to a string variable, such as N$.

```
N$ = "GEORGE"
```

Now think of different numbers from 0 through 1023 and make a print command such as the one above to print your name.

```
PRINT@600, N$
```

Remember, if there is a lot of typing on the screen, you can clear it off by first pressing the CLEAR *key on the right of your keyboard.* Then type in your print instruction. Before you do, though, try guessing where your name is going to show up! Each time you type the instruction, use a different number for the print position.

** How would you get the computer to print the following message in roughly the middle of the screen?

```
*****************************************************
INTRODUCING GEORGE, THE WORLD'S GREATEST PROGRAMMER!
*****************************************************
```

The screen would look much nicer without the faithful "READY" showing up after each line is printed. You can avoid having the message print after each line by putting several PRINT@ commands on one line. *Do this by separating them with a colon.* For example, the above message can be printed with the following line. (Don't press ENTER until you get the entire line typed in.)

```
PRINT@391,"**********************************************
*******";:PRINT@455, "INTRODUCING GEORGE, THE WORLD'S
GREATEST PROGRAMMER!";:PRINT@519,"*********************
********************************";
```

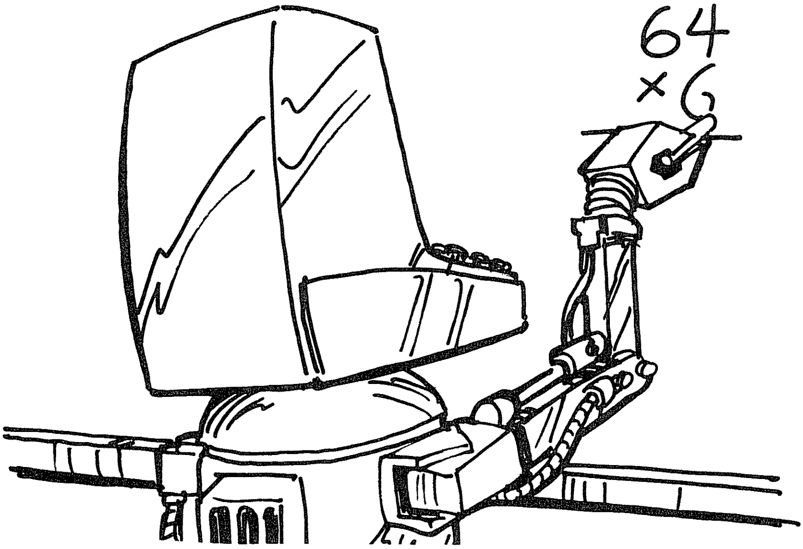
Arithmetic on the TRS-80

Recall that each line has 64 characters. You can use that fact to plan where you want items printed. Notice that the numbers used with the PRINT@ statements in the above exercise are exactly 64 apart; that is 391+64 is 455, and 455+64 is 519. Each string to be printed, then, will begin at the same column on the screen, but on successive rows.

For instance, if you want to type a string of question marks at the start of the sixth line, you can simply type

```
PRINT@64*6,"????????????????"
```

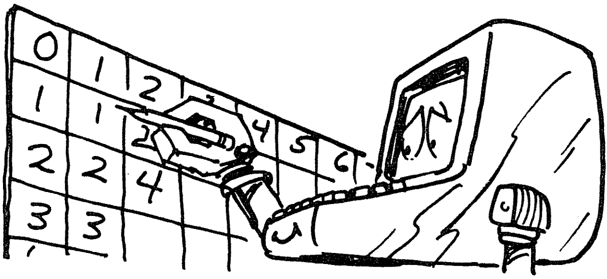
The asterisk (*) in the PRINT@ command tells the computer to *multiply* 64 times 6. Most computer languages use the asterisk to mean "multiply two numbers." Similarly, you tell the computer to divide with a slash (/), to subtract with a minus sign (-), and to add with a plus sign (+). The following exercises will allow you to practice using the arithmetic *operators,* as the symbols are called.



Numeric Variables

You already know about string variables. There is another kind of variable—for numbers. Numeric variables follow the same rules as string variables, except that you don't use the dollar sign and you don't use quotes when you assign the values. As you might expect, numeric variables will only accept number values. Values may be in the form of either numbers or numeric variable names.

** Let's see if the TRS-80 knows its times tables! Try the following.



```
A = 2:PRINT A ENTER
```

This line instructs the computer to assign the value of *two* to variable A and print that value. Now type:

```
A = A * 2:PRINT A ENTER
```

Do this several times.

```
A = A * 2: PRINT A ENTER
```

What is happening to the value of A? You are actually telling the computer, "Multiply the old value of A by two and *store* the result as the *new* value of A. Print the new value." Keep in mind that expressions and numbers may only appear on the *right-hand* side of an assignment statement. The numeric variable is the only thing that should be on the lefthand side. Experiment with some other variables and values.

Here are a few examples.

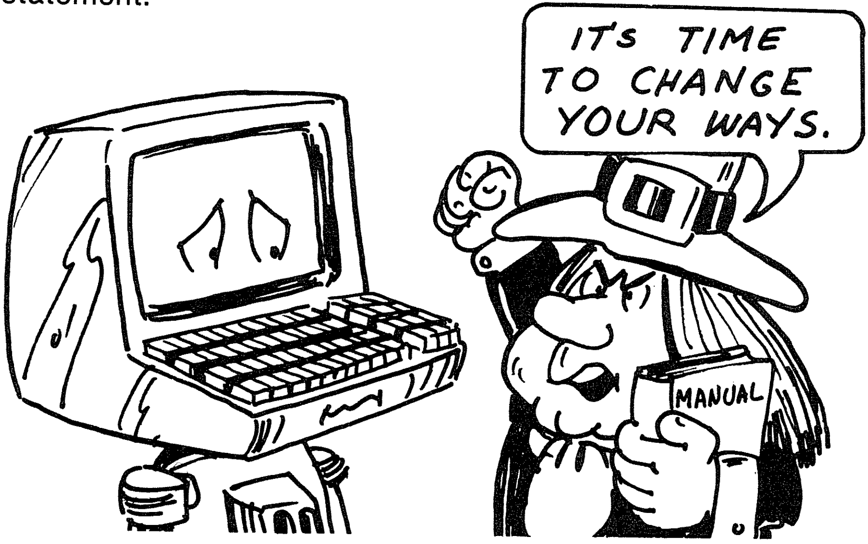
```
X = 1200: PRINT X  ENTER     B = 2345678: PRINT B  ENTER
X = X/2: PRINT X  ENTER      B = B + B: PRINT B  ENTER

K = 33.5: PRINT K  ENTER     S = 1: PRINT S  ENTER
K = K-5: PRINT K  ENTER      S = S*(1/2): PRINT S  ENTER
```

Notice that you can use decimals, fractions, and any combination of operations you want in assigning variables. You can also assign the value of one variable to another, as the following example shows.

```
A = 23:B = A: PRINT A,B
```

(The comma in your print statement will separate the values of A and B with eight spaces, or a *tab.)* Now *both* A and B contain the value of 23. If you then change the value of A to, say, 213, the value of B will *remain* 23. Once a numeric variable value is assigned, it will remain the same as long as the computer is turned on–*unless* you change it with another assignment statement.



Precedence

Computer arithmetic is not very democratic. The computer will go through an expression such as

K = 356–34/5 + 72*8

and do all the multiplying and dividing *first* (from left to right). The result, then, would be

K = 356–6.8 + 576

Only then will the computer go back and perform all of the addition and subtraction. Thus:

K = 925.2

The term *precedence* among arithmetic operations simply means that multiplication and division are performed *before* any addition and subtraction. If you wanted the 356–34 done first in the above example, there is an easy way you could do it. The computer will do any operations first (again from left to right) that are inside parentheses *before* it follows the usual rules. In other words, operations inside parentheses take *precedence* over those outside. The expression above would then be

K = (356–34)/5 + 72*8

which would turn out to be

K = 322/5 + 72*8

or,

K = 640.4

That is quite a difference! Try the two different problems. Remember to type PRINT K after you enter each problem.

Word Problems

Word problems are a familiar part of any math course you might study. Word problems are also an important part of learning BASIC programming. That is why the more practice you get at translating a word problem into a workable arithmetic expression, the better you will become at creating interesting programs. As you probably noticed in the exercises with numeric variables, there are a few differences between normal algebraic expressions and assignment statements in BASIC. Even though the instruction

```
A = A + 3/B
```

looks somewhat like an algebraic equation, it would not make sense in true algebra (since A appears on both sides). In BASIC, though, the expression makes perfect sense. It says "Divide 3 by the current value of B, add the result to the current value of A, and store the final result as the new value of A." The value of B is unchanged.

** Here are two examples of word problems for you to try. The BASIC expressions for the first one are printed here to get you started.

1. Jane's cat had 6 kittens. Her dog had 9 puppies. If she gave away 2 *kittens* and 4 *puppies,* how any *animals* did Jane have left?

```
KITTENS = 6                              (total kittens)
PUPPIES = 9                              (total puppies)
ANIMALS = KITTENS - 2 + PUPPIES - 4      (total animals =
                                         remaining kittens
                                         plus remaining
                                         puppies)
```

Remember, you must instruct the computer to *print* the final value of ANIMALS in order to see the result , i.e.,

```
PRINT ANIMALS
```

2. John built a fence around all four sides of his *yard,* which measures 120 by 82 feet. He left a ten-foot opening in one side for parking his boat, intending to add a *gate* later. How many feet of *fencing* did he need?

(Think of some problems of your own.)

More Numbers

** Experiment a little with large numbers, small numbers, and very long numbers. See what happens if you make the following assignment. Type in the following line and press [ENTER] .

```
BIG = 123456789*123456789
```

Now to print the value of BIG, type this line and press [ENTER] .

```
PRINT BIG
```

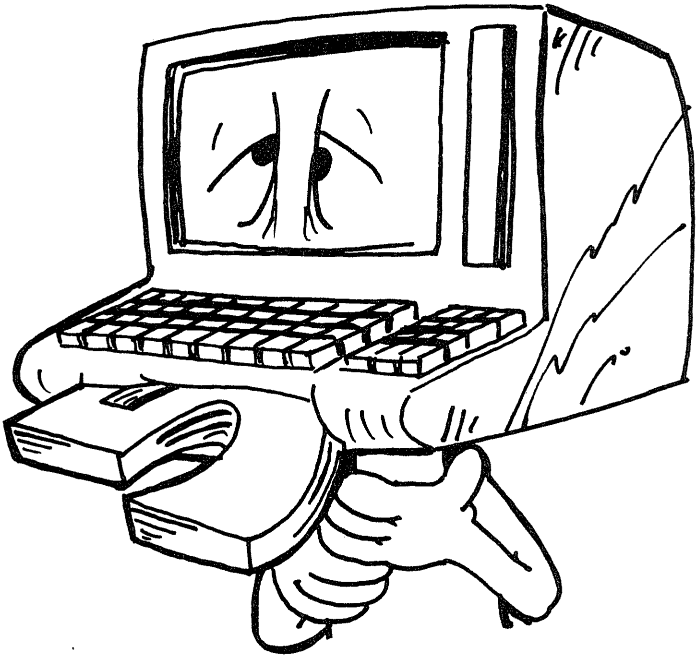
You should see the answer,

```
1.52416E+16
```

appear on the screen. No, the computer did not get mixed up. This is the way *scientific notation* is expressed on the TRS-80. The "E+16" part says that the number in front of it is actually multiplied by 10^{16}. (In case you are not familiar with exponents, 10^{16} means 10 multiplied by itself 16 times.) In other words, the real number is

```
15241600000000000
```

The TRS-80 can handle even bigger numbers. There is a limit, however! You'll know when you reach it, because the computer will gasp, "OVERFLOW".



One way to make big numbers is by using exponents, as mentioned above. For example, in the number 10^{16} in the paragraph above, 16 is the *exponent*. You can tell the TRS-80 to raise 3 to the power of 2 (that is, 3 with an exponent of 2)by typing the following.

```
X = 3↑2
```

Don't worry if you see

```
X = 3[2
```

That is the way the $\boxed{\uparrow}$ key prints on some models.

Now type PRINT X and $\boxed{\text{ENTER}}$.

Graphics

The characters you see on the keyboard are not the only ones you can print. Each character has a code number, which the computer can recognize. This number is usually the same, no matter which computer you are using. This standard code is called ASCII (American Standard Code for Information Interchange). For example, a capital "A" has the code number 65. (Just for your information, a lower case "a" has a *different* code number. That's why it is important to give BASIC its instructions in capital letters.)

Graphics characters on the TRS-80 have ASCII codes of 128 through 191. Some models have special graphics and other characters above 191. The table on page 125 lists the entire "cast of characters" to make TRS-80 graphics, together with their code numbers. To print character 181, just type

```
PRINT CHR$(181) [ENTER]
```

You can use the PRINT@ instruction to have these characters appear anywhere you want on the screen. In order to make character 181 appear in the lower righthand corner of the screen, you would type

```
PRINT@1023, CHR$(181) [ENTER]
```

That is almost all you need to know in order to make pictures on your TRS-80!

** Try printing a character three or four times across the screen. Character 156 looks interesting. See what results if you type the following.

```
PRINT@64*15+10,CHR$(156);:PRINT@64*15+11,CHR$(156);:
PRINT@64*15+12,CHR$(156)
```

It's beginning to look interesting. It's a lot of work, though! Note the colons in the line you just typed. You can use colons to put several complete commands on the same line, such as the three separate PRINT commands. In this case, you wanted to print several characters at once without having the TRS-80 faithfully chime,

```
READY
```

after each one. (Did you notice the semi-colons as well? These, you may recall, keep the computer from automatically starting each print item at the beginning of the next line. This is a handy fact to know when you are a struggling computer artist!)

At this point, you may be wondering, "Isn't there an easier way? What about how *fast* computers are supposed to be?" Yes, there is an easier way! You have seen for yourself that the computer can do many things, one at a time. One of its real charms, though, is that you can tell it do the same thing *many* times. Not only that, it will perform *all* the work you give it before coming back for more orders. One way to do this is by using a *loop.* It is called that because the computer keeps looping, or circling, back to repeat your orders as many times as you tell it to.

** There is an old trick that some teachers like to use. It is called "punishment by boredom." There are many variations, but usually the errant student is given the task of writing something very dull 500 times or more. From that point on, it is a contest between which gives out first—the student's brain or his hand. Remember something we said earlier about the computer never getting bored or tired? Let's prove it.

Type and ENTER the following line.

```
FOR N = 1 TO 500: PRINT "I WILL BE A GOOD COMPUTER": NEXT N
```

By the way, how's *that* for fast? This is the way you tell the computer to loop in BASIC. It tells the computer to count from 1 to 500, printing the message with each count. In fact, you can use a similar loop to cover the entire screen with your name!

Now back to graphics! By using a loop together with the graphics characters, you can create any pattern you wish on the screen.

**Just for starters, try the following line. Then make up your own variations, using different combinations of graphics characters.

```
FOR X =  1 TO 1ØØØ: PRINT
CHR$(153);CHR$(166);CHR$(128);:NEXT X
```

This example uses a plain PRINT statement, but you can also experiment with the PRINT@ form.

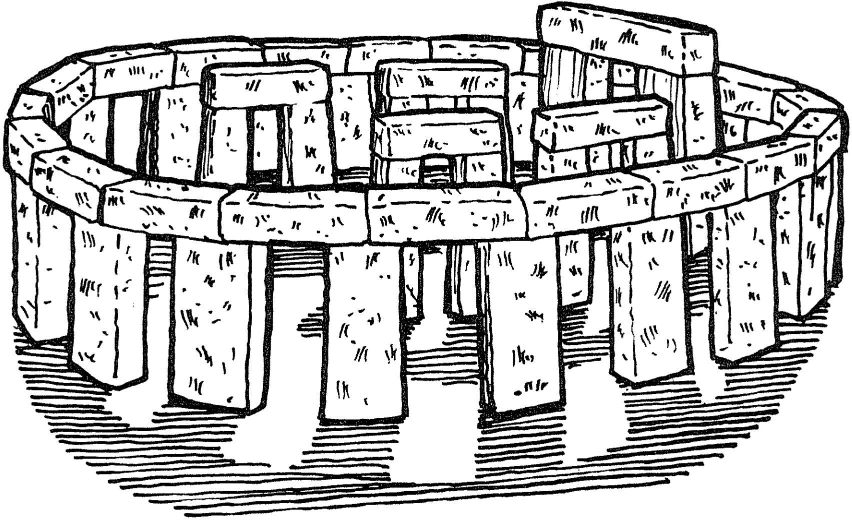
Now you have some of the "basic" ideas on how to make the TRS-80 do things for you. Experiment on your own. See what kinds of pictures you can make with the graphics characters.

In following chapters you will learn how you can animate them!

HARDWARE

Let Me Count The Ways

The word compute means to figure something. People have been computing for thousands of years. Probably the first "computers" were fingers; these "computers" are still in use today. Through the years there have been a lot of other computing devices, such as the abacus and the slide rule. Even Stonehenge, the ancient group of huge stones set in a large circle in the countryside west of London, is said to be a kind of primitive computer. Electronic computers are simply the newest on the scene.



What Is A Computer?

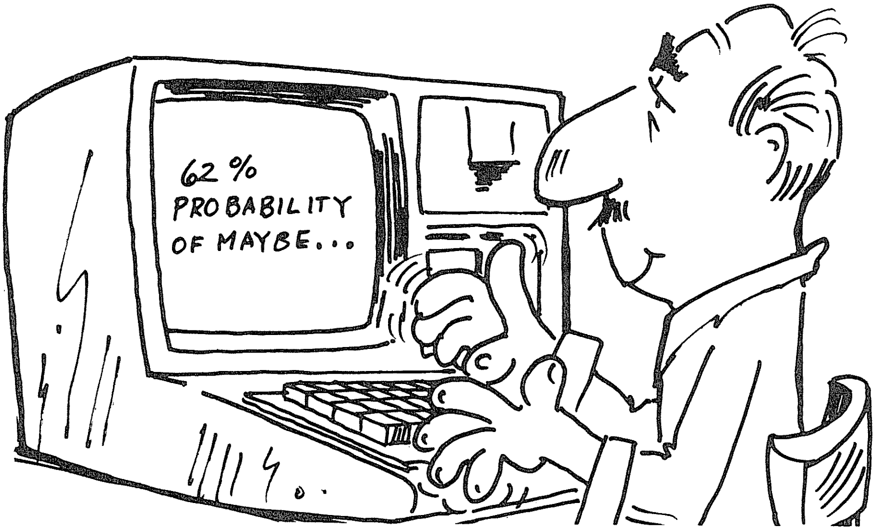
Computers are machines. You can give them information called **"data"** and instructions to do certain things with that information. A computer will follow your orders to do the job and show you the results when you ask for them. Computers also store information in their memory and use it when needed.

Computers, however, can't think or reason in the way people do. A computer can't, for example, collect several pieces of information and draw conclusions. Computers can't combine ideas or take the best parts of several ideas to come up with a brand new idea. This ability to think and reason is reserved *only* for the human brain.

The computer *can* do many of the simpler things our brains can do, but can do them much faster. The computer can give you "yes" or "no" answers if you tell it exactly what data equals "yes" and what equals "no" and as long as it has this data in its memory. The computer can arrange bits of information in order and can sort things out that are alike or find those that are different. A computer's memory can store much information and it never forgets unless you tell it to or turn it off!

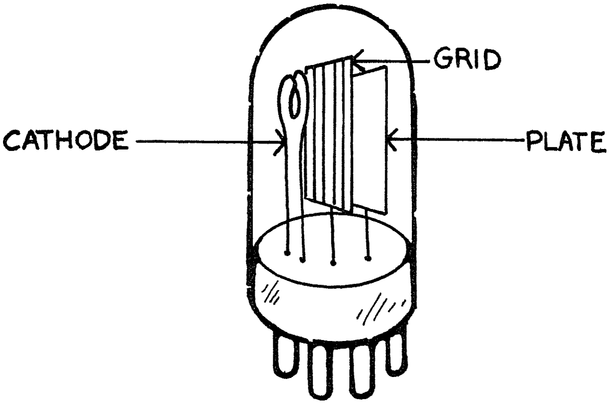
In order for a computer to work a *person* must give it two things:

- information (data)
- instructions on what to do with the information



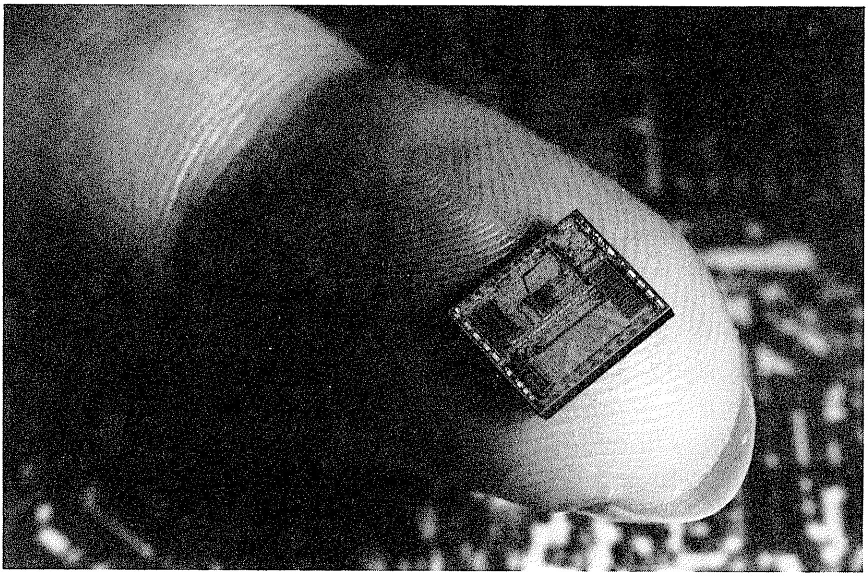
The Family Tree

Computers have been around for four computer generations. The first computer that could be called an ancestor of the present day microcomputer was the Mark I, which was invented in 1944. It is considered to be the first true computer because it could not only do calculations, but also store the instructions for carrying out a group of jobs. In the late 1940's, the first generation of computers in the TRS-80's family tree was invented. These first generation computers were very large, heavy machines in big metal cabinets. They were used mainly by the government to store large amounts of data. They would often take up entire floors of buildings. This first generation of computers was controlled by vacuum tubes like those in old time radios. These tubes were large, some as big as a six year old child. When used, vacuum tubes became very hot and tended to burn out quickly. As a result, vacuum tube computers were not very dependable.



With the invention of the transistor in the early 1960's, the second generation of computers was born. Transistors were smaller, ran cooler, and served the same purpose as the vacuum tube. Second generation computers were smaller and operated up to ten times faster than vacuum tube computers. Second generation computers were also cheaper and more dependable.

Around 1965, an improved computer that was controlled by integrated or printed circuits came on the market. These third generation machines could do a million calculations a second. They were even smaller and more reliable than the second generation machines. Third generation computers are still very much in use in business and government today.

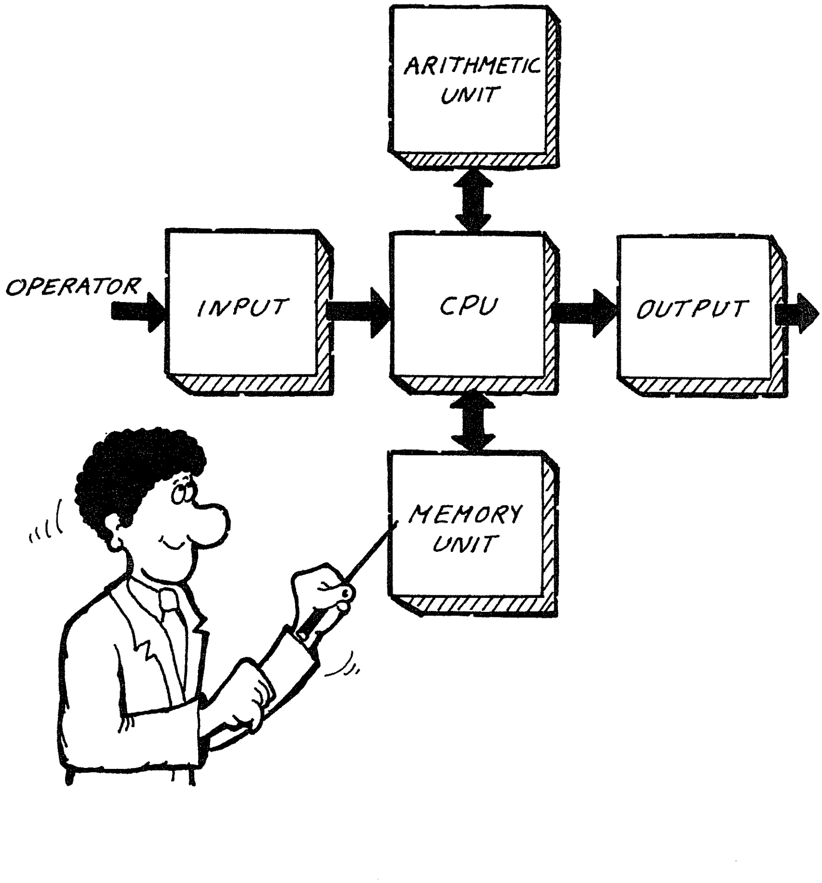


"Personal", or microcomputers, developed about 1976, are the fourth generation of computers. These computers are controlled by microchips. Microchips are tiny circuits etched onto crystals so small that they could pass through the eye of a needle. Each chip can hold as many as one thousand individual circuits. Microcomputers are much smaller, much cheaper, and fifty times faster than the third generation types. The TRS-80 is a fourth generation general purpose computer.

The fifth generation computers are being developed now. They will be controlled by chips with many times the number of circuits found on current microchips. With .this development, you will be able to fit a computer in your pocket that has the same power as one that took up an entire floor.

A Computer Is The Sum Of Its Parts

All computers, whether the large computers used in business or the small microcomputers like the TRS-80, have the same system parts. All parts are needed to make the system work. The five main parts in a computer system are shown in the diagram below.



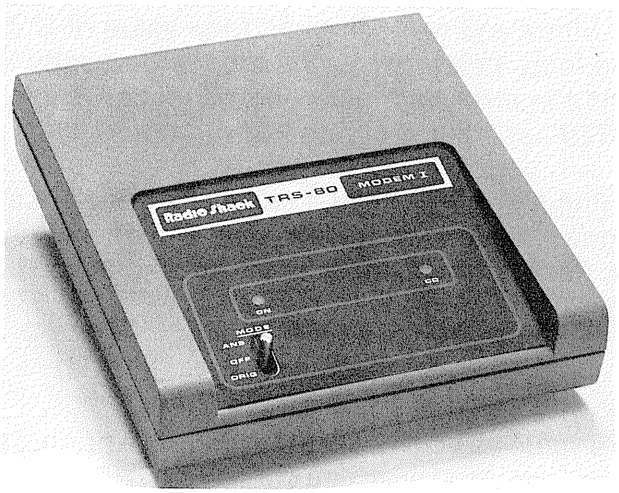
The pieces of equipment that make up these five parts are called **HARDWARE.** The term **SOFTWARE** is also used by computer buffs when speaking "computerese". SOFTWARE is the name given to the instructions that the computer follows.

Input

You may give information to your computer in several ways, just as you can give information to a friend in a letter, a telegram, or on the phone. Machines designed to put information into a computer are called **input** equipment. Examples of input equipment include:

- Card Readers
- Disk Drives
- Keyboard
- Tape Readers
- Program Recorder
- Modem

Large computers may use a card reader, which reads punched cards, or tape readers, which read punched or magnetic tape. They may also use large disk drives that read hard magnetic disks. Information can be typed directly into a computer by using a keyboard, and sometimes a person may even speak to the computer through a microphone unit. Also, a unit called a "modem" that will hold the family telephone, can be connected to your computer. This modem enables your computer to "talk" to another computer at the other end of the line. This receiving computer might be located at the library, the university, a friend's house, or anywhere computers are at work.

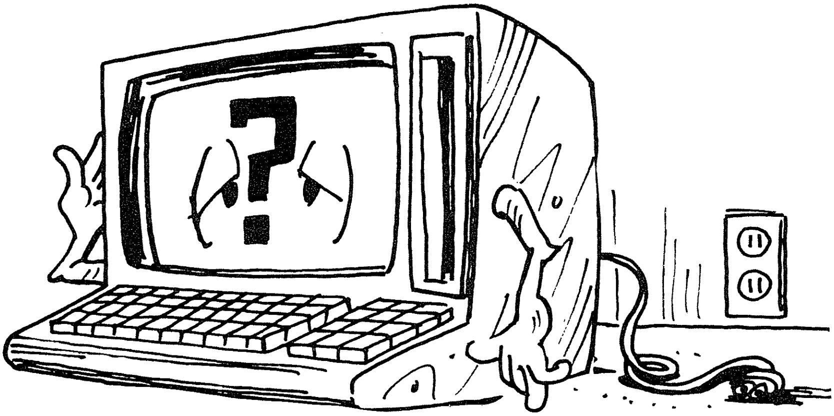


With the TRS-80 Model I or Model III you will use a diskette in the small disk drive, or a cassette tape in the program recorder to "talk" to your computer. When you type on the TRS-80 keyboard, you are also "feeding" information *in*to the computer.

Memory

When you input information to the computer, it goes into the computer's random access memory unit after you press ENTER. The computer stores the information until it is needed. You have a memory unit too; it sits on your neck and has an outer covering of hair (in most models!). Even though the TRS-80 may store as much as 48,000 **bytes** of information at one time, your memory unit is even better and *you* have the ability to think on your own. That's why *you* control the computer.

There are two kinds of memory in the memory unit, the **RAM** and the **ROM.** The RAM (*R*andom *A*ccess *M*emory) has the capacity of storing information sent in through the input equipment. The advantage of the RAM is that you can change it easily to do something new. The disadvantage of RAM is that it forgets everything you have told it when the computer is turned off.

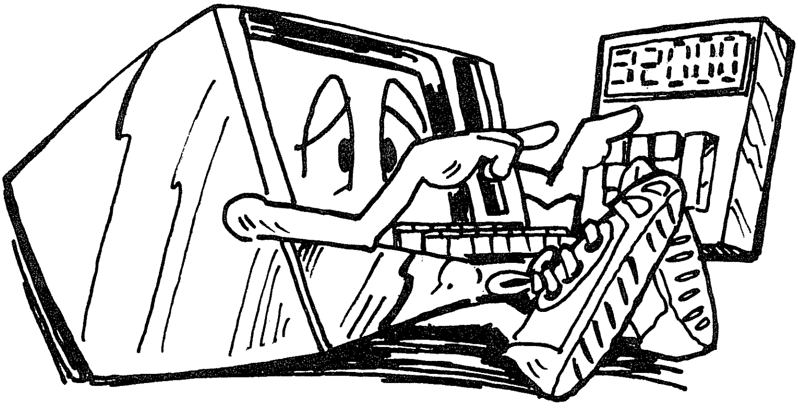


The other type of memory is called ROM. The ROM (*R*ead *O*nly *M*emory) has memory cells that are programmed at the factory with information about the number system, how to load and save information, and all those skills the computer must have to do its work. The advantage of the ROM is that it remembers its instructions even after the power has gone off and on again. The ROM, however, as its name implies, can only be read. It can't be changed by the input equipment and so does not act or react with you. You can see, therefore, that both RAM and ROM are needed as part of the computer's memory.

Three Cheers For The K's

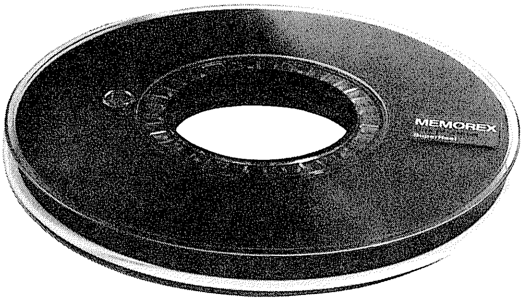
You may hear people talking about the number of "K's" a computer has. They might say, "I have a 16K machine"; or "You have to have a 32K computer to run this program". "K" is a way of showing how large the memory unit is, or how much information the computer can remember at one time. The letter K stands for thousand, just as the "K" in *k*ilogram or *k*ilowatts. A computer that is 32K, for example, can remember 32,000 pieces, or bytes, of information at one time.

The exact number of pieces of information is actually counted by multiples of 1024. The 1000 is just for convenience. Thus, a computer with 32K memory has space for 32,768 pieces, or bytes, of information.



Storing Information Outside The Computer

Often businesses need to save "data" that won't be needed for awhile or that contains old records that must be saved. We do that too when we save photos in an album or put our last year's income tax records in a file box in the attic or basement. Computers can store information on punched cards, magnetic tape, or disks. These "records" can then be stored away in cabinets or files. By doing this, the computer's memory can be erased, making room for work that needs to be done now. Storing records externally is sometimes called *bulk storage*, since a great deal of information can be stored. This bulk storage is much the same as information "stored" in the books in a library or the LP's in your record collection. Bulk storage items can be pulled out of storage and the information they contain can be fed back into the computer's memory for use anytime they are needed.



Control Unit

Once you "input" information into the computer, what happens to it? Information may be sent several different places depending on what needs to be done with it. Examples include sending it to memory for storage, or sending it to that part of the computer that does arithmetic, or combining it with other information already in the computer to become part of a picture on the screen. That part of the computer that decides where each bit of information is sent is the important **CONTROL unit.**

The CONTROL unit is very busy. It directs the movement of all information through the computer, acting as the "traffic director" like a policeman directing traffic at a highway intersection. The control of the flow of information is important, so that no data bumps into any other as it moves through the computer circuits to its destination. Just as two cars might be destroyed or damaged by a collision at an intersection, information that "collides" with other information may be destroyed.

The CONTROL unit also must remember exactly where each bit of information has been sent, so that it can call back that information when it is needed again. All information that is stored in either the RAM or ROM memory is stored in a particular place. This place is called the "address" and identifies the location. The CONTROL unit remembers all these addresses.

The ALU

Another busy and important part of your computer is the **Arithmetic Unit,** which does all the math for the computer. All information entered into the computer's ROM (for example, numbers, letters, punctuation marks) is recorded using numbers. You can see, therefore, how busy the Arithmetic Unit is and how basic it is to the operation of the computer.

The Arithmetic Unit is often referred to as the ALU, which stands for *A*rithmetic *L*ogic *U*nit. The ALU has been "taught" how to do all kinds of arithmetic problems. That is, special instructions have been stored in the ROM memory that can be called on at any time to instruct the ALU. Your TRS-80 is a *DIGITAL* computer. Digital refers to numbers, as in a digital clock or a digital thermometer.

> All the "work" that the computer does is processed by either the Control Unit, the Arithmetic Unit, or a combination of these two. These two partners make up the part of the computer called the *C*ENTRAL *P*ROCESSING *U*NIT, or CPU.

Output

No matter how quickly a computer can follow instructions to do a job,it would be useless if there were not a way to get the information or solution from the computer. The equipment used to retrieve this information so that you can see, hear,or hold it is called *OUTPUT EQUIPMENT.*

Computers usually have a way to connect to a TV or monitor, so that the operator will be able to see what is entered into the computer as well as see the computer's response. Reading the information on the screen is the easiest way to get information from the computer. The game pictures or the information about your checkbook shows up on the screen. In this way, you can see the information entered into the computer by a cassette or a diskette.

If you want a written copy of the **output** from your computer, you can also connect the computer to a printer to record the information on paper. The printer may be a simple electric typewriter which receives information from the computer and automatically types it, or a high speed printer which is able to print the information on paper at speeds as high as 1200 lines a minute. These printers are also considered output equipment. You can also store output information on magnetic disks or tape, and on punched cards or tapes. Most INPUT equipment can also be used as OUTPUT equipment as well. Chapter 3 explains how to "save" information from your computer's memory on magnetic disks or tapes.

Now You Know

You now know how this marvelous machine works. It's really quite simple when you think of its five basic parts. The input equipment, control unit, arithmetic unit, memory unit and output equipment are all ready to work for you. Remember the computer can't think on its own. It must have information and instructions from a *person.*

Crying Over Spilled Milk

The parts of your computer system do not take kindly to bits of cookie, other food, drops of milk, or soda spilled on it. Consider making it a rule that no one can eat or drink while using the computer. An accidentally spilled drink could mean many dollars worth of damage to the TRS-80's insides.

Words of Wisdom

For carefree use in the care and feeding of the TRS-80, review the following hints:

- Protect your diskettes from dust, heat, and electrical appliances.
- If you are using a Model I, turn off power to the computer *first* when shutting down.
- Keep food and beverage away from your computer.

BEGINNING PROGRAMMING

Up to this point, you have given the TRS-80 its orders one at a time. As soon as you press ENTER, the computer carries out an instruction. You know a way to have the computer repeat the same order many times. But what you haven't learned yet is how you can instruct the TRS-80 to carry out *many* orders before coming back for more. That's what people mean when they talk about "computer programs."

What is a Computer Program?

In this chapter, you will learn the first steps toward writing a computer program. You will also learn ways to correct mistakes in lines of your program without having to type them over. A *computer program* is a set of instructions that the computer will save until you tell it to *execute* (or, *run*) them. A program can contain hundreds of instructions, all of which it will follow without any further help on your part. So how do you keep the computer from following orders as soon as you hit ENTER? You do so by using *line numbers.* For example, instead of typing the line

```
PRINT "IS THIS WHAT THEY CALL PROGRAMMING?"
```

you might type this one.

```
1Ø PRINT "IS THIS WHAT THEY CALL PROGRAMMING?"
```

You may not be terribly impressed, but you just wrote your first computer program! Notice that the computer did nothing when you pressed ENTER. That's because it is waiting for you to tell it to *run* the program. Pressing ENTER in this case just tells the computer, "That's all for line number 10." To run the program you simply type RUN. Go ahead, try it!

You may write and run longer programs in the same way. Each line of a program has a line number. The computer will carry out each line in numeric order. In other words, it will execute the line with the lowest number first, then the next lowest, and so on. There are several ways to have the computer go back and repeat one or more lines and to only execute certain ones. You will find more about these methods as you work later exercises. For now, it is enough to practice writing very simple programs that step from one line number to the next.

The Line Editor

Often you won't have to wait until your program fails to discover a mistake. You'll see it right away. If you wish, you can just replace the entire line by typing it over—line number and all. If you discover a line that you don't need at all, just type its line number and press ENTER. That will *delete* the line.

But some lines need painstaking work to type in correctly. It is annoying to have to type the entire line over when only one character is wrong. That's why a handy *line editor* comes with your TRS-80. The line editor allows you to change an existing line—to add or delete as many characters as you like, replace one character with another, list the line, and so on.

Here are a few simple rules for editing. Since the best way to learn is by doing, you should type each of the examples that follow, so that you may see for yourself how the editing commands work. Suppose you have a long PRINT statement, such as the following.

```
3Ø PRINT "THERE IS NOTHIG WRONG WITH THIS LINE"
```

(Sometimes computers tell lies.) You notice that you left out the second N in NOTHING and want to add it. Just type

EDIT 3Ø and press ENTER

and the TRS-80 will respond with

```
3Ø■
```

The line number and cursor let you know that the line editor is ready for your editing commands; in other words, it is in editor *mode.* In editor mode, the computer treats certain letters you type as *commands.* For example, typing the letter I means you want to Insert one or more characters; typing D means you want to Delete. The computer will ignore many letters typed in this mode, because it doesn't recognize them as commands.

** If you have not already entered line 30, then enter it and type EDIT 30. As soon as the computer is ready for you to edit, hold down the space bar. You will see that the cursor moves along characters of your line until you let go. When the cursor reaches the G of the word NOTHIG, release the space bar. To move it one character forward, press the space bar; to move it back one character, press the back arrow. When you insert a character, it will always be above the cursor. Everything else that was originally above the cursor and to the right will be moved to the right.

Now type I to let the computer know you wish to Insert.

```
30 PRINT " THERE IS NOTHI█
```

Type the letter you want—in this instance "N"—and press ENTER. The computer will print the entire line with the added letter.

```
30 PRINT " THERE IS NOTHING WRONG WITH THIS LINE"
```

But perhaps you really want to say, "THERE IS NOTHING WRONG WITH THIS PROGRAM LINE". Just type EDIT 30 again, move the cursor to the place you want to insert PROGRAM, type I and PROGRAM. Then press ENTER. You should see:

```
30 PRINT "THERE IS NOTHING WRONG WITH THIS PROGRAM LINE"
```

** Practice inserting more to your line. Insert THAT I CAN'T FIX. after the word LINE.

Your procedure goes like this:

1. Type EDIT 30.
2. Run the cursor to the ending quotation mark.
3. Type I.
4. Press the space bar once. (Do you know why?)
5. Type THAT I CAN'T FIX.
6. Press ENTER.

If everything went right, your line looks like this:

```
30 PRINT "THERE IS NOTHING WRONG WITH THIS PROGRAM LINE
THAT I CAN'T FIX."
```

To *D*elete a character, move the cursor over to it and type D. To delete more than one character, type the number you want to delete (starting with the one where the cursor is) and *then* type D.

** Type and ENTER the following line.

```
5Ø PRINT "THERE IS SOMETHING ROTTENNN IN DENMARK."
```

Now type EDIT 50. To get rid of the extra N's, use the space bar to move the cursor to the second N and type 2D . What you will see is

```
5Ø PRINT "THERE IS SOMETHING ROTTEN!NN!■
```

No, the computer hasn't added the exclamation points to your line! It is simply showing you which characters it removed by printing them between exclamation marks. Now press ENTER and your line will read:

```
5Ø PRINT "THERE IS SOMETHING ROTTEN!NN! IN DENMARK."
```

Now type RUN and your line will read:

```
THERE IS SOMETHING ROTTEN IN DENMARK.
```

In a similar way, you can *C*hange characters. Typing C will tell the computer, "Change the character over the cursor to the one I type next." If you want to change three characters, type 3C, followed by three new characters.

Below is a table of the editor commands.

Editing Commands

EDIT x	Edit line number x. (Places the computer in edit mode and positions the cursor in front of the first character.)
(space)	Move cursor along line.
I	Insert character(s).
D	Delete character(s); use xD to delete x characters, unless x is one.
C	Change character(s); use xC to change x characters, unless x is one.
L	List line without changing anything.
X	Extend the line.
H	Drop (Hack) off all characters from the cursor and beyond. (You *may* then add new characters if you want to.)
A	Cancel any changes and begin editing *A*gain.
shift- ↑	End this editing command and wait for the next one.
ENTER	Exit editing mode and store the changes made.

Listing Program Lines

Just in case you forget which lines you have in your program, you can have the computer print them for you. Simply type LIST to see all the lines of your program. Watch out, though! If your program is long, it will print all the lines and "scroll" them right off the screen—far faster than you can read them. There are several ways to view only part of your program. One way is to halt the printing every few lines by pressing SHIFT @. To continue, just press ENTER.

You can also look at a small part of a long program by typing a *range* of line numbers you want. For example, if you want to see lines 100 through 230, type

```
LIST 1ØØ-23Ø
```

If you want to list *just* line 230, type

```
LIST 23Ø
```

Error Messages

One last item before you begin programming! Just in case you miss an error in one of your program's lines (line 5Ø, for example), the computer will tell you as soon as it finds the mistake. Your TRS-80 will stop everything and wait for you to correct the offending line. In fact, it will automatically place itself in editor mode at the line with the mistake, printing,

```
?SN ERROR IN 5Ø
READY
5Ø■
```

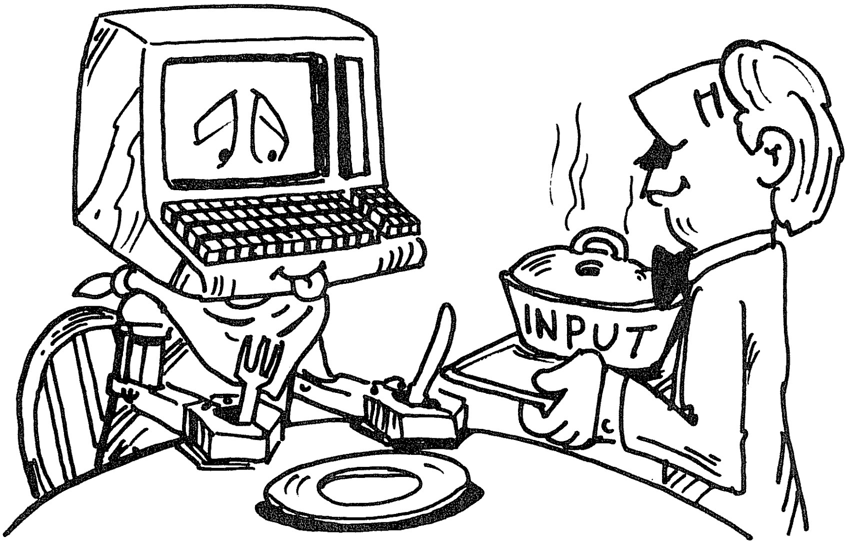
As you learned in the line editor exercises, the TRS-80 will happily print strings containing misspelled words or nonsense. Computers are not so happy with wrong or misspelled BASIC commands and punctuation, however. LSIT seems to be a popular blunder among programmers. Other common *syntax errors,* as these mistakes are called, include typing periods instead of commas, and forgetting quotation marks. You will without doubt make a variety of such errors as you write computer programs—*everyone* does.

When you run a program with an error in it, the BASIC interpreter (a program that translates your instructions for the machine) will halt execution and inform you of which error you have made and the offending line number. Just make whatever change you need to, following the editing directions and press ENTER. Your program is then ready for another try at running.

Writing Programs

It's time to start writing *real* programs now! First, there is a handy BASIC command you should know about. One of the really fun things about programming is getting different results (called *output*) from different information that you give a program (called *input*). You already know about one kind of input. This is the information you give the computer at the time you write the program, using variables and PRINT statements. The only problem with that kind of information is that, except for calculations your program makes, you can't change it—until you change the program.

There is another kind of input that lets you enter information into a program *while* it is running. In fact, the program will refuse to do anything further until you enter that information!



You tell the computer to expect such an entry with the INPUT statement. Much as assignment statements, INPUT statements also use variables. For instance, suppose you have the following lines.

```
2Ø INPUT A$,B
3Ø PRINT A$;B
```

When you RUN your program, the computer will halt as soon as it comes to line 20 and display

?

This means that you are to type a string of characters and then a number, since A$ is a *string* variable and B is a *numeric* variable. So you might type

```
? LIFE BEGINS AT  ENTER
?? 6Ø  ENTER
```

And the program will continue on its merry way. The double question mark (??) means that the computer has gotten *some* of the input it expects, but is not yet completely satisfied. You could, if you wanted, enter both values on the same line, since that's how your INPUT statement is set up. All you need to do is type

```
? LIFE BEGINS AT ,6Ø  ENTER
```

The *comma* tells the computer, "That's all for A$; the next thing I enter will be the value of B." Once you have entered both values, the computer will execute line 30, which should print

```
LIFE BEGINS AT 6Ø
```

Either way you entered it, when you typed in the input, the program stored LIFE BEGINS AT in A$ and 60 in B. The computer can go ahead and use these variable values just as though you had *assigned* them when you wrote the program.

Suppose you entered, instead,

```
? LIFE BEGINS AT ,AN EARLY AGE  ENTER
```

The computer wouldn't be too happy with AN EARLY AGE as the value for the *numeric* variable B. That is just what you attempted to give it! So, you will see a message

```
?REDO
```

?REDO means that you have to enter *both* values again—for A$ *and* B. The computer will patiently give you as many chances as you need to do it correctly.

** Here is a fun exercise that will illustrate how the INPUT statement can be used inside a loop. Recall from Chapter 1 that a loop is written with the following set of BASIC statements:

```
FOR N = 1 TO 10: (Statements to be executed each time) :NEXT N
```

You can use loops quite effectively in programs with line numbers. Try the following program, which will loop 10 times, each time expecting two string INPUTs and printing the results.

```
1Ø PRINT "TYPE A FRIEND'S NAME, THEN A KIND OF FOOD."
2Ø FOR N = 1 TO 10
3Ø INPUT X$,Y$
4Ø PRINT X$;" EATS ";Y$
5Ø NEXT N
```

(Note the spaces on either side of "EATS".)

The results should be amusing, when the computer prints such statements as

```
ANNIE EATS BEANS
FRANK EATS WAFFLES
```

and so forth.

A Customized Story Program

** Here's a fun activity that you can play with your friends. Before you start, however, you must tell the computer that you are finished with the last program and wish to erase it from memory. You can turn the computer off and on again which would wipe any earlier program out of its memory, or you can type NEW and press ENTER. Now the computer knows that you're doing something new. Anytime you type NEW and press ENTER, it clears the program in the RAM memory of the computer. When your friends run your program, they will be asked to enter information; but they won't know how that information will be used until after it has been typed in. You may type in the program shown here—or, you can make one up. You know enough by now to do a lot of experimenting on your own! The following is a listing of the INPUT portion of the program.

```
NEW
5 PRINT "TYPE THE FOLLOWING:"
1Ø INPUT "A FRIEND'S NAME";N$
2Ø INPUT "A NUMBER";X
3Ø INPUT "A TOWN";T$
4Ø INPUT "A LIVING THING";L$
5Ø INPUT "A KIND OF FISH";F$
```

When your program reaches the first of these INPUT statements, it will halt and print

```
A FRIEND'S NAME?
```

and so forth. The computer will print the strings that you entered in front of the question marks, just as though you combined a PRINT and an INPUT statement into one. That way, your friend will know just what kind of input is expected each time. Besides, it's much more pleasant to see the question, A KIND OF FISH?, than a bare question mark.

(Putting helps like these into your program is one way of being "user-friendly," as people who work a lot with computers say.)

Now type in the part of your program that will *use* the input your friend enters. Before you print anything, though, you will probably want to clear the screen. "How can I have the computer press the "Clear" key in the middle of my program?" you might ask. To clear the screen while a program runs, use the simple command CLS. The remainder of the sample program for the activity is printed below.

```
1ØØ CLS: PRINT "THERE WAS ONCE A CLOWN NAMED ";N$;"."[1]
11Ø PRINT N$; "WAS A REAL FLOP AT THE CIRCUS. NOBODY
    LAUGHED!"
12Ø PRINT "ONE DAY THE OWNER SAID, ' " ; N$; " YOU JUST
    AREN'T FUNNY! YOU'VE GOT TO GO."
13Ø PRINT X;" PEOPLE WALKED OUT OF THE SHOW LAST NIGHT!' "
14Ø PRINT "SO ";N$;" SADLY PACKED UP AND TRUDGED ALL THE
    WAY TO ";T$;"."
15Ø PRINT N$;" SAT DOWN AND CRIED AND CRIED. A NOISE MADE
    ";N$;" LOOK UP."
16Ø PRINT "THERE STOOD A BEAUTIFUL ";L$;", LAUGHING. SHE
    EXPLAINED THAT ";N$;" 'S TEARS HAD FILLED THE LAKE AND
    SAVED THE TOWN'S RARE TROPICAL" ;F$;" ."
17Ø PRINT "NOW THEY WOULD ALL BE RICH! ";N$;" MARRIED
    THE ";L$;", AND THEY ALL LIVED HAPPILY EVER AFTER."
18Ø PRINT "MORAL: SUCCESS IS NO LAUGHING MATTER!"
```

The best part about a program like the one above is that your friend won't know how the INPUT information will be used until *after* it's entered! It is a special surprise to see one's best friend's name printed on the screen in a story, for example. It makes the computer seem to know more than it does. As *you* know, the computer is "just following orders."

[1] *Don't forget, if you want to end your sentence with a period, you will have to* tell *BASIC to do it.*

Testing a Program

Before you try your program on a friend, you should test it yourself and make sure it works right. You're almost sure to find that you have left out a space, so that two print strings are jammed together. Or you might discover that the person running your program may get confused over just what is expected. Then too, chances are that you made some error in typing the program, no matter how careful you were. Don't worry. BASIC will tell you just what line an error is in, as described earlier. *You,* on the other hand, are the best judge of whether the program is easy to use and understand.

The following section will explain how you can store programs, so that you need type them only once. As your programs become longer and more complicated, you will appreciate being able to do so. While your program is still in memory, read completely through the instructions for program storage and then follow the steps.

Storing a Program

If you like the program, you may want to save it for another time. You can easily do so with a cassette or a disk drive. If you have a cassette recorder, here is what you do.

Cassette Program Storage

1. Load the cassette you are using into the machine and position it where you wish to begin recording.

2. Set the cassette volume control in the middle (at about 4).

3. Press the Record and Play buttons on the recorder at the same time. Then type

```
CSAVE"P"
```

(The "P" tells the computer that P is the name you want to give your program. Actually, you may give it any one-letter name you wish.)

Later, when you want to load program P from the recorder, just follow these steps.

1. Press the Play button on the recorder.

2. Type

```
CLOAD"P"
```

The computer will search through the tape until it finds program P, and then load it into the computer's memory. Your program should be back and ready to run—or edit.

Disk Drive Program Storage

If yours is a disk drive system, you already have access to disk storage, since you are using Disk BASIC. You may name your program anything you like. In the examples that follow, we will use the name "GAME". To store your program on a flexible diskette, make sure you have a diskette you are willing to use in the proper disk drive. Make sure there is no write-protect sticker on the diskette your program is to be stored on.

1. If you have only one drive or wish to save your program on the system diskette in Drive 0, just type

```
SAVE "GAME"
```

2. If you have more than one drive and wish to save the program on a diskette in Drive 1, type the following:

```
SAVE "GAME:1"
```

A system diskette *must* be in Drive 0 at all times, whether you store your program on it or on a diskette in another drive. Don't forget the quotes; BASIC can't understand your file name if you don't. If you want to load the program later, type

LOAD "GAME"	if your program is on the system diskette in Drive 0, or
LOAD "GAME:1"	if your program is on a diskette in Drive 1.

Something important to know, whichever system you have, is the following: If you are working on a program and then type LOAD or CLOAD and the name of another program you have stored, the computer will load the program you asked for and *wipe out* the one you were just working on! To avoid losing your program then, SAVE any program you are working on *before* you LOAD another one.

You now know how to create, edit and store simple computer programs. The following two chapters will teach you ways you can greatly increase your power to write interesting and enjoyable programs.

GETTING YOUR PROGRAM ORGANIZED

In Chapter 1 you learned about a *loop.* You found that you could get a lot of work out of the computer with a fairly short instruction. Loops are especially useful in programs with line numbers. There is no limit to how complicated your program instructions can become using loops and other programming "tricks." Ah, but there's a catch! The more complicated your program becomes, the harder it will be to keep it organized, easy to understand, and "bug" free. You *could* just experiment with a program until it appears to work right. But experimenting can take a lot of your time. It also isn't as much fun as running a program that works!

For this reason, this chapter will teach you *all* the basic steps needed to write a well-organized program: creating an *algorithm,* making a *flowchart,* and—once the program has been written —*debugging.* In the process, you will also learn more ways to use loops. The best way to become familiar with the basic programming steps is to work through an exercise, performing each one as you go.

More with Graphics

It's quite easy to create "bugs" you *don't* want—that is, program errors. In the exercise that follows, you will create a friendlier "bug," a picture that will print wherever you wish on the screen. This will be a graphics exercise, so you will be using the special graphics characters you first used in Chapter 1.

First, though, how would you go about creating a picture of a "bug" on the screen? How would you know your program was put together as well as possible? How could you save time in locating errors? An *algorithm* will help you answer these questions.

Algorithms

The word "algorithm" is just a fancy term for a "how to do it" set of instructions. It is a very important word in programming, however. What you need to know to write a program can be stated basically as follows.

- What is the purpose of the program?
- What information do you need to write the program?
- What values change often while the steps are carried out?
- What values don't change at all?
- What step should be done first, then next, and so on?
- Which steps are repeated several times, and which ones don't repeat?

For the current exercise, you already know some of the answers to these questions. The program is supposed to print a bug somewhere on the screen. The needed information includes which graphics characters you want to use for the picture, the order in which they should be printed and their character numbers. But is that all you need to know?

First, it would be nice to be able to print your bug at different places on the screen (a moveable "beast"?) without having to change your program each time. That means that you will need to use variables in your PRINT@ statements. You must plan then, a *general* PRINT@ statement (or a set of them). Remember that there are 64 characters per row? Using that fact, you can print on different rows with a statement such as the following.

```
PRINT@64*X, "STUFF TO PRINT"
```

where X is a variable that is from 0 through 15. You can print on any column in that row with another variable Y, where Y may range from 0 through 63.[1]

```
PRINT@64*X+Y, "STUFF TO PRINT"
```

Adding Y to 64*X will cause printing to start at column Y of row X. For example, if X=10 and Y=15, printing will begin at position 640+15, or 655.

Program Steps

Now you have the information you need for your algorithm. Let's write down the steps your program should carry out.

1. Make a sketch of the bug you wish to create. (Use a TRS-80 Video Display Worksheet.)
2. Figure out which graphics characters should be printed in the first row, since the bug will print on a portion of several rows on the screen. (You did want a *big* bug, didn't you?)
3. Figure out which characters go on the other rows.
4. Assign values for X and Y, for the row and column where you want to start printing the bug.
5. Write your PRINT@ statements, in the order of the rows to be printed. The second row should be X+1, and the third row should be X+2. Y's value should remain constant.

Now check your algorithm to make sure of the following:

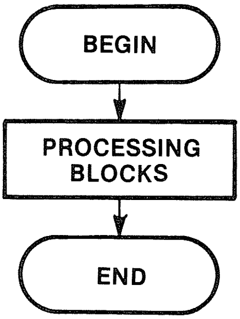
- The instructions didn't leave out any important steps;
- There is a definite beginning and ending point to the program;
- All the *input* information is there, and you know all the *output* that you expect from your program.

[1] *You will often find, as you work with computers, that you begin counting with 0, rather than 1. For this reason, each row of 64 characters is numbered from 0 to 63.*

Making a Flowchart

In the computer world, as elsewhere, "a picture is worth a thousand words." For this reason, flowcharts were invented. A *flowchart* is a diagram of how your program steps are to be carried out. Certain symbols are employed, a few of which will be described below. Creating a flowchart before you program will provide you with a clear idea of the order of steps to be performed. A flowchart will also serve as an aid during the debugging process. As you mentally step through the program's execution, you can compare each line with your flowchart, making sure each line performs as expected.

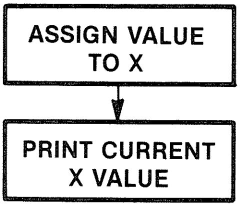
The following is a brief explanation of the flowchart symbols for program steps you are already familiar with. Chapter 5 will present additional flowchart symbols that go with the new programming tools you will learn there.



BEGIN and END

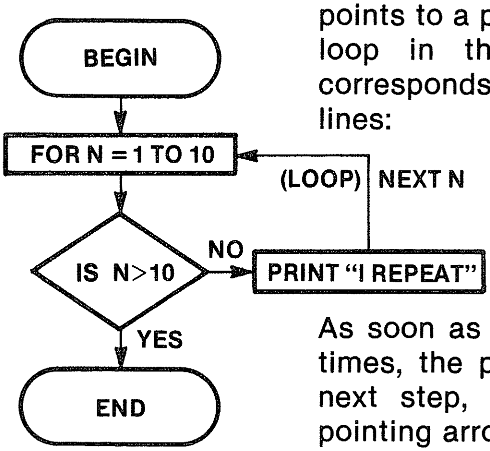
A clear flowchart shows the beginning and end of the program. The words BEGIN and END commonly appear in ovals on the flowchart, as shown here. In between these, the program steps should generally be represented in order, from top to bottom.

Fundamental Operations



A fundamental operation is a simple step for the computer to perform; it is represented on the flowchart by a box. Fundamental operations include assignment and INPUT statements, and PRINT statements. The arrows indicate the order in which steps are executed.

Loops

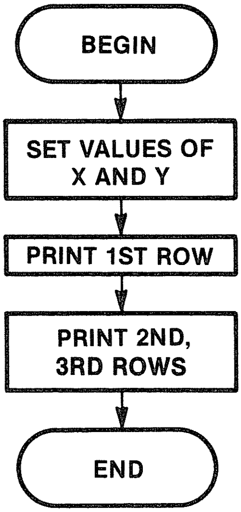


Loops are represented by an arrow that points to a previous program step. The loop in the diagram to the left corresponds to the following BASIC lines:

```
10 FOR N = 1 TO 10
20 PRINT "I REPEAT"
30 NEXT N
```

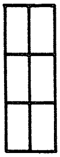
As soon as the loop has executed 10 times, the program will go on to the next step, as shown by the down-pointing arrow.

For practice, you should flowchart the algorithm for your bug program. A sample flowchart is shown below.

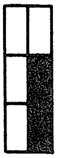


Writing the Program

Now that you have reviewed what steps to follow, let's write the program. How do you go about doing the first step? Take a look at page 125, which lists the graphics characters and their corresponding numbers. Notice that each character is composed of a grid that is divided into six parts.

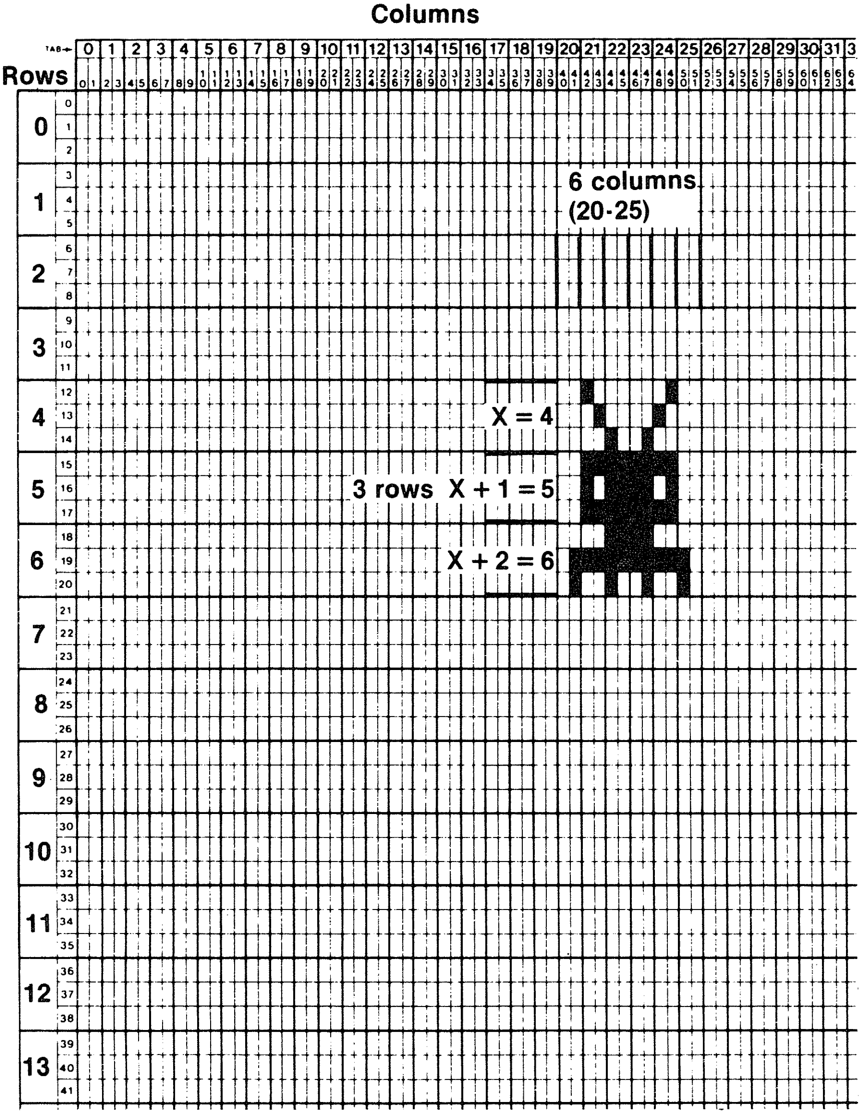


The different characters are the result of different shading combinations in the grid. For instance, character 168, shown here,

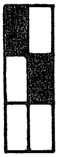


has two of the little rectangles shaded in. Character 128 has none, while character 191 has all six of the rectangles shaded.

An important first step in making a picture that looks the way you intend it to is to sketch it first. Your reference manual contains a worksheet that shows the rows, columns, and grids that match the characters. Make several copies of the worksheet, so that you can experiment with your sketch until it is just the way you want. Your last sketch should be one that follows the exact grid outlines. That way, you will know just which characters you should print, and the order in which to print them. A sample "bug" sketch is shown below. You can make yours the same or entirely different.



Notice that the bottom row starts one character sooner than the first two. In order to be able to print your bug wherever you wish on the screen, it will help to have each row the same width. To do this, fill in the first character in rows one and two with the blank character, number 128. The second character in row 1 requires a shading pattern like this one.



As you look through the "cast of characters," you find that character 137 is the one you want. Just follow this method until you locate each of the six characters in the first row. Your first PRINT@ statement should print the six characters, one after the other. (Recall that you indicate character 137 with

```
CHR$(137)
```

in your PRINT@ statement.)

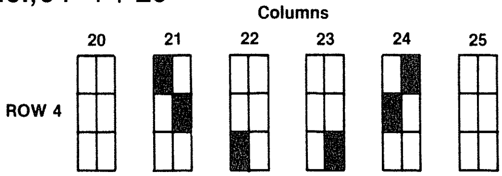
The second and third rows can be put together in the same way. Keep in mind that the first column you use to begin the first row should match the first column of the other rows; that is, Y should remain constant. Otherwise, you will have pieces of your poor bug all over the screen! The X value you set should be the starting row. The second row, then, would be X + 1, and so forth. Here is a copy of the program you should end up with.

```
1ØØ X = 4:Y = 2Ø
11Ø PRINT@64*X + Y,CHR$(128);CHR$(137);CHR$(144);
    CHR$(16Ø);CHR$(134);CHR$(128);
12Ø PRINT@64*(X + 1) + Y,CHR$(128);CHR$(183);CHR$(191);
    CHR$(191);CHR$(187);CHR$(128);
13Ø PRINT@64*(X + 2) + Y,CHR$(168);CHR$(140);CHR$(159);
    CHR$(175);CHR$(140);CHR$(148);
14Ø END
```

Let's run through the program.

Line 100 start your first row of graphics characters at row 4 and column 20.

Line 110 print six graphics characters right in a row starting at location 276.
i.e.,64*4+20



Line 120 print six graphics characters right in a row starting at location 340.
i.e., 64*(4+1)+20

Line 130 Print six graphics characters right in a row starting at location 404.
i.e.,64*(4+2)+20

Line 140 ends the program

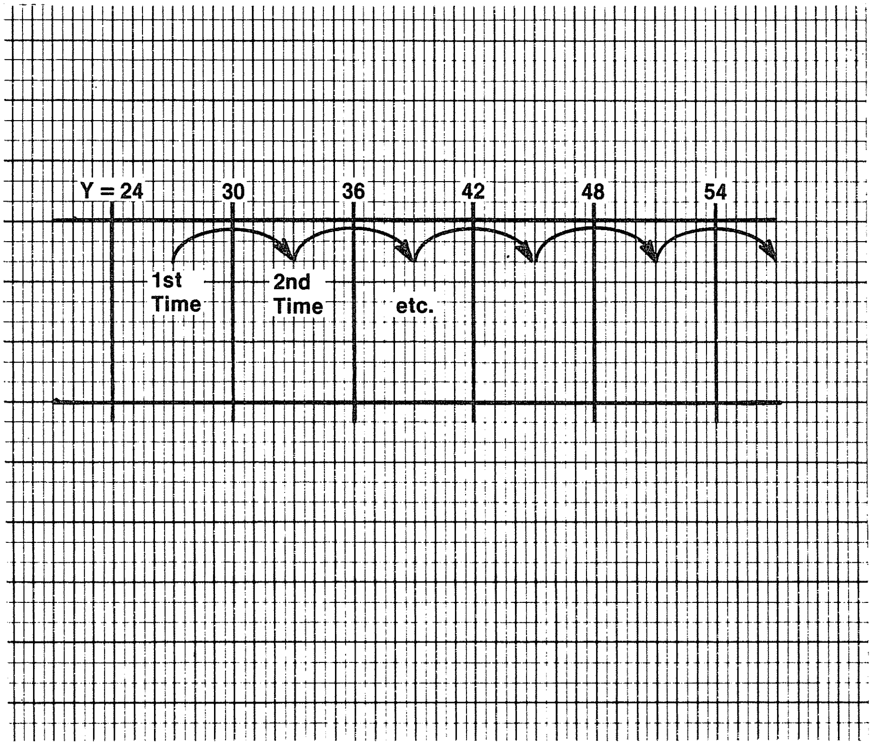
Voilà! You have a new friend! But don't stop there. All you need to do to print your bug in different places is to change the values of X and Y. Try doing this with an INPUT statement. For example, you can change line 100 to

```
1ØØ INPUT "ROW,COLUMN";X,Y
```

Then each time you run your program, you can enter the starting row and column without changing a program line each time. Before you proceed further, you should SAVE your program. If someone were to trip over the power cord, all your efforts would be lost! A good habit to get into is to SAVE programs frequently, in case of accident or power failure. Doing so doesn't affect what is in memory; it merely stores the current version of your program.

Loop Nesting

Suppose your bug is lonely for company. You *could* write the same few lines over and over with different values of X and Y and then run the whole program. You probably wouldn't want to, though, since considerable effort goes into typing a PRINT@ statement for six graphics characters. That's where the power of the *loop* comes in handy. Remember that your bug is six characters wide? If you want an entire row of bugs to print in a loop, you can't simply use Y as the loop index. (The loop *index,* you recall,`is the variable that the computer uses to keep track of the loop count.) This is because Y will normally increase by only one (programmers call this *incrementing*) each time through the loop. You want it to increase by *six* each time, as the following picture shows, so that each new bug doesn't overprint the preceding one.



There is a handy way to increase your loop index by any number you want each time. With the STEP instruction, you can have the computer take any size STEP you wish. Remember the basic loop format?

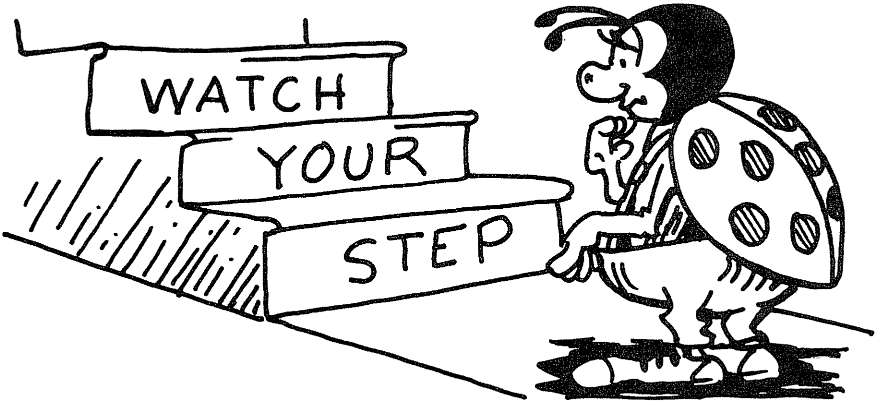
```
1ØØ FOR Y = Ø TO 59
•
•
•
14Ø NEXT Y
```

(That's right, the loop index can begin with zero!) To the basic loop format, add a new part, called STEP as follows.

```
1ØØ FOR Y = Ø TO 59 STEP 6
•
•
•
14Ø NEXT Y
```

The first time through the loop, Y's value will be 0. The next time around, its value will be 6, the next time 12, and so on. As with any other loop, each time the computer performs the loop, it checks Y's value against the ending value you set. In the example above, the computer checks to see if Y is 60 or more. As soon as it is, the loop is finished and the program continues.

A Bug that STEPs



** Try the STEP instruction above with your program. Simply "sandwich" the lines that print your characters between the instructions that control a loop, like this.

```
1ØØ CLS:X = 7:FOR Y = Ø TO 59 STEP 6
11Ø PRINT@64*X+Y,CHR$(128);CHR$(137);CHR$(144);
    CHR$(16Ø);CHR$(134);CHR$(128);
12Ø PRINT@64*(X+1)+Y,CHR$(128);CHR$(183);CHR$(191);
    CHR$(191);CHR$(187);CHR$(128);
13Ø PRINT@64*(X+2)+Y,CHR$(168);CHR$(140);CHR$(159);
    CHR$(175);CHR$(140);CHR$(148);
14Ø NEXT Y
2ØØ END
```

** Now figure out on your own how you would use the STEP instruction to print a *column* of bugs, rather than a *row.* (Hint: X is the row variable, and each bug takes up three rows; there are 16 rows on the whole screen.)

Experiment with the STEP function. You may start a loop index from any value you want, not just 0. Of course, the TRS-80 won't like being asked to print a bug at position 64*X if X has a large value like 36! This is because 64 times 36 is 2304, and there are only 1024 valid places to print on the screen.

You may also start your loop with a large number and end with a small one, thus printing in a reverse direction on the screen. To do this, you just use a *negative* STEP value. Try replacing line 100 in the "Bug that STEPs" program with the following one and see what happens.

```
1ØØ X = 7:FOR Y = 59 TO Ø STEP -6
```

In this instance Y is 59 at the start of the loop; the next time around, it is 53, and so on. And just in case you were wondering. . . Yes, the following two statements do exactly the same thing!

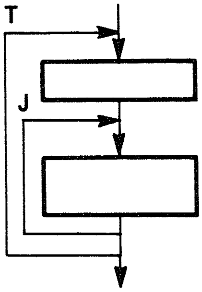
```
1Ø FOR Y = ØTO 59
```

and

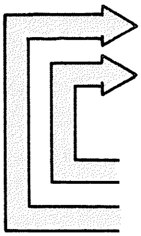
```
1Ø FOR Y = Ø TO 59 STEP 1
```

Nested Loops

Still not enough bugs in your program? How about a whole screen full? Once you understand how to do this, you will have a programming tool that you can use for countless other programs. You have already made a row and a column of bugs print on the screen. Now you will merely combine these by placing one loop (for instance, the one controlling the *column*) inside another! This is called loop *nesting,* since the outer loop makes a "nest" for the inner one. Nested loops can be represented on a flow chart as follows.



** Each time the outer loop repeats once, the inner loop executes completely. Try the following program to see how this works. This program will print several rows of stars (asterisks) on the screen. Variable T determines the row, and J determines the column.



```
5 CLS
1Ø FOR T = 1 TO 1Ø
2Ø FOR J = 6 TO 9
3Ø PRINT@64*T+J,"*";
4Ø NEXT J
5Ø NEXT T
```

When you run this program, you will see how the J values run from 6 to 9 *every* time the T value changes by one. Notice that the NEXT J comes *before* the NEXT T. You wouldn't want the J to fall out of its *nest!* The same applies to three or even more nested loops. The computer will always try to match the last FOR instruction it found with the first NEXT instruction it runs into. If it finds a different loop index than it expected, it will complain with an error message and halt execution.

A Screenful of Bugs

** Let's try the new trick with our bug friends. Try the following replacements in your "Bug that STEPs" program.

```
1ØØ CLS:FOR X = Ø TO 13 STEP 3: FOR Y = ØTO 59 STEP 6
14Ø NEXT Y: NEXT X
```

Debugging

How did your screen look after running the last program? Whether your program worked or not—you will eventually need a few "debugging" tips. Before you do anything else, stop and think through the steps of your algorithm. That will often clear up many problems. Second, give the program lines a careful looking over. Sometimes the hardest bugs to find are the ones that are simple typing errors.

But suppose you have already done all that and the program still does not work? You know that—even if the program doesn't run—your BASIC grammar is okay. Otherwise, the computer would tell you where it hurt. The problem, then, must be a *logic* error. In other words, your thinking process is confused at some point. Usually, it is something fairly simple, such as doing Step 3 when you should do Step 4 first, for example. It might on the other hand, be something a little harder to spot, such as getting mixed up on which variable is used for which item.

Debugging is an entire art in itself when one works with large and complicated programs. Since the programs you write with the aid of this book are very simple ones, the following is a handy list of general rules to follow.

- Make sure you have a list of all the variables you use. Tell how each is used and, if needed, its location within your program.

- Make sure your program follows the algorithm you wrote. It is quite possible that your concept of the program changed as you got into writing it. It saves time to go back and change your algorithm to fit the new method of solving your problem. By stating the idea in simple terms in an algorithm, you might see how your thinking became confused and be able to fix the program quickly.

- Do a "walk-through" of your program. Pretend you are the BASIC interpreter and start working the lines one by one. When you come to a loop, keep track of the loop index and loop back. If any calculation is to be made inside the loop, figure out what it should be and jot down the new value. In other words, pretend you are a very slow but wily computer, and see whether values are being changed when they should not be, or whether a line of code is inside a loop that should be outside of it, etc.

- It often helps to check the value of some variable before another part of the program changes it. You can place a temporary instruction in your program that will halt execution; once it stops, you may ask the computer to print the current value of as many variables as you like. You halt execution with the STOP instruction. When BASIC comes to the STOP command, that is just what it does.

Temporary instructions, such as the STOP command described above, provide an important debugging tool. The following example illustrates how you might use a STOP statement for debugging. Suppose you were writing a program to print multiples of 7. You used K for a loop index in your program and set I equal to 7, as shown here.

```
2Ø REM ***EXAMPLE OF A PROGRAM
25 REM *** THAT WON'T WORK!
5Ø I = 7
6Ø FOR K = 1 TO 25
•                        (The dots just stand for lines that
•                        don't relate to the current problem.)
9Ø K = K*7
1ØØ PRINT I
125 NEXT K
```

You find that—even though your loop should repeat 25 times—the loop ends after only two times through. Not only that, the computer prints 7 each time, instead of 7, 14, 21, etc. You may add a temporary line 120, as follows, in order to print the current values of K and I.

```
12Ø STOP
```

As soon as the program reaches line 120 during execution, it will dutifully STOP and state,

```
READY
```

Then you can take a look at the values of K and I. Do this just as you did at the beginning of the book—without line numbers.

PRINT K,I $\boxed{\text{ENTER}}$

If your program worked correctly, the value of K would be 1, since the program was instructed to STOP the first time through the loop. Instead, you discover that K's value is 7. You then look more carefully at line 90

```
9Ø K = K*7
```

There's your bug! You set I = 7 before coming into the loop. Then in line 90, you type K = K* 7 instead of I = K* 7, as you intended. Since you were changing the value of K, instead of letting the loop *increment* it, the loop ended early. Once you have found the error and made the correction, be sure to edit out the STOP, so your program will run normally.

You may find it helpful to add a temporary PRINT statement to your program, so that you may view values *as* the program executes. For instance, in the above example, you could have added a line

```
12Ø PRINT K
```

so that K's value as well as that of I would be printed each time through the loop. That way, you could watch the values change and not interrupt the program. If the program executes too quickly for you to see the changing variable values, here is yet another handy temporary instruction: Remember that the computer will wait for you to make an entry when it encounters an INPUT statement. Thus, along with the temporary PRINT statement in the above example, you may add an INPUT instruction, as follows.

```
12Ø PRINT K:INPUT A$
```

Each time through the loop, the program will print the value of K and then wait for input from you. As soon as you see what K's current value is, you may then press any key (besides the special keys). Then press ENTER and the program will continue.

One further use of temporary program lines deserves mentioning. Suppose your program is becoming long and complicated. You aren't sure whether BASIC even gets to a few lines of it. Just to see whether it reaches the section in question, try inserting a temporary PRINT statement in that section. Have the output appear somewhere on the screen where you will notice it, such as the lower righthand corner.

```
335 PRINT@1Ø1Ø, "OKAY HERE!"
```

If you *don't* see the message "OKAY HERE!" appear, you can be certain the program never got to the section in question.

The programming process has only been briefly outlined in this chapter. If you make a habit of following the important steps in that process—writing an algorithm, making a flowchart, and carefully debugging the completed program—you will find programming much easier and enjoyable. Of course, the greatest teacher is experience!

PROGRAMMING CONTINUED

In Chapter 4 you learned several ways to make your TRS-80 loop. There are a number of other means of changing the order in which program lines execute. In this chapter you will learn about three of these ways: the GOTO statement, the *selection* structure, and the *subroutine.*

The GOTO Statement

The GOTO statement is really quite simple. Whenever the computer comes across an instruction such as

```
15 GOTO 65
```

it will ignore lines between 15 and 65 and *jump* to line 65, carrying out any instructions it finds there. It will then continue executing any lines that follow 65. You can even use the GOTO statement to loop *back* to a previous line. That fact deserves a note of caution! The GOTO statement has a poor reputation among many programmers, because it is so easy to use GOTO to write carelessly constructed programs. Look at the following program.

```
1Ø PRINT "IS THERE ANY WAY OUT OF THIS?"
2Ø GOTO 1Ø
```

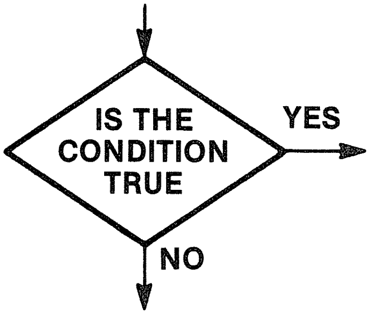
Before you read on, try to guess what will happen when you run this program. That's right! The TRS-80 will tirelessly print "IS THERE ANY WAY OUT OF THIS?" over and over, because line 20 repeatedly sends execution back to line 10. The program above is an example of an *infinite loop,* because the program will keep running forever unless it is interrupted. Programmers must always be careful to avoid an infinite loop, since the computer will never be able to complete a program that contains one. An unintended infinite loop is only one example of ways you can "lose control" of your program.

Fortunately, there *is* a way to halt infinite loops. In fact, some programs are written intentionally as infinite loops! Some action will be repeated continually until the user halts execution. To stop an infinite loop, just press the BREAK key, and the computer will stop the program and return to the BASIC command mode; that is, it will be ready for you to change lines in your program.

GOTO *can,* however, be quite useful. Just be sure to keep track of *where* you are sending program execution, and *how* the program will end. The exercises in this chapter will illustrate some of the uses of the GOTO statement.

Selection

One of the most powerful programming tools you can use is having the computer make *decisions.* Suppose you only want to run part of your program if certain conditions are true. If those same conditions are false, then you might want to simply skip that part or run another part of the program. That is why this program structure is called *selection,* since the computer chooses only certain parts of the program for execution. The diagram below shows how decisions are represented on flowcharts.



Unlike the flowchart symbols introduced in Chapter 4, the symbol for a decision has one entrance and *two* exits. The program will take one path if the stated condition is true, and the other path if the condition is false. You can easily program the TRS-80 to make decisions. But does that mean your computer is pretty smart after all? Not at all. As usual, the computer merely does exactly what you tell it to do.

But do you know just what you're telling the computer to do? Remember, every order you give the computer is changed into 1's and 0's. That means that it can easily make yes-or-no decisions—a "yes" answer is 1 and a "no" answer is 0. (The computer can't handle an answer of "maybe"!) Let's take a closer look at what is involved in each computer decision. The computer can compare one variable value with another; it can also compare strings. If two strings are just alike, then the computer considers them *equal.* They are either equal or *not* equal, period. With numbers, though, there are always *three* possibilities: one number is either *equal to, less than,* or *greater than* another number. That is, "not equal" means one number is either larger or smaller than the other. It is important to keep this in mind when you write a program that makes decisions about numbers.

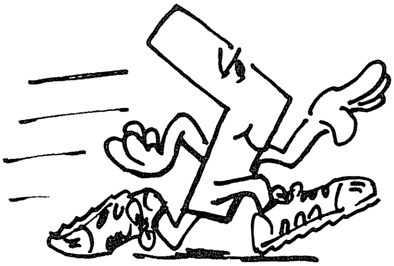
The BASIC instructions for making a decision are as follows.

```
IF (condition) THEN (orders to follow if condition is true)
```

If *you* have a decision to make, you might say to yourself, "If our team won more than eight games, then we are great!" If you wanted to say the same thing in BASIC, you might type

```
IF WINS > 8 THEN TEAM$ = "GREAT!"
```

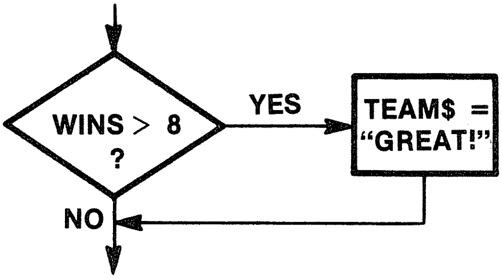
This would tell the computer, "Check the variable named WINS. If its value is greater than 8, then our team (represented by string variable TEAM$) is GREAT! If your team won fewer than eight games *or* if it won *exactly* eight games, then the rest of the statement is ignored. These last two conditions are called the *complement* of the condition that the computer is checking. In other words, the complement of "greater than" is "equal to or less than."



The following is a table that shows each symbol that BASIC uses to make decisions. Next to each symbol is its complement.

Condition	Symbol	Complement	Symbol
equal	=	not equal	< >
not equal	< >	equal	=
greater than	>	less than or equal to	<=
less than	<	greater than or equal to	>=
greater than or equal to	>=	less than	<
less than or equal to	<=	greater than	>

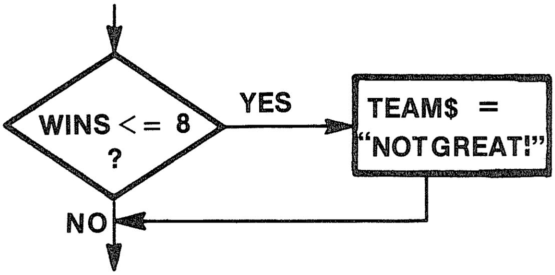
The example used earlier could be expressed either of two ways. One way would be



and the matching BASIC statement would be, as you already saw,

```
IF WINS > 8 THEN TEAM$ = "GREAT!"
```

You could say basically the same thing by using the complement of "WINS > 8".



The corresponding BASIC expression would be

```
IF WINS < =8 THEN TEAM$ = "NOT  GREAT!"
```

You can create several instructions for the computer to carry cut if the condition is true. Just use a colon to separate them. For instance, the following is a good example of an IF. . .THEN statement.

```
1Ø IF X = Y THEN Z = Z + 1: PRINT "KEEP IT UP!": GOTO 5Ø
```

When the computer reaches line 10 and finds that X *does* equal Y, it will increment Z, print a message, and GO TO line 50. One of the ways GOTO can best be used is in IF. . .THEN statements. The above could be written simply

```
1Ø IF X = Y THEN GOTO 5Ø
```

At line 50 you would have all the instructions you want performed *if* the condition is true.

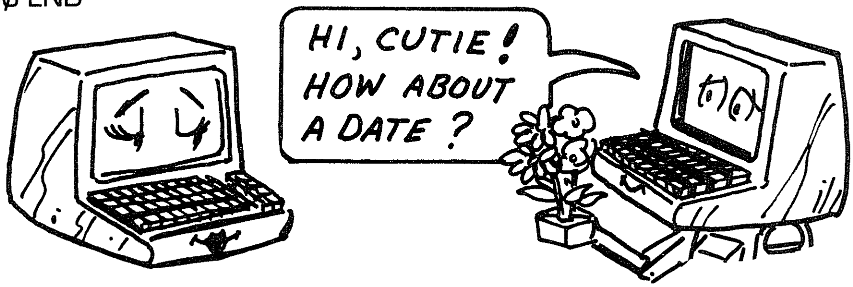
```
5Ø Z = Z + 1
6Ø PRINT "KEEP IT UP!"
7Ø (other instructions)
```

Make sure, however, that you don't allow the computer to execute lines 50 and 60 if the condition is *not* true. If, for example, X is *not* equal to Y, then the computer will execute the line following line 10. If you don't want it to simply step around to line 50, you must put an instruction before line 50 to make it go around. A sample way to do this is as follows.

```
1Ø IF X = Y THEN GOTO 5Ø
2Ø PRINT " THIS IS A FILLER LINE"
3Ø GOTO 7Ø
5Ø Z = Z + 1
6Ø PRINT "KEEP IT UP!"
7Ø (other instructions)
```

** As a very simple example of how a program might use a decision, consider how the TRS-80 might make a date. It could ask each user whether he or she were a machine, and print a reply based on the user's response. The following lines show how this may be done.

```
1Ø PRINT "HI! ARE YOU A MACHINE? (TYPE 'YES' OR 'NO')"
2Ø INPUT A$
3Ø IF A$ = "YES" THEN PRINT "YOU'RE JUST MY TYPE! HOW
   ABOUT A DATE?":GOTO 5Ø
4Ø PRINT "YOU'RE CUTE, BUT I HAVE A HEART OF STEEL."
5Ø END
```



Note that the computer will check for the *exact* string—in this instance, "YES"—to determine whether it is equal to the value entered for A$. The strings "Yes" (with lower case letters) or " YES" (with a blank in front) will be rejected as *not* equal.

The Numbers Game

** Here is a game that will impress your friends. The computer will "think" of a random number between 1 and 20 inclusive, which your friend will try to guess. The computer will let her know when she is getting close to the number and tell her if she guesses it.

Functions

The random number is selected by using a *function* called RND. You can think of a function as a small, built-in program that you can use in the programs you write. There are many functions that come with most versions of BASIC—some for helping you with strings and some for numbers. The RND function is for numbers. You can have the computer pick a random number between 1 and 20 inclusive with the following statement.

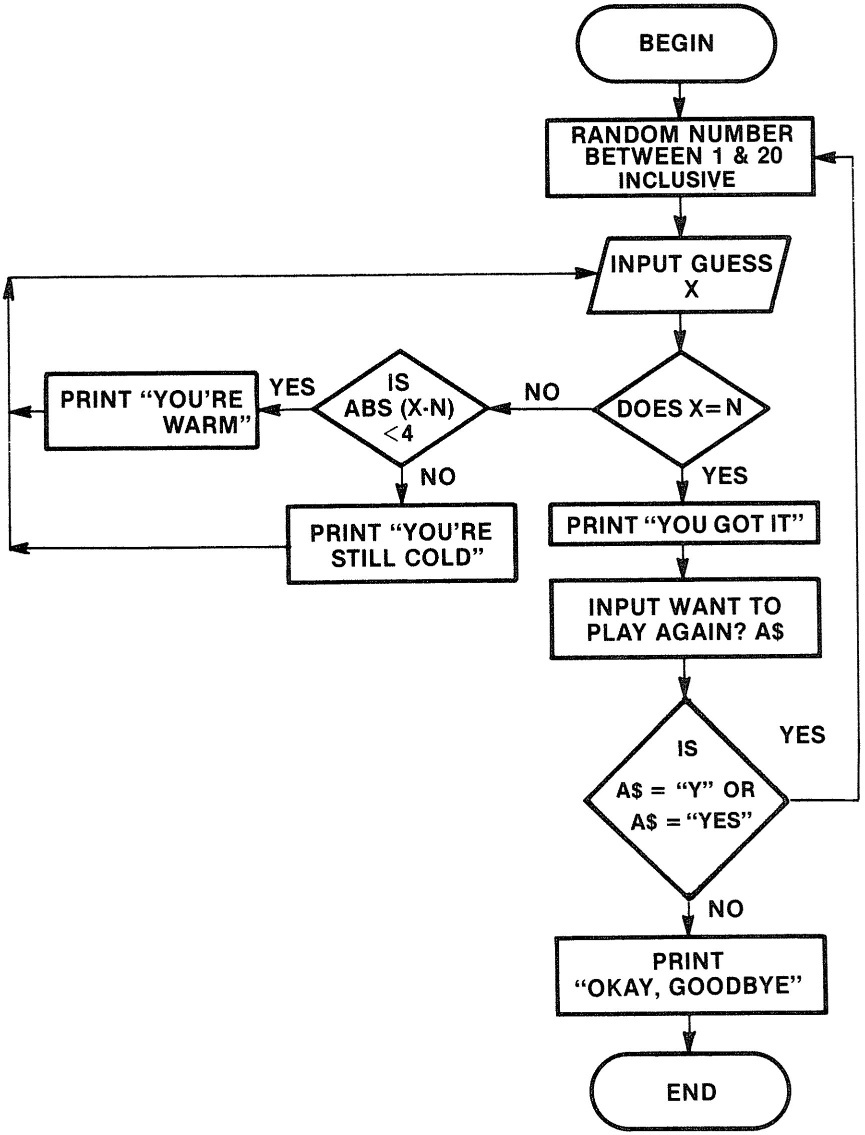
```
X = RND(2Ø)
```

(The number inside the parentheses is called the *argument* of the function. An argument may be a number, an expression, or a variable.)

Another function this game will use is ABS, for ABSolute value. The value of the expression ABS(6-9) is 3, just as the value for ABS(9-6) is 3. It is of no importance which number in the argument is larger, it only matters what their difference is. It will always be positive.

You'll find that the "guessing" program is really very simple. It uses several IF. . .THEN statements to have the computer choose which "helper" statement to print for your friend.

For practice at flowcharting, you should step through the sample flowchart of the program shown below. Following that is a listing of the program. You can make your own program as simple or as complicated as you like. Try customizing the message you print for the friends who play your game!



Here is a listing of the program.

```
1Ø X = RND(2Ø)
15 PRINT "ENTER A NUMBER BETWEEN 1 AND 2Ø."
2Ø INPUT "YOUR GUESS"; N
3Ø IF X = N THEN PRINT "YOU GOT IT!": GOTO 1ØØ
4Ø IF ABS(X-N)<4 THEN PRINT "YOU'RE WARM!":GOTO 2Ø
5Ø PRINT "YOU'RE STILL COLD!"
6Ø GOTO 2Ø
1ØØ PRINT "WANT TO PLAY AGAIN?"
11Ø INPUT A$
12Ø IF A$ = "Y" OR A$ = "YES" THEN GOTO 1Ø
13Ø PRINT "OKAY, GOODBYE!"
14Ø END
```

Logical Operators

Take a good look at line 120 in the program above. Notice that its form is

IF (condition 1) OR (condition 2) THEN . . .

The "OR" is one example of a *logical* operator. You are already familiar with the arithmetic operators (such as "+" and "-") and the symbols for comparing strings and arithmetic expressions (see Page 91). There are additional BASIC words that allow you to combine two or more conditions within a selection structure, as in line 120 of the preceding exercise. When the computer finds the statement

IF (condition 1) OR (condition 2) THEN . . .

it will execute the THEN part of the statement if *either* condition 1 or condition 2 *or both* is true.

A second logical operator is AND. Whenever the computer finds a statement such as the following

IF (condition 1) AND (condition 2) THEN. . .

it will execute the THEN part of the statement *only* if *both* conditions are true. Yet another logical operator is the word NOT, which simply negates an expression. For example,

NOT(A = B)

is the same as the expression

A< >B

Experiment with these operators to see for yourself just how they work.

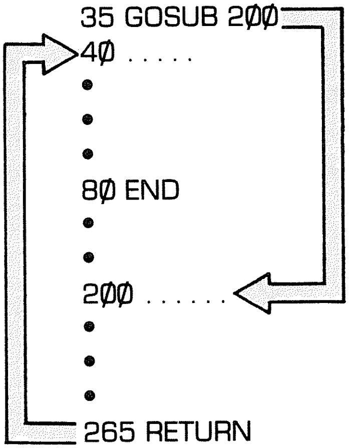
** The following exercise illustrates how the logical operator AND can be used. Suppose a princess wants to marry only a handsome prince. She might make a computerized application form for prospective husbands. If an applicant responds with the right combination of characteristics, he is the one for her. The princess might write the following program.

```
1Ø PRINT "PLEASE ANSWER THE FOLLOWING QUESTIONS
   TRUTHFULLY "
2Ø PRINT "TYPE YES OR NO"
3Ø INPUT "ARE YOU OVER 9Ø YEARS OF AGE";A$
4Ø INPUT "ARE YOU A REAL PRINCE";P$
5Ø INPUT "ARE YOU HANDSOME";H$
6Ø PRINT
7Ø IF A$ = "YES" THEN PRINT "SORRY—I'D RATHER HAVE A
   YOUNGER PRINCE.":GOTO 9Ø
8Ø IF P$ = "YES" AND H$ = "YES" THEN PRINT "WHAT ARE YOU
   DOING SATURDAY?":GOTO 1ØØ
9Ø PRINT "BETTER LUCK NEXT TIME!"
1ØØ END
```

As you see, line 80 makes sure that the applicant is both a prince *AND* handsome.

Subroutines

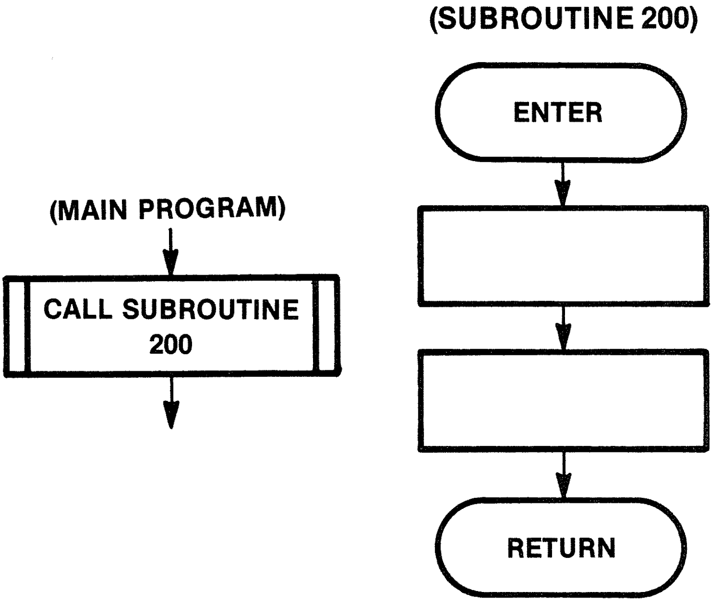
Remember your "bug" friend from Chapter 4? How would you like to make it hop around on the screen? The following describes how you can do that, as well as any "animation" you like, by using *subroutines.* A subroutine is a part of your program that is executed several times. How is it different from a loop then? A loop is also a part of the program that is executed several times, but sequentially. Then the execution moves on. The subroutine, on the other hand, is a part of the program that is executed once here, then once later on, and so forth. Programmers say that a program "calls" a subroutine. The way to call a subroutine is as follows.



```
35 GOSUB 2ØØ
4Ø .....
•                  (other program lines)
•
•
8Ø END             (an END statement, which prevents
•                    accidental execution of
•                    subroutine lines)
2ØØ ......         (first line of subroutine)
•
•
•
265 RETURN         (last line of subroutine)
```

Line 35 in the example above *calls* the subroutine, whose first line is 200. The last line of the subroutine is 265, a RETURN statement. The computer must find exact pairs of "GOSUB" and "RETURN" statements in order to successfully execute the lines of the subroutine and RETURN to the main part of the program. In the example above, the RETURN statement will cause program execution to return to line 40, the line following the GOSUB statement. The END statement in line 80 prevents the computer from stumbling into the subroutine illegally. It is a recommended practice to make all of your subroutines with higher line numbers than the body of your program; you should also use an END statement at the end of the main program.

The flowchart for a subroutine looks much the same as that for the main program. However, instead of BEGIN and END, the symbols ENTER and RETURN are used, as the following example shows.



There is another use for subroutines besides executing the same few lines at different places in the program. You will find that your programs are much easier to debug and simpler to understand if you break them up into parts and make each part a subroutine. For example, if you have three major steps in your program, you might write a main program that consists of the following.

```
•
•
2Ø GOSUB 1ØØ
•
•
3Ø GOSUB 2ØØ
•
•
4Ø GOSUB 3ØØ
5Ø END
```

Then you would write subroutines 100, 200, and 300. When you finished writing the program, you could test each subroutine separately to make sure it was free of errors. You'll be sure to find that this style of writing programs is much easier—and thus much more fun!—than writing long, rambling sequences. *Again, a key to success is to write a clear algorithm and then a flowchart before you start to program.*

** If you saved your first "bug" program from Chapter 4, you may now simply turn the lines that print your bug into a subroutine. Just for practice in calling a subroutine and returning to the main program, load your bug program and then add the following lines.

```
1Ø GOSUB 1ØØ
2Ø END
```

(Be sure to change the END statement on your original program to a RETURN statement.)

Next, write a *second* subroutine that prints the bug in its "hopping" position, and a *third* one that "erases" the bug. You probably understand already just how the bug can appear to "hop" around on the screen. The basic sequence would be

1. Print "regular" bug.
2. Erase it.
3. Print "hopping" bug.
4. Erase *that.*
5. Print "regular" bug in a new place on the screen.

If you repeatedly run a program that contains this sequence of steps, you will see a hopping bug! A sample program for the sequence is printed below. Use your imagination! The program printed here can serve merely as a guide. Think up different things to do with your bug. Design different creatures!

```
   1 REM BUG HOPPING PROGRAM
[1]1Ø INPUT X,Y
   2Ø GOSUB 1ØØ
   3Ø GOSUB 3ØØ
   4Ø GOSUB 2ØØ
   5Ø GOSUB 3ØØ
   6Ø GOSUB 1ØØ
   7Ø END


   1ØØ REM REGULAR BUG
   11Ø PRINT@64*X+Y,CHR$(128);CHR$(137);CHR$(144);
       CHR$(160);CHR$(134);CHR$(128);
   12Ø PRINT@64*(X+1)+Y,CHR$(128);CHR$(183);CHR$(191);
       CHR$(191);CHR$(187);CHR$(128);
   13Ø PRINT@64*(X+2)+Y,CHR$(168);CHR$(140);CHR$(159);
       CHR$(175);CHR$(140);CHR$(148);
   14Ø RETURN
```

[1] *limit X, from 0 through 12 and Y, from 0 through 57*

```
2ØØ REM HOPPING BUG
21Ø PRINT@64*X+Y,CHR$(152);CHR$(140);CHR$(144);
    CHR$(160);CHR$(140);CHR$(164);
22Ø PRINT@64*(X+1)+Y,CHR$(128);CHR$(183);CHR$(159);
    CHR$(175);CHR$(187);CHR$(128);
23Ø PRINT@64*(X+2)+Y,CHR$(140);CHR$(172);CHR$(140);
    CHR$(140);CHR$(172);CHR$(140);
24Ø RETURN
```

```
3ØØ REM ERASE BUG
31Ø PRINT@64*X+Y,"      ";[1]
32Ø PRINT@64*(X+1)+Y,"      ";[1]
33Ø PRINT@64*(X+2)+Y,"      ";[1]
34Ø RETURN
```

The REM Statement

How is your bug behaving? Notice the first line of the program and the first line of each of the subroutines. They all begin with "REM". REM is a handy BASIC word that means the remainder of the line is a REMark. BASIC will ignore anything that follows the REM instruction (that is, until the beginning of the next line). It is a very good practice to put remarks in your program to inform you—and others who might read your program listing—what each part does. Often, you will want to change an old program. Instead of having to figure out what a section does, you will find it helpful to put a REMark that quickly explains that section. A good programming practice is to place a remark at the beginning of each program indicating the program's name and its purpose.

[1] *(Press space bar 6 times between quotation marks)*

"Do-Nothing" Loops

Sometimes computers work faster than you would really like them to. In the preceding program, you probably had to watch closely in order to see your bug hop. You can easily slow down the execution of different steps of a program—an important ability, when doing computer graphics! Since the computer isn't satisfied without *something* to do, why not give it some "busy work"? A good way to do so is by inserting loops that do nothing except loop. For example, the following line inserted in a program will cause a delay of about three seconds.

```
4ØØ FOR X =  1 TO 1ØØØ:NEXT X
```

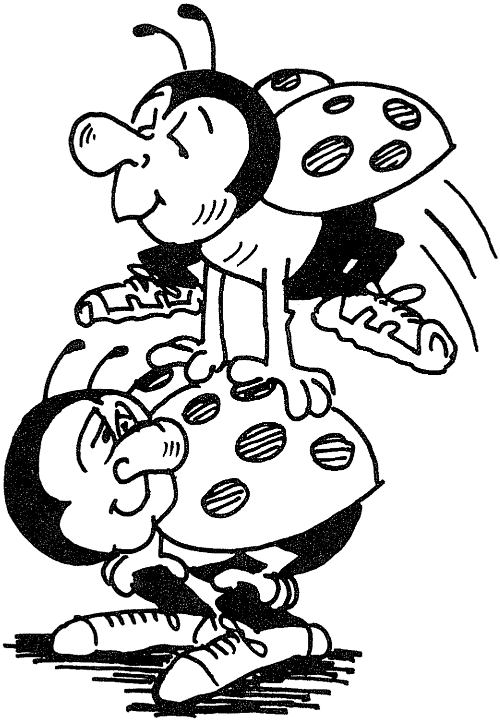
(Of course, the line number you assign depends on the point at which you want the pause to be executed.) Experimentation will provide you with the numbers that are best for the amount of delay you want in each instance. For example, in our "BUG HOPPING PROGRAM," add the following lines:

```
25 GOSUB 4ØØ
45 GOSUB 4ØØ
4ØØ FOR K = 1 TO  25:NEXT K
41Ø RETURN
```

Animation

You can go even further with your program and *animate* your bug by placing the main part of the program (lines 1 through 60 in the sample above) inside a loop. Using the STEP instruction you learned about in Chapter 4, you can make the bug hop across or down the screen—or just about anywhere you choose. A modification of the main part of the bug hopping program, listed below, shows how you can create the animation. As you have probably found on your own, there is no limit to the number and kinds of interesting things you can do with the programs you write!

```
5 REM NEW IMPROVED BUG HOPPING PROGRAM
1Ø FOR X = Ø TO 12 STEP 3
15 FOR Y = Ø TO 57 STEP 6
2Ø GOSUB 1ØØ
25 GOSUB 4ØØ
3Ø GOSUB 3ØØ
4Ø GOSUB 2ØØ
45 GOSUB 4ØØ
5Ø GOSUB 3ØØ
6Ø NEXT Y
65 NEXT X
7Ø END
```



```
1ØØ REM REGULAR BUG
•
•
•
2ØØ REM HOPPING BUG
•
•
•
3ØØ REM ERASE BUG
•
•
•
4ØØ FOR K = 1 TO 25: NEXT K
41Ø RETURN
```

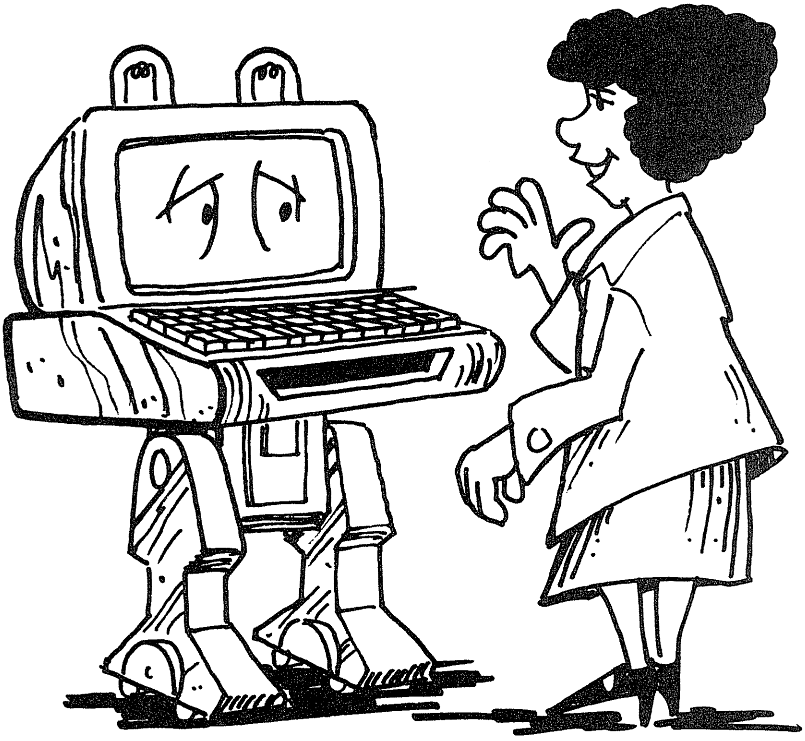
Conclusion

You should know enough by now—providing you have worked through the exercises—to design and write a variety of computer programs. This book has presented many of the important steps in creating programs, as well as the major BASIC instructions needed. Your own experience, however, is the best teacher in the world! You will get great satisfaction out of seeing graphics on the screen or having your friends think the computer knows them by first name, realizing that such effects are the result of your own effort and imagination. Enjoy your TRS-80!

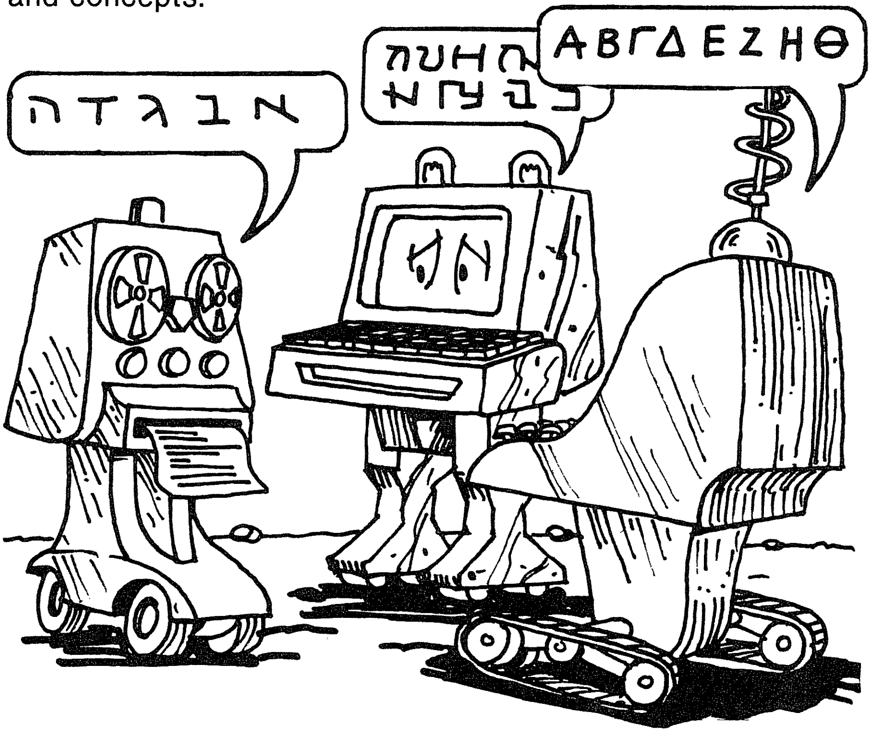
COMPUTER LANGUAGES

From your introduction to microcomputers, you know that computers are not able to function unless an interested person provides the directions. Computers need instructions in the form of programs for every task. Some of these programs are built into the computer when you buy it and are stored in the computer's memory (in the ROM) ready to use.

These instructions tell the computer how to respond to your commands, perform all arithmetic operations and perform other fundamental tasks that you might take for granted.These built-in programs also include instructions which allow you to communicate with the computer in an English-like language.



There are many different languages with which people can communicate with their computers, just as there are many different languages used by people from different parts of the world. These languages developed because people wanted to convey different ideas that involved a unique variety of terms and concepts.



In The Beginning

The very first computers were programmed simply by the way the various parts were connected. A new idea was communicated to the computer simply by changing some of the connections. Information was held in vacuum tubes and later in transistors by a simple code based on whether the vacuum tube was turned on or off. Thus everything in the computer had to be written in a code that had only two symbols such as + and - or Ø and 1. Can you imagine trying to write a message to a friend in a secret code with only two symbols?

Assembly Language

The computer itself still understands only the two symbols, Ø and 1, called BInary digiTS or BITS. Since people don't think in a language like that and writing a coded page filled with zeros and ones without making a mistake is difficult, easier ways to "talk" to the computer were needed. Assembly language code is easier for people because it uses more symbols,letters, and numbers. The computer translates this code into zeros and ones and follows the instructions. This code works very well for people who use the language to program computers regularly. For beginning computer users this code appears strange, secret, and difficult to use.

Typical code in assembly language looks like this:

```
Ø3B1   F6   EF   F4   LDB
Ø3B4   C5   Ø2        BITB
Ø3B6   27   F9        BEQ
Ø3B8   84   7F        ANDA
Ø3BA   B7   EF   F5   STA
Ø3BD   39             RTS
```

This particular routine checks whether a character has been typed on the keyboard, waits until one is typed,and then puts that letter or number on the screen.

Computer languages that use familiar words and symbols are known as high level languages. They were developed beginning in the 1960's. In these high level languages, every word represents an entire series of instructions for the computer. Each word entered into the computer refers to a sequence of instructions already stored in the ROM as "low level" computer code similar to the assembly language routine already described.

FORTRAN

The very first high level language, FORTRAN, remains one of the more popular computer languages and is now available on some microcomputers. FORTRAN means FORmula TRANSlation because everything in this language is based on the use of mathematical formulas. FORTRAN is primarily used by scientists and mathematicians. A strength of the FORTRAN language is that the computer can run a FORTRAN program very quickly.

Here is a sample of a FORTRAN program designed to analyze a True-False questionnaire of 50 questions.

```
      LOGICAL ANS (5Ø)
      READ (5,1ØØ) ANS
      KOUNT = Ø
      DO 1Ø I=1, 5Ø
      IF (ANS (I) ) KOUNT = KOUNT + 1
   1Ø CONTINUE
      WRITE (6,2ØØ) KOUNT
      STOP
  1ØØ FORMAT (5Ø L1)
  2ØØ FORMAT (1X, I3)
      END
```

BASIC

As you already know, the language you used is BASIC. BASIC stands for Beginners All-purpose Symbolic Instruction Code. Most microcomputers use the BASIC language, but it was not originally intended to be a programming language. In 1963 a group at Dartmouth College wrote BASIC as a program to help non-mathematics students learn about the computer. BASIC was so successful that it grew into a language of its own. BASIC is easy to learn and convenient for programmers to write, but is not as fast for the computer to use as some other languages.

Here is a sample of a BASIC program.

```
NEW
1Ø  PRINT "HI,WHAT IS YOUR NAME?"
2Ø  INPUT N$
3Ø  PRINT "DO YOU HAVE A FRIEND WITH YOU?"
4Ø  INPUT A$
5Ø  IF A$ = "YES" THEN 1Ø
6Ø  PRINT "TOO  BAD, GOOD-BYE ";N$
7Ø  END
```

Try this on the TRS-80 now. It will work for you because the TRS-80 speaks BASIC. Press ENTER at the end of each line. Type RUN and press ENTER to see it work.

There are other languages available for microcomputers and someday you may want to learn another one. Each computer language has its own characteristics and specific strengths.

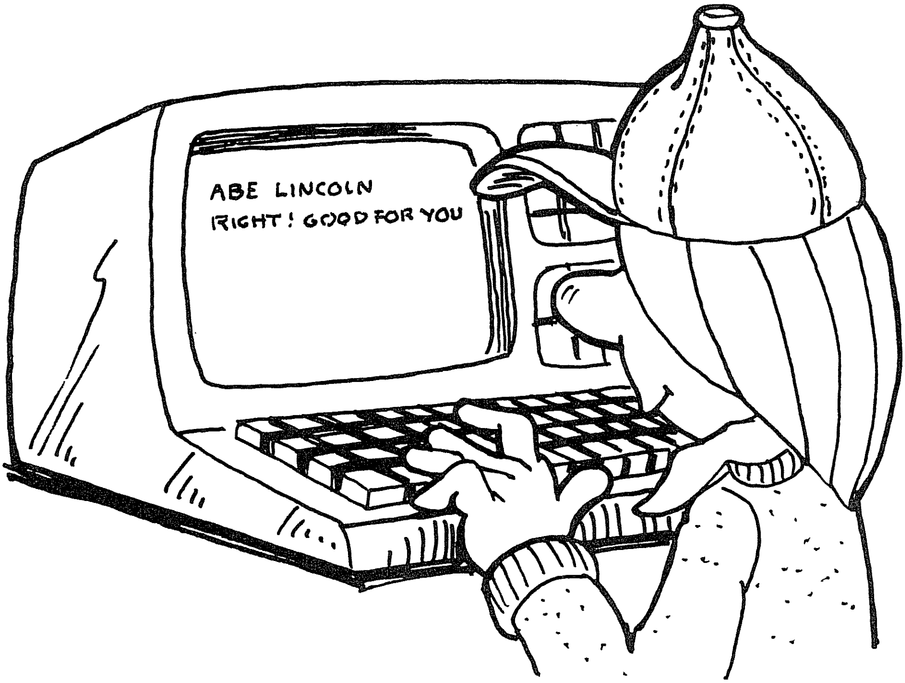
PILOT

PILOT was originally designed for teachers to write instructional material. The name itself stands for Programmed Inquiry Learning Or Teaching. With the PILOT language a teacher can easily ask questions, receive and check answers, and give varied responses. The language also includes ways of presenting material on the screen in an interesting way. There are several versions of PILOT available for microcomputers. Since the commands in PILOT are one or two letters only, it is an ideal language for beginners. Some simplified versions of PILOT are designed to be an introductory language.

A program written in PILOT looks like this:

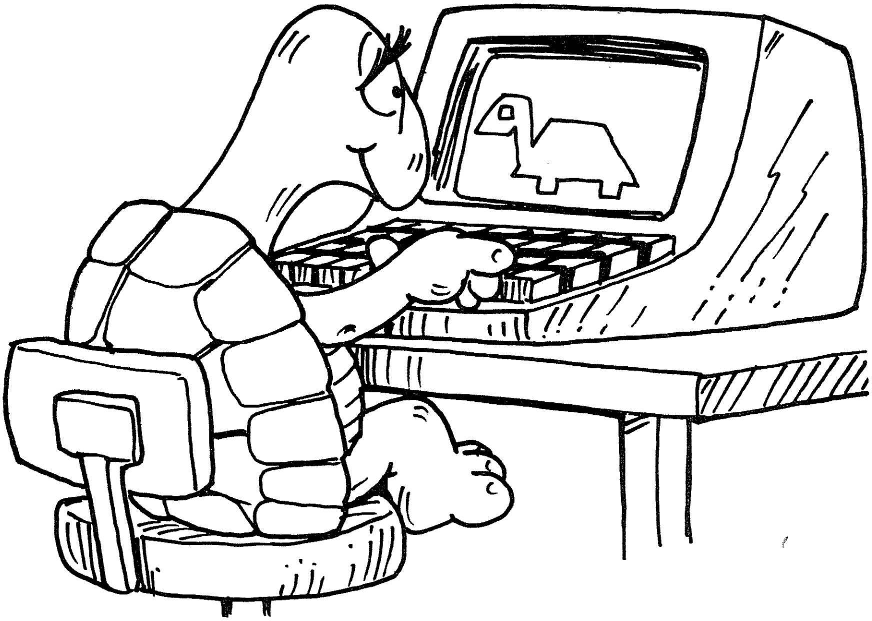
```
1∅ R: CONDITIONAL MATCH PROGRAM
2∅ T: WHO WAS THE 16TH PRESIDENT OF THE UNITED STATES?
3∅ A: $P
4∅ M: ABRAHAM LINCOLN, LINCOLN, ABE LINCOLN
5∅ TY: RIGHT! GOOD FOR YOU.
6∅ TN: NO, THE ANSWER IS ABRAHAM LINCOLN.
7∅ T: COME BACK LATER FOR ANOTHER PRESIDENT QUESTION.
8∅ E:
```

This program prints the question on the screen and waits for an answer. When the answer is given, the program checks to see if it matches the correct answer. The response to the student depends on whether the answer is a match or not a match.



LOGO

LOGO is another language available on many microcomputers. A special part of LOGO called Turtle Graphics has been included in other languages and is available on even more computers. LOGO is quite different from PILOT and BASIC. You can write procedures in LOGO with just a few commands and then build the language to fit your ideas by defining your own terms. You begin to "teach" the computer the language you want it to know.



A procedure written in LOGO looks like this:

```
TO SQUARE  :SIDE
REPEAT 4 (FORWARD  :SIDE RIGHT 9∅)
END
```

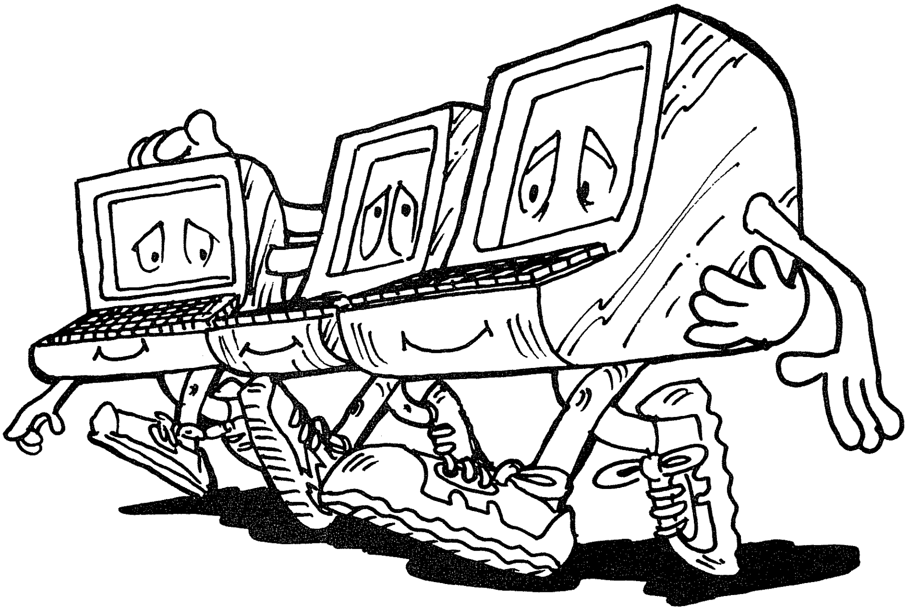
This program will draw a square of any size you choose on the screen. For example, the command SQUARE 1∅ will make a square with 10 units on a side.

LISP

LOGO was derived from the language called LISP, which is very different in appearance from most of the other languages. For example, (Times 9 3) would put 27 on the screen. After entering

```
(SET'FRIENDS' (DICK JANE SALLY) )
```

into the computer, (DICK JANE SALLY) would appear on the screen when you typed FRIENDS!



PASCAL

Another language that is commonly used with microcomputers is PASCAL. This is a relatively new language compared to BASIC and FORTRAN. You may want to learn PASCAL and become a programmer. PASCAL was named for Blaise Pascal, the French mathematician who designed the first mechanical adding machine while he was still a teenager.

PASCAL is called a structured language and looks quite different from the other languages discussed previously. Once you're familiar with PASCAL, however, it is very easy to read. PASCAL was designed to write very large programs and for teams of programmers working together on a large project. One of the advantages to PASCAL is that it runs more quickly in the computer than BASIC. You can do many things more easily with PASCAL than with BASIC, but it is a more complicated language to learn.

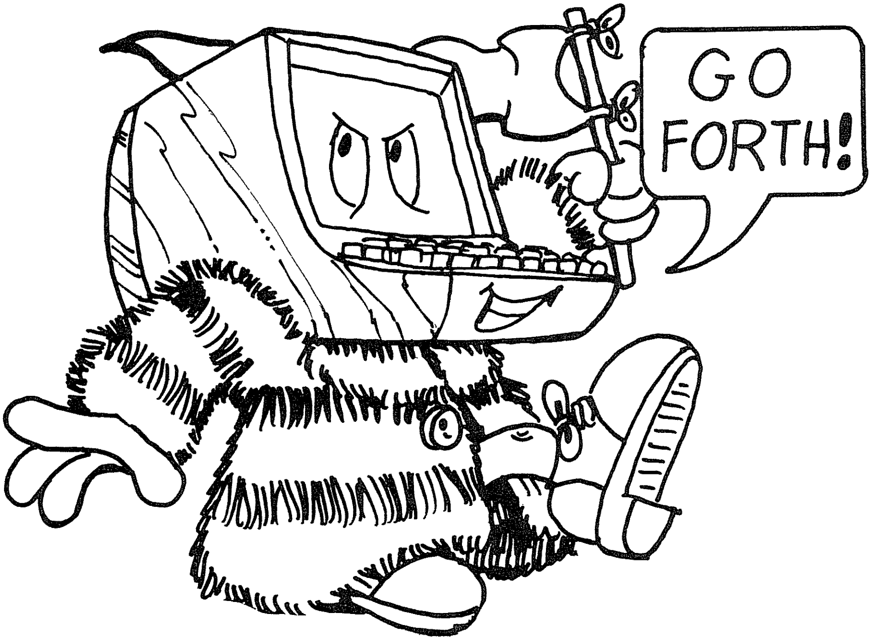
A PASCAL program that instructs the computer to print the squares of the numbers from 1 to 1Ø would look like this:

```
PROGRAM SquareNumber (output);
  VAR number:integer;
BEGIN
FOR number : =  1 to 1Ø DO
    Writeln (number * number);
END .
```

FORTH

Another language used to write many commercial microcomputer programs is FORTH. It resembles assembly language more than the other high level languages and, as a result, programs written in FORTH run very quickly. It differs from assembly language, however, because you may create new words that the language can understand—words that perform whatever you tell them to. Here is a sample FORTH program that would print HI on the screen five times when you run it.

```
: PRINTHI . " HI" ;
: 5HI 5 Ø DO PRINTHI LOOP ;
```



Other Languages

Some of the other languages you may hear about are mentioned below. Most of these languages are available on the big main frame computers, but may be rewritten for microcomputers in the future.

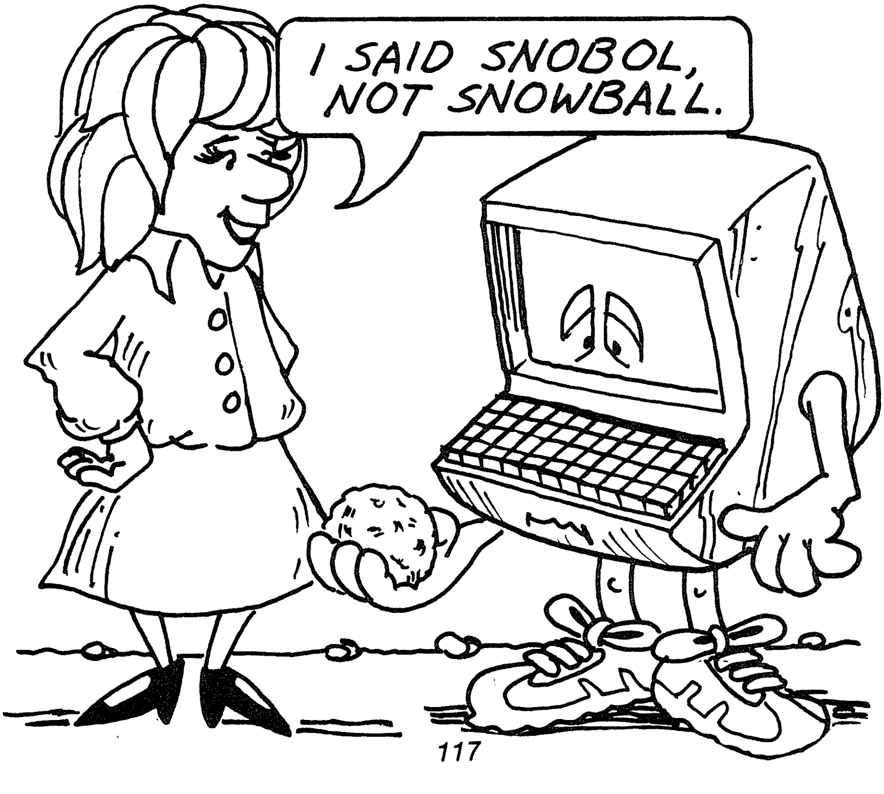
ALGOL, stands for ALGOrithmic Language. It is one of the first high level languages and is rather mathematical. It has many similarities to FORTRAN.

COBOL, COmmon Business Oriented Language, was written for business applications.

ADA is a new language used by the U.S. Department of Defense. It was named after Ada Lovelace, who was a pioneer computer programmer.

PL/1,(Programming Language#1) is an early computer language still used in scientific and engineering problems.

SNOBOL, StriNg-Oriented symBOlic Language, was written especially to handle words—"strings". It is used in artificial intelligence.



LISP is an acronym for LISt Processing and is very different from the others. LISP is the language from which LOGO was derived.

C is the language used to write the popular and powerful UNIX operating system.

Any computer language is just a set of rules for the computer to follow. Each language has its own distinctive style and philosophy. Each has a special purpose and an application for which it is best suited. Many languages were written by individual companies or for particular computers. FORTRAN and PL/1 were written by IBM, while SNOBOL was written at Bell Telephone Labs.

Any computer language is a powerful tool when you learn to use it. Practicing with the language most suited to your needs on the TRS-80 will increase your enjoyment of this exciting machine. The TRS-80 will do what you ask when you “speak” a language it understands.

GLOSSARY

algorithm A procedure for solving a problem or making a decision.

argument The variable or information that a function needs as information. For example, in the function ABS (X), X is the argument of the function.

Arithmetic Logic Unit (ALU) The part of the computer that does all the math in the computer.

assignment The storing of a value in the variable using the assignment instruction (variable name = value).

BASIC (Beginner's All Purpose Instruction Code) A programming language.

bug A programming term for an error or mistake.

call To execute a subroutine. A call implies that the subroutine will return control when it has finished its task so the program can continue with the next BASIC statement.

Cathode Ray Tube (CRT) A picture tube used to show information. One kind of output equipment.

Central Processing Unit (CPU) The part of the computer that interprets instructions and carries them out; the "core" chip of the computer.

character A letter, digit, or other symbol used to represent information (data) for the computer. All symbols that appear on a computer keyboard are characters.

complement (logical complement) The exact negation of a statement.

control unit The part of the computer that directs the flow of data or information through the computer.

cursor A square symbol that tells where on the video screen the next character will appear; the "place keeper".

data The information that is put into the computer to be processed by a program or that results when the program is RUN.

debugging The process of finding and correcting the mistakes in a computer program.

disk drive A machine that plays and records on disks and diskettes; one kind of input or output equipment.

diskette A storage material about 5¼" (13.34 cm) or 8" (20.32 cm) across with a magnetic surface for recording. Enclosed in a square, plastic, protective covering.

Disk Operating System (DOS) A set of computer programs that control other programs and allow for the transfer of information from and to a diskette.

editing The process of changing or correcting a program line.

execute To perform or carry out, as to execute (or RUN) a computer program.

flowchart A picture or little map using symbols to show the steps of a program.

function A small built-in program that allows special actions to be performed on a program and its data. Some functions handle strings and others handle numbers and the values of numeric variables.

graphics Pictures; a way of showing information by pictures rather than by words.

hardware The computer and its parts, as opposed to *software.*

incrementing Increasing the current value of a numeric variable (usually by one).

infinite loop A loop that has no way of ending.

information Organized data used in a computer program.

input Information put into the computer.

input equipment Any of the pieces of equipment used to put information into the computer such as a keyboard, disk drive, program recorder or modem.

instructions Directions for the computer to follow.

interpreter One of the programs in the computer's system that translates BASIC or another language into information the machine can understand.

jump A change in the usual order in which instructions are followed; a branch. For example, a computer may follow one set of instructions if a number is greater than 5 and another set of instructions if the number is less than 5. In BASIC this is signaled by a GOTO statement. It may also be implied in an IF statement.

keyboard The part of the computer that operates like a typewriter to give input to the computer.

language A set of words and other symbols used to give instructions to a computer.

line number The number in front of program instructions. The computer will normally follow instructions in order of line numbers.

LIST The command to the computer to give a listing of the program in use.

LOAD The command to the computer to take information into the memory unit.

logic The process of breaking down a problem in step-by-step order.

logic error A mistake in constructing a program so that it operates as intended.

logical operator One of the words—AND, OR, and NOT—that allow two or more conditions to be combined in a selection structure.

loop A set of instructions that repeats.

loop index The variable used in a loop that stores the number of times the loop has already been executed.

magnetic tape A plastic ribbon that has a chemical coating that can be magnetized. Information can be recorded on magnetic tape and "fed" into the computer's memory.

memory The part of the computer that stores information.

menu A list of options, choices, from which the user may choose.

microchip A tiny circuit etched onto silicon crystal.

mode An operating state of a computer; a method of working.

modem A machine used to allow a computer to receive or send information over a telephone line.

nesting (loop nesting) The method of placing one program loop inside another.

operating system A program or set of programs in machine language that controls the execution of computer programs.

operation One of the basic functions (jobs) a computer can do, such as addition, subtraction, and movement of data.

output The information or results coming out of the computer.

precedence In a computer language, refers to the performance of one type of arithmetic or logical operation before others within the same statement. Among arithmetic operations, exponentiation has the highest precedence, followed by multiplication and division (same precedence), and finally, addition and subtraction (same precedence).

PRINT A statement to the computer to put something on the video screen.

process A systematic order of operation followed to produce a certain result. When the computer adds or otherwise performs operations on numbers, it is *processing* the numbers.

program The set of instructions that tell the computer what to do step by step; to write the instructions that tell the computer what to do.

programmer A person who writes a program, instructions, for a computer to follow.

Random Access Memory (RAM) The part of the computer's memory that stores information temporarily. The computer user puts information into this memory.

Read Only Memory (ROM) The part of the computer's memory that contains built-in instructions. It cannot be used for storage of programs by the person using the computer.

REM A program statement short for remark.

RUN The command to the computer to start or begin.

SAVE The command to the computer to record information to use another time.

selection (selection structure) An instruction to the computer to decide which action (or actions) to perform, depending on whether one or more conditions are satisfied.

software The programs and instructions that go into a computer.

string A series of characters. A string may contain any combination of letters, numbers or punctuation.

subroutine A self-contained set of instructions used outside (after the end) the main program.

syntax error An error caused by a mistake in the way a command or information is given to the computer.

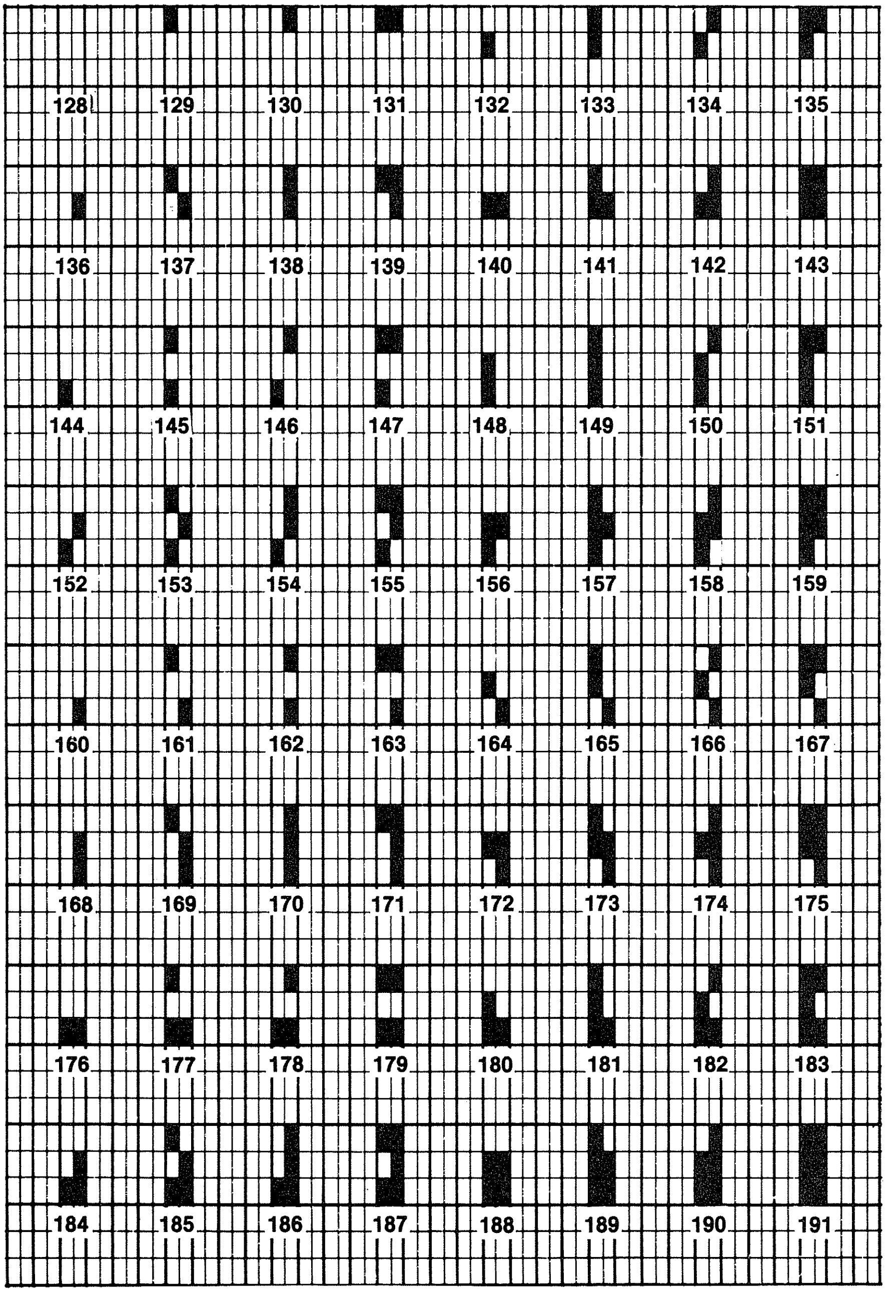
system A collection of people, machines, and methods organized to meet certain goals.

system program One of the computer programs that come with the computer. A system program "keeps house" to allow other programs to be written, stored, and retrieved.

text The writing in a program, as opposed to the graphics.

variable A part of the program that is changeable or may have different values.

video monitor A machine with a CRT screen for showing information; one kind of output equipment.


128 129 130 131 132 133 134 135
136 137 138 139 140 141 142 143
144 145 146 147 148 149 150 151
152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183
184 185 186 187 188 189 190 191

$8.95

NEW

Computer Books

from

ENRICH/OHAUS®

THE GOOD IDEA PEOPLE

Finally, microcomputer handbooks you can understand. Written in everyday language for the beginning beginner, these handbooks . . .

- Start at the absolute beginning
- Provide the information to make you "Computer Literate"
- Guide you as you discover what your computer can *really* do
- Give you "hands-on" experiences so that you can write your own programs
- Put you in complete control

FUNCTIONAL TOO! The built-in easel allows these books to stand up at your computer for easy use!

Other computer books from ENRICH:

The Illustrated Computer Dictionary & Handbook $9.95

Free Software for Your ATARI $8.95
Free Software for Your Commodore $8.95
Free Software for Your Apple $8.95
Free Software for Your TI $8.95

Apple for the Beginning Beginner $8.95
ATARI for the Beginning Beginner $8.95
PET for the Beginning Beginner $8.95
TRS-80 for the Beginning Beginner $8.95
TI for the Beginning Beginner $8.95

Ask for these books wherever good books are sold!

0 30531 79224

Printed in the U.S.A.
ISBN: 0-86582-121-6